Mälardalen University Licentiate Thesis
No.155

On the Development of Hierarchical Real-Time Systems

Mikael Åsberg

June 2012

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 987-91-7485-075-8
Printed by Mälardalen University, Västerås, Sweden

Populärvetenskaplig sammanfattning

Begreppet ”realtid“ används ofta i sammanhang där en uppgift genomförs i samma tempo som ett pågående informationsflöde. Ett exempel kan vara ett företag som övervakar sin egen verksamhet (inköp, leverans etc.) i realtid för att på så sätt kunna korrigera avvikelser innan det är för sent.

Inom det vetenskapliga området Datateknik har begreppet realtid (Real-Time Systems) en helt annan innebörd. I det föregående exemplet finns det inga direkt strikta krav på hur länge tidsfördröjningen kan vara från t.ex. ett inköp till att systemet notifierar detta till användaren (men man räknar med att det bör ske inom några minuter). En intressant observation är att det inte påverkar företaget negativt om det dröjer någon minut extra p.g.a. överbelastning i nätverket exempelvis. Med andra ord, man nöjer sig med en medelmåttig prestation av systemet i avseende på tid. Däremot har den funktionella delen av systemet hårda krav på sig, t.ex. att korrekt data ska presenteras för användaren.

Realtids system inom Datateknik kräver, utöver att systemen är funktionellt korrekta, att de även håller alla satta tidsgränser. Om ett system har 1000 olika tidsgränser, varav dessa ligger i ordningen på några mikrosekunder, så får aldrig en enda av dessa 1000 tidsgränser överskrida den satta tiden. En sådan försening skulle kunna leda till katastrof där t.o.m. dödsfall skulle kunna förekomma. Eftersom konsekvensen av en försening är så pass katastrofal, så analyserar man och testar dessa system noga. En viktig aspekt är att varenda del i systemet ska kunna analyseras med avseende på tid, så att förutsägelser kan göras innan systemet sätts i bruk. I många fall introducerar man feltoleranstekniker som hanterar situationer där en tidsgräns är överskriden, för att minska eller helt undvika en katastrof.

Ett av dagens stora mjukvaru/hårdvaru problem är att många system innehåller en stor och nära ohanterlig mängd datorer och nätverk mellan dessa. Ett exempel på en industri med dessa problem är fordonsindustrin. Problemet är att antalet funktioner ökar (anti-sladd, parkerings-assistans etc.) vilket leder till fler datorer. En bil är idag full av kablage och datorer vilket ökar vikt, kostnad och komplexitet.

Målet är att minska antalet datorer (och kablage) genom att låta realtidsmjukvara samsas på ett mindre antal datorer. Nya mjukvarustandarder för bilindustrin bygger på detta koncept. Dessa mjukvaror har strikta tidskrav som beskrivet ovan, vilket orsakar problem när dessa ska integreras. Integreringen är ett stort riskmoment eftersom tidsgränserna riskerar att brytas. Det kan därför bli en dyr och svår uppgift att lösa.

Avsikten med denna licentiatavhandling är att hjälpa till att lösa denna uppgift. I detta arbete använder vi oss av en teknik vid namn ”hierarkisk schemaläggning“ för att underlätta mjukvaruintegrering. Detta görs genom att partitionera mjukvaror i separata delar, vilket gör systemet säkert och lätt att analysera. Tekniken härrör från 60-talet men har ej applicerats i så stor utsträckning p.g.a. att den inte har anpassats för användning inom olika teknikområden.

Avhandlingen presenterar anpassningar av hierarkisk schemaläggning för att göra den mer användbar. Vi presenterar en teknik som möjliggör att köra en realtidsmjukvara i en partition på ett operativsystem i prototypsyfte. Detta kan ses som en förstudie för att testa en mjukvarupartition utan att behöva implementera och exekvera alla andra mjukvarupartitioner samtidigt. All programmeringskod genereras automatiskt och passar de flesta realtidsoperativsystem, inklusive Linux (standardversionen). Vi har även utvecklat två stycken partitionsschemaläggare. Den ena är skriven manuellt för realtidsoperativsystemet VxWorks. I syfte att kontrollera dess korrekthet har en program-spårare utvecklats speciellt för denna typ av schemaläggare. Den andra versionen är modellerad och verifierad med tidsautomater. Dess programkod är automatiskt genererad från modellen och passar de flesta realtidsoperativsystem. Dessa två versioner har en avvägning vad det gäller prestanda gentemot korrekthet (både tidsmässigt och funktionellt). Den verifierade schemaläggaren har sämre prestanda än den manuellt utvecklade varianten. I det fortsatta arbetet ingår det bl.a. att förbättra prestandan för verifierade schemaläggare.

Abstract

Hierarchical scheduling (also referred to as resource reservation) is a hot topic within the research of real-time systems. It has many advantages including that it can facilitate software integration, fault isolation, structured analysis, legacy system integration etc. The main idea is to partition resources (processors, memory, etc.) into well defined slots. This technique is rarely used in the most common real-time applications; however, it is used in the avionics industry to isolate error propagation between system parts, and to facilitate analysis of the system.

Much of the research within resource reservation deals with theoretical schedulability analysis of partitioned systems, including shared resources (other than the processor). We will in this thesis address more practical issues related to resource reservation. We focus on implementation and prototyping aspects, as well as verification and instrumentation. One of our assumptions is that we deal only with fixed-priority pre-emptive scheduling (FPPS).

The first part in this thesis deals with individual software systems that may have its own tasks as well as a scheduler and it is assumed to be part of another larger system, hence, we refer to this individual system as a subsystem. The subsystem is assumed to be integrated together with other subsystems, but at an early stage, we make it possible to simulate the subsystem running together with the rest of the subsystems. This ”simulation“ does not require the actual resource reservation mechanism, the only requirement is an operating system that supports FPPS. This pre-study may be a natural step towards the ”real“ integration, since each individual subsystem can be test-executed within its assigned partition. All subsystems are assumed to run together using a resource reservation mechanism (during the actual integration). We have developed two prototypes of this mechanism. The first prototype is hand-crafted and it is equipped with a program tracer for partitioned based schedulers. This instrumentation is useful for debugging and visualization of program traces for this

type of scheduling. The second prototype is developed using timed automata with tasks (task automata). This model-based scheduler is verified for correctness and it is possible to automatically generate source code for the scheduler. We have successfully synthesized this scheduler for the real-time operating system VxWorks. However, it can easily be extended for other platforms. Both prototypes have pros and cons. The first version has good performance while the second can guarantee its correctness; hence, there is a trade-off between performance and correctness.

Acknowledgments

The two most important people that got me engaged in PhD studies and that have supported me from master thesis up until now is my excellent supervisor Prof. Thomas Nolte and my wonderful (and former master thesis supervisor) colleague Dr. Moris Behnam. Special thanks also goes to my former study mates Alexander Casal and Amir Shariat for their encouragement during my undergraduate studies.

I owe a lot of gratitude to my co-authors Prof. Paul Pettersson and Dr. Shinpei Kato (Nagoya University). I cannot even put words into how helpful and supportive they have been. I would also like to thank Dr. Reinder 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), Dr. Johan Kraft and Dr. Insik Shin (Korea Advanced Institute of Science and Technology) for the interesting discussions and valuable feedback.

A bunch of teachers at IDT gave me a lot of inspiration and encouragement during my years as a undergraduate student. These teachers are simply great, big thanks to Åsa Lundkvist, Christer Sandberg, Anders Pettersson, Frank Lüders, Daniel Sundmark, Rikard Land, Ingrid Runnérus, Johan Stärner, Mohammed El Shobaki, Daniel Flemström, Jan Gustafsson, Jukka Mäki-Turja, Henrik Thane, Mats Björkman, Damir Isovic, Gordana Dodig-Crnkovic, Andreas Ermedahl and Dag Nyström.

I would also like to thank some of the professors for interesting discussions and PhD courses; Hans Hansson, Ivica Crnkovic, Sasikumar Punnekkat, Björn Lisper, Mikael Sjödin and Kristina Lundkvist.

My gratitude also goes to some of the administrative staff at IDT who makes life as a PhD student easier; Carola, Jenny, Susanne, Malin Å. and Else-Maj (who recently retired).

A lot of gratitude to the people that make it even more fun to go to work;

Aida, Aneta, Séverine, Nima, Rafia, Ana, Sara D., Adnan, Andreas H., Hüseyin, Christina, Tibi, Stefan By., Yue, Jagadish, Abhilash, Jan C., Nikola, Holger, Federico, Linus, Saad, Mehrdad, Svetlana, Juraj, Luka, Leo, Josip, Antonio, Rikard Li., Thomas Le., Hang, Barbara, Andreas G., Anna, Andreas J., Batu, Mobyen, Afshin, Shahina, Gunnar, Malin R., Fredrik, Jörgen, Lars, Carl, Radu, Giacomo, Elisabeth, Stefan C., Ella, Daniel H., Baran, Kivanc, Raluca, Eduard, Guillermo, Sara A., Mohammad, Shihong, and others. My warmest gratitude goes to the ones that left IDT but who brought a lot of joy; Farhang, Kathrin, Eun-Young, Rui, Etienne and Amine.

I have supervised three bachelor students; Nils, Per-Erik and Tim. I want to thank them for their contributions to my research and I wish them the best of luck.

I am grateful for the support and love from my parents; Arne and Joyce, and my siblings; Patrik, Anne, Jenny and Josefin. Lots of gratitude goes to the "big" Karlsson family, Lorena Åsberg and Pontus Einarsen.

Last but not least, lots and lots of gratitude to my wonderful girlfriend, Malin Karlsson, for the support, love and fun.

This work has been supported by the Swedish Foundation for Strategic Research (Stiftelsen för strategisk forskning) and the Swedish Research Council (Vetenskapsrådet).

Mikael Åsberg
Västerås, June, 2012

List of publications

Papers included in the licentiate thesis[1]

Paper A *Prototyping and Code Synthesis of Hierarchically Scheduled Systems using TIMES*. Mikael Åsberg, Thomas Nolte, Paul Pettersson. Journal of Convergence (FTRA), pages 77-86, December, 2010.

Paper B *Towards Hierarchical Scheduling in VxWorks.* Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, Reinder J. Bril. In 4^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08), pages 67-76, July, 2008.

Paper C *A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux*. Mikael Åsberg, Thomas Nolte, Shinpei Kato. In 17^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), pages 380-387, August, 2011.

Paper D *Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling*. Mikael Åsberg, Paul Pettersson, Thomas Nolte. In 23^{rd} Euromicro Conference on Real-Time Systems (ECRTS'11), pages 172-181, July, 2011.

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

Additional papers, not included in the licentiate thesis

Conferences and workshops

- *ExSched: An External CPU Scheduler Framework for Real-Time Systems*. Mikael Åsberg, Shinpei Kato, Thomas Nolte, Ragunathan Rajkumar. In 18^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12), pages 1-10, August, 2012.

- *Towards Hierarchical Scheduling in AUTOSAR*. Mikael Åsberg, Moris Behnam, Farhang Nemati, Thomas Nolte. In 14^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'09), pages 1181-1188, September, 2009.

- *Prototyping Hierarchically Scheduled Systems using Task Automata and TIMES*. Mikael Åsberg, Thomas Nolte, Paul Pettersson. In 5^{th} International Conference on Embedded and Multimedia Computing (EMC'10), pages 1-8, August, 2010.

- *Implementation of Overrun and Skipping in VxWorks*. Mikael Åsberg, Moris Behnam, Thomas Nolte, Reinder J. Bril. In 6^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10), pages 45-52, July, 2010.

- *A Loadable Task Execution Recorder for Linux*. Mikael Åsberg, Shinpei Kato, Johan Kraft, Thomas Nolte. In 1^{st} International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS'10), pages 31-36, July, 2010.

- *Towards Adaptive Hierarchical Scheduling of Real-Time Systems*. Nima Moghaddami Khalilzad, Thomas Nolte, Moris Behnam, Mikael Åsberg. In 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11), pages 1-8, September, 2011.

- *On Adaptive Hierarchical Scheduling of Real-Time Systems Using a Feedback Controller*. Nima Moghaddami Khalilzad, Thomas Nolte, Moris Behnam, Mikael Åsberg. In 3^{rd} Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'11), pages 1-6, April, 2011.

- *Overrun and Skipping in Hierarchically Scheduled Real-Time Systems*. Moris Behnam, Thomas Nolte, Mikael Åsberg, Reinder J. Bril. In 15^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09), pages 519-526, August, 2009.
- *Synchronization Protocols for Hierarchical Real-Time Scheduling Frameworks*. Moris Behnam, Thomas Nolte, Mikael Åsberg, Insik Shin. In 1^{st} Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08), pages 53-60, November, 2008.
- *Hierarchical Scheduling of Complex Embedded Real-Time Systems*. Thomas Nolte, Moris Behnam, Mikael Åsberg, Reinder J. Bril, Insik Shin. In École d'Éte Temps-Réel (ETR'09), pages 129-142, August, 2009.

Work in progress

- *Execution Time Monitoring in Linux*. Mikael Åsberg, Thomas Nolte, Clara M. Otero Pérez, Shinpei Kato. In Work-In-Progress (WIP) track of the 14^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'09), pages 1601-1604, September, 2009.
- *Towards Hierarchical Scheduling in Linux/Multi-Core Platform*. Mikael Åsberg, Thomas Nolte, Shinpei Kato. In Work-In-Progress (WIP) track of the 15^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'10), pages 1-4, September, 2010.
- *Towards Real-Time Scheduling of Virtual Machines Without Kernel Modications*. Mikael Åsberg, Nils Forsberg, Thomas Nolte, Shinpei Kato. In Work-In-Progress (WIP) track of the 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11), pages 1-4, September, 2011.

Technical reports

- *Comparison of Priority Queue algorithms for Hierarchical Scheduling Framework*. Mikael Åsberg. Technical Report, Nr. 2598, Mälardalen Real-Time Research Centre, Mälardalen University, October, 2011.
- *Model of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling*. Mikael Åsberg. Technical Report, Nr. 2379, Mälardalen Real-Time Research Centre, Mälardalen University, January, 2011.

Contents

I

Thesis

Chapter 1

Introduction

The increasing product competition and customer demand for more functionality in electronic products, such as consumer electronics [1] (smart phones), cars [2], aeroplanes [3, 4] etc., makes them more complex to develop. Take a mobile phone for example, which as of today not only has the ability to make phone calls but it can also navigate, take photos, browse the internet etc. Due to the demand on its limited size, vendors can not afford to increase the amount of hardware. However, the amount of software increases rapidly since this gives rise to more functionality. Car vendors experience the same kind of problems, i.e., more functionality in the form of selective shock absorber, steering assistance, electronic stability programme, braking assistance, parking assistance, navigation etc. Due to weight and volume demands, the amount of onboard computers, referred to as Electronic Control Unit (ECU), as well as connecting cables need to be reduced. The difference, as compared to mobile phones, is that much of the software executed on ECUs have real-time requirements with strict time deadlines. An example of such an application is an airbag, which can cause severe human casualty if not executed correct according to time. The problem of integrating a rapidly increasing amount of software on a steadily decreasing amount of hardware, without violating extra functional properties (time deadlines), is not completely solved yet but under progress using new standards such as the automotive standard AUTOSAR [2]. Similar problems are solved in the aerospace industry through the ARINC653 standard [3, 4]. ARINC653 adapts to partitioning of systems to solve integration related problems. Techniques similar to ARINC653 are also used to get predictability and composability in hardware such as memory controllers [5].

We believe that system partitioning techniques has the potential to solve integration related problems in industry but it has not gotten any foothold yet. This is most likely because the technique itself brings some complexity/problems which are also needed to be solved. Another reason could be the difficulty to adapt it to fit with operating systems, standards, development processes etc. in industry. Hence, the intention with this thesis is to develop the partitioning technique, which we refer to as hierarchical scheduling, in different ways. One of our goals is to maintain total isolation and separation between partitions; hence, we try to avoid dependencies between partitions. This property is worth to maintain since it can facilitate independent development, verification, analysis, tests etc. prior to system integration. This increases safety and it is less costly since partitions will not affect each other during the development and integration phase.

1.1 Contributions

The main contributions of this thesis are as follows.

1. **Prototyping**
   We have proposed a technique that can create a periodic partition using only a set of periodic tasks. What is needed to run the partition is a task scheduler that can schedule periodic tasks with offsets. We have used the TIMES [6] tool to generate such a scheduler automatically once the task parameters are set. We introduce an algorithm that can generate these parameters. In this way, a partition can be executed on a platform (which has support for periodic releases of tasks) and get the same interference as it would in the final integrated platform. Hence, this conform to independent development of partitions and suits well for prototyping partitions. Paper A directs this contribution.

2. **Implementation**
   We present an implementation of a two-level partition scheduler in one of the most popular real-time operating systems: VxWorks. We give details on implementation aspects and performance measurements. This implementation is presented in Paper B. We have also developed a recorder that is specialized for debugging partitioned systems. This instrumentation does not require any kernel modifications but still it performs well compared to existing recorders. Our recorder is presented in Paper C.

3. **Modelling and Verification**
   A second version of an implementation of a two-level partition scheduler is presented in Paper D. The difference, as compared to Paper B, is that this scheduler is modelled and verified using timed automata and logics. The partitions are verified independently of each other, which conforms to our goal of independent partition development. The last step, after verifying each partition scheduler, is the verification of the entire system, i.e., the global scheduler. Modularized verification is useful since modifications to a part of the system do not require re-verification of the entire system. We have also synthesized the scheduler for VxWorks and measured its performance and compared it to the scheduler in Paper B.

1.2 Thesis outline

The outline of the thesis is as follows. Chapter 2 presents theory of real-time systems, hierarchical systems and operating system scheduling. In Chapter 3 we give an overview of the research presented in this thesis. In Chapter 4 we present our conclusion and future work. The technical overviews of the papers that are included in this thesis are presented in Chapter 5, and we present these papers in Chapters 6 - 9.

Chapter 2

Background

2.1 Real-time systems

A real-time system can be defined as a computer system (including both hardware and software) that has strict demands on timing [7]. Most often, these requirements require tasks to finish their execution before a pre-defined point in time. Real-time systems have not only demands on the functional correctness, but also demands on that functions within these systems should be guaranteed to complete within exact time boundaries. The guarantees that these systems must provide are normally 100% guarantees, i.e., they **may not ever exceed these deadlines**.

A system is usually divided into several software parts called *tasks* which execute a sequence of operations. These tasks execute in parallel, either on a single processor in which case they interleave each other (called *pseudo parallelism*), or on multiple processors in which case they execute at the same time in parallel. Each task has special attributes related to the timing demands and these are used in analysis [8] (calculations) in order to check that all tasks timing demands are met before executing them together. The timing attributes can consist of a *deadline* which is the latest point in time when the task should finish its execution (delays that other parts of the system causes usually affects the actual finishing time of the task) and *Worst Case Execution Time* (WCET) which is an analyzed maximum value of time that a task needs in order to complete its execution. Tasks may be triggered to execute based on time points or other events. In this case it executes (not more than specified in its WCET) and then waits for its next activation time. If the task gets triggered at every

fixed interval of time *period*, then we call this task a periodic task [9]. If a task gets triggered with a bounded minimum interval of time (but possibly a larger interval) then the task is referred to as *sporadic*. A task that gets triggered at any arbitrary time instant is referred to as an *aperiodic* task.

There are two main categories of real-time systems; *hard real-time systems* and *soft real-time systems* [7]. Tasks in hard real-time systems are **never** allowed to miss their deadlines. Soft real-time systems on the other hand can tolerate some deadline misses. A system referred to as a *safety-critical system* is a type of hard-real time system which can lead to catastrophic incidents if any task deadlines are missed. Examples of products that contain such safety-critical systems are cars, aeroplanes, medical devices etc.

2.2 Hierarchical systems

The *task* is often the entity in which a system is divided into, i.e., the tasks together form the system [7]. System requirements which deal with temporal aspects (deadlines) of the functionality, called *non-functional* properties (as opposed to *functional* properties which is related to the functionality of the system), can be fulfilled by having a set of tasks with "correct" timing attributes that meet these requirements. The functional requirements are checked with respect to what operations the tasks perform.

The reason why the task has become the natural entity to divide a system into is mainly due to how operating systems are built. Operating systems are the backbone of most real-time systems with respect to the software. For example, *UNIX* based operating systems have an internal architecture which is based on *processes* [10, 11]. The process can be viewed as a task. These processes execute applications and internal operating-system operations. All of the functionality of an UNIX-based operating system resides in all of the processes that run within the operating system. It is usual that vendors use operating systems (e.g. UNIX) in their products and adapt their system to it, considering that UNIX-based operating systems are widely used in PCs (GNU/Linux and Mac OS/X), mobile phones (Android and iOS), industrial applications (Embedded Linux, RTLinux, VxWorks) etc. Hence, it becomes natural to divide a system into entities that are found in the operating system.

There exist products (with real-time systems) where the vendors of the product (and the industry itself) have realized that it can be more safe and easier to develop software systems with a more coarse-grained division of the system than just using tasks. Such an example is the aerospace indus-

try. They even created a standard/specification related to this issue, they call it ARINC653 [3, 4] (Avionics Application Standard Software Interface). ARINC653 specifies that a system should be divided into *partitions* (instead of just tasks) and that these partitions have separate memory and time allocations. Within each partition, tasks may execute in the same fashion as they have been doing in a regular operating system context. In this way, we get two levels of abstraction; high level functions can be divided and separated into different partitions giving rise to protective boundaries of each other which is safer; each high-level function, in the second level, can execute its own tasks separate from other partitions. We call this a *hierarchical system*.

Operating system vendors, such as WindRiver, have adapted their operating system to the aerospace industry and the ARINC 653 standard. This is natural since the aerospace industry has important customers for companies such as WindRiver. WindRivers most sold operating system is called VxWorks and it is widely used in many different industries. The VxWorks operating system has a special version called VxWorks 653 which is adapted for the ARINC 653 standard. This is a perfect example were we can see how a trend of hierarchical software systems has emerged from an industry and later transferred to the operating system which they use.

However, this is unfortunately a special case. We can see some efforts being done in the GNU/Linux operating system, however, the results are limited so far (*Control Groups*). The real-time systems research-community is working hard and pushing to adopt GNU/Linux to the partitioning scheme (SCHED_EDF/SCHED_DEADLINE [12, 13]), however, it is not quite there yet.

2.3 Operating system scheduling

This Licentiate thesis has a strong focus on the practical side of operating system scheduling, i.e., almost all included research articles in this thesis present an implemented scheduler. Hence, this section will shed some light on operating system scheduling.

Most operating systems, e.g. Microsoft Windows, VxWorks etc., have an important software component called *scheduler*. The duty of the scheduler is to schedule the *tasks*, i.e., choose in which order they should execute on the processor. If there are more tasks than available processors to execute them on, then they have to share the processor. The scheduler's job is to schedule the tasks according to some rule such that all tasks execute for some amount of

time that complies with this rule. For example, the GNU/Linux scheduler will execute most processes in a fair way, i.e., in a way that they all get an equal amount of time of the processor. This is done by running each process for a small amount of time (in the order of milli seconds), and then switch to another process etc. All processes are stored in a list and the scheduler executes each of them in turn. When reaching the end of the list, the scheduler starts all over again in the beginning of the list. This type of scheduling is called *round robin* and it is a type of *fair scheduling*. A process in GNU/Linux has a so called *priority* associated with it. Most operating systems have a priority for their corresponding task. A higher priority value means that a task is more important, hence, the scheduler can choose the next task to run based on the tasks priority. GNU/Linux has two levels of priority: *fair* and *real-time* priority. Tasks with real-time priority will always have higher priority to the processor wrt fair priority tasks. In turn, a task with higher real-time priority will monopolize the processor wrt lower real-time priority tasks. If the priority of a task does not change during its life-span, then we say that the scheduler schedules according to the *fixed-priority* scheduling algorithm. If the scheduler re-assigns task priorities during runtime, then it implements *dynamic-priority scheduling*. If the scheduler *activates* a task, it means that the task is eligible to execute on a processor, but it will only run if it has the highest priority of all *active* tasks. If the scheduler activates a task periodically, then we refer to this as scheduling of *periodic tasks* [9]. Note that all schedulers can not schedule tasks periodically, for example, GNU/Linux and VxWorks has no native support for this.

2.4 Assumptions of the thesis

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

Real-Time Systems:
We assume hard real-time systems for the main part of this research, although some parts relate to soft real-time.

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

Scheduling Protocol:
We assume fixed-priority scheduling.

Synchronization Protocol:
This work does not account for resource sharing between tasks or partitions. However, our future work will include resource sharing.

Chapter 3

Development of hierarchically scheduled systems

This chapter presents the main idea of this thesis and connects the different research results into a coherent story.

3.1 Independent subsystem development

Assume that software systems development will strive towards a partition based software division. As we described in Chapter 2, Section 2.2, aerospace has already adopted to this mechanism. In this scenario, it is likely that partitions, which we also refer to as subsystems, will be part of a software system. This software system may itself form another subsystem and so on. For example, the sensor software in an Anti Lock Brake System (ABS) forms a subsystem in the ABS application, while the ABS system itself can be considered as a subsystem in a vehicle. Independent of which level in this hierarchy we are focusing on, subsystems will most likely be developed by different development teams residing in different locations in the world, in different companies etc. Take a car for example, the Original Equipment Manufacturer (OEM), i.e., the company that produces the final product (Volvo, BMW, Honda etc.), does not develop and produce all subsystems in their cars. Subcontractors, for example

BOSCH, are responsible for delivering subsystems, e.g., an ABS system. One of the big challenges in software development is the software integration where all software parts, i.e., subsystems, are integrated. This challenge has become such a huge problem that the automotive industry has developed a software-architecture standard called AUTOSAR [2] in order to tackle integration related problems. One of the concerns is how the non-functional properties of subsystems will change (and perhaps violate requirements) once you integrate them. An ARINC based approach solves timing and memory related problems at integration phase, i.e., non-functional properties can be preserved. However, the first part of this thesis (paper A) will focus on the development phase **prior** to the integration of subsystems where an operating system like VxWorks 653 is needed to execute them together in a safe manner. The main idea that we have is to facilitate so that subsystem developers can execute their subsystem, given timing parameters such as period etc., in an environment which gives the illusion that their subsystem executes together with the rest of the system. This emulation can be done in most real-time adapted operating systems (including GNU/Linux) without the need for the actual mechanism like in VxWorks 653. The assumption here is that there is no communication between subsystems and it is limited to fixed-priority scheduling. Once the subsystem has been tested with this technique, it will behave exactly the same (wrt to time) when it is integrated together with other subsystems, assuming that the system parameters are the same.

3.2 Operating system mechanism for supporting hierarchical systems

We have developed two operating system schedulers; the first one is developed for VxWorks and the second can be considered as platform independent. These two implementations give the mechanism support for hierarchical systems in operating systems. The reason for developing two new schedulers (paper B and D), even though similar schedulers already exist in GNU/Linux and VxWorks 653 for example, is because we want to extend the scheduler functionality and property compared to existing solutions. The advantage with developing our own schedulers is that we can measure their overhead easier which is interesting since the increase in overhead is a drawback with partitioned scheduling. We can test more advanced scheduling schemes, i.e., we support both fixed and dynamic priority scheduling in all levels. The scheduling schemes that we have developed has theoretical research behind it [14, 15, 16]. Our schedulers are

also easy to adapt to resource sharing which is an interesting line of research. The second scheduler that we have developed has the interesting property that it has been mathematically verified, i.e., we can guarantee its correctness wrt its specification. Moreover, it has a simple structure making it easily adaptable to GNU/Linux, VxWorks etc.

We always implement our schedulers with the intention that they should never require modifications to the operating system. This is an important property in industry in order to preserve stability in the operating systems and to avoid tedious updates to the scheduler when new versions of the operating system are released.

3.3 Testing and verification

Our last theme for this thesis is related to testing and verification of partitioned schedulers (paper C and D). We have developed a tool which can record the execution of tasks and subsystems; hence, we can then analyze the behaviour of the scheduler. This is a useful tool for schedulers which have not been analyzed wrt correctness, i.e., verified. The recorder is based on a platform independent framework which makes it possible to record tasks and subsystems on any operating system for which the framework has support for. We have developed this framework as well and currently it supports VxWorks and GNU/Linux. The recorder is compatible with the (partitioned scheduler) trace visualization tool Grasp [17].

We have also developed a method for verifying two-level partitioned schedulers using the timed automata language. Verification and certification of software (including schedulers) is very important in industries which have rigorous safety standards to follow such as ISO 26262 which is related to the automotive industry.

3.4 Summary

In this chapter we presented an overview of the contributions of the thesis. The first part relates to independent subsystem development prior to the integration phase. The second part of the thesis contribution relates to the practical implementations of two different schedulers. The final part relates to testing and verification of these two schedulers.

Chapter 4

Conclusions

4.1 Summary

In this thesis we have proposed techniques to aid in the development of hierarchical real-time systems. We have partly focused on the pre-integration phase where each subsystem developer may develop and test their subsystem in isolation of other subsystems without any partitioned scheduling support in the operating system.

The main part of this thesis relates to the practical implementations of partitioned schedulers.

Finally, we have implemented a tool (together with a framework) which can record traces of partitioned schedulers, and hence, make it possible to debug such schedulers. We have also proposed a technique to model and verify two-level hierarchical schedulers.

4.2 Future work

In the future we plan to extend our work by including logical resource sharing between subsystems. We will also develop a platform/scheduling independent framework which can aid in the development of any real-time scheduler. Another interesting line of work, connected to this thesis, is to improve the code synthesis (of schedulers) in terms of its runtime efficiency. This will also have a great performance impact if the target is to synthesize the code for other hardware platforms such as Graphics Processing Units (GPUs). In this line of work we aim at running the scheduler on the GPU which makes it possible to run schedulers with much more complex behaviour than state-of-the-art scheduling algorithms such as FPS and EDF.

We already have some preliminary research results aiming at connecting resource reservation with AUTOSAR. We see that AUTOSAR fits well with partitioning; hence, there is a potential future work within this area as well.

Chapter 5

Overview of papers

5.1 Paper A

Mikael Åsberg, Thomas Nolte and Paul Pettersson. *Prototyping and Code Synthesis of Hierarchically Scheduled Systems using TIMES*. Journal of Convergence (FTRA), pages 77-86, December, 2010.

Summary In hierarchical scheduling a system is organized as a tree of nodes, where each node schedules its child nodes. A node contains tasks and/or subsystems, where a subsystem is typically developed by a development team. Given a system where each part is subcontracted to different developers, they can benefit from hierarchical scheduling by parallel development and simplified integration of subsystems. Each team should have the possibility to test their system before integration. Hence, we show how a node, in a hierarchical scheduling tree, can be analyzed in the Times tool by replacing all interference from nodes with a small set of higher priority tasks. We show an algorithm that can generate these tasks, including their parameters. Further, we use the Times code-generator, in combination with operating system extensions, to generate source code that emulates the scheduling environment for a subsystem, in an arbitrary level in the tree. Our experiments include two example systems. In the first case we generate source code for an industrial oriented platform (VxWorks) and conduct a performance evaluation. In the second example we generate source code that emulates the scheduling environment for a video application, running in Linux, and we perform a frame-rate evaluation.

My contribution The basic idea of this paper was suggested by Thomas Nolte. Mikael Åsberg was responsible for conducting the experiments and writing the paper.

5.2 Paper B

Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg and Reinder J. Bril. *Towards Hierarchical Scheduling in VxWorks*. In 4^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08), pages 67-76, July, 2008.

Summary Over the years, we have worked on hierarchical scheduling frameworks from a theoretical point of view. In this paper we present our initial results of the implementation of our hierarchical scheduling framework in a commercial operating system VxWorks. The purpose of the implementation is twofold: (1) we would like to demonstrate feasibility of its implementation in a commercial operating system, without having to modify the kernel source code, and (2) we would like to present detailed figures of various key properties with respect to the overhead of the implementation. During the implementation of the hierarchical scheduler, we have also developed a number of simple task schedulers. We present details of the implementation of Rate-Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Finally, we present the design of our hierarchical scheduling framework, and we discuss our current status in the project.

My contribution The idea of this paper was suggested by Moris Behnam. Moris Behnam was the main driver in writing the paper. Mikael Åsberg was responsible for the implementation and evaluation of the scheduler proposed in this paper.

5.3 Paper C

Mikael Åsberg, Thomas Nolte and Shinpei Kato. *A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux*. In 17^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), pages 380-387, August, 2011.

Summary This paper presents a Hierarchical Scheduling Framework (HSF) recorder for Linux-based operating systems. The HSF recorder is a loadable kernel module that is capable of recording tasks and servers without requiring any kernel modifications. Hence, it complies with the reliability and stability requirements in the area of embedded systems where proven versions of Linux are preferred. The recorder is built upon the loadable real-time scheduler framework RESCH (REal-time SCHeduler). We evaluate our recorder by comparing the overhead of this solution against another (patched) recorder. Also, the tracing accuracy of the HSF recorder is tested by running a media-processing task together with periodic real-time Linux tasks in combination with servers. The tests are recorded with the HSF recorder, and the Ftrace recorder, in order to show the correctness of the experiments and the HSF recorder itself.

My contribution Mikael Åsberg was the main driver in writing the paper and performing the experiments.

5.4 Paper D

Mikael Åsberg, Paul Pettersson and Thomas Nolte. *Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling*. In 23^{rd} Euromicro Conference on Real-Time Systems (ECRTS'11), pages 172-181, July, 2011.

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.

My contribution Mikael Åsberg was the main driver in writing the paper and conducting the modelling, verification and synthesis.

Bibliography

[1] D. Andrews, I. Bate, T. Nolte, C. M. Otero Pérez, and S. M. Petters. Impact of Embedded Systems Evolution on RTOS Use and Design. In *1st International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 13–19, July 2005.

[2] T. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Mate, and K. Nishikawa. AUTOSAR - Challenges and Achievements. In *12th International VDI Congress Electronic Systems for Vehicles*, Oct 2005.

[3] ARINC. *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), 1996.

[4] ARINC/RTCA-SC-182/EUROCAE-WG-48. Minimal Operational Performance Standard for Avionics Computer Resources. 1999.

[5] Benny Åkesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. Composability and Predictability for Independent Application Development, Verification, and Execution. In *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*. Springer, Dec 2010.

[6] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A Tool for Modelling and Implementation of Embedded Systems. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464, April 2002.

[7] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, 2004.

[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Journal of Software Engineering*, 8:284–292, 1993.

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

[10] S.N. Bokhari. The Linux operating system. *Journal of Computer*, 28(8):74–79, Aug 1995.

[11] J. Wiegand. The cooperative development of Linux. In *29th IEEE Professional Communication Conference*, pages 386–390, Oct 1993.

[12] D. Faggioli, M. Trimarchi, and F. Checconi. An implementation of the Earliest Deadline First algorithm in Linux. In *24th Annual ACM Symposium on Applied Computing*, pages 1984–1989, March 2009.

[13] D. Faggioli and F. Checconi. An EDF Scheduling Class for the Linux Kernel. In *11th Real-Time Linux Workshop*, Sep 2009.

[14] Rob Davis and Allan Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *26th IEEE International Real-Time Systems Symposium*, pages 389–398, Dec 2005.

[15] T.-W. Kuo and C.H. Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *20th IEEE International Real-Time Systems Symposium*, pages 256–267, Dec 1999.

[16] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *24th IEEE International Real-Time Systems Symposium*, pages 2–13, Dec 2003.

[17] Mike Holenderski, Martijn Heuvel, Reinder Bril, and Johan Lukkien. Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 37–42, July 2010.

II

Included Papers

Chapter 6

Paper A: Prototyping and Code Synthesis of Hierarchically Scheduled Systems using TIMES

Mikael Åsberg, Thomas Nolte and Paul Pettersson
In Journal of Convergence, pages 77–86, December, 2010

Abstract

In hierarchical scheduling a system is organized as a tree of nodes, where each node schedules its child nodes. A node contains tasks and/or subsystems, where a subsystem is typically developed by a development team. Given a system where each part is subcontracted to different developers, they can benefit from hierarchical scheduling by parallel development and simplified integration of subsystems. Each team should have the possibility to test their system before integration. Hence, we show how a node, in a hierarchical scheduling tree, can be analyzed in the Times tool by replacing all interference from nodes with a small set of higher priority tasks. We show an algorithm that can generate these tasks, including their parameters. Further, we use the Times code-generator, in combination with operating system extensions, to generate source code that emulates the scheduling environment for a subsystem, in an arbitrary level in the tree. Our experiments include two example systems. In the first case we generate source code for an industrial oriented platform (VxWorks) and conduct a performance evaluation. In the second example we generate source code that emulates the scheduling environment for a video application, running in Linux, and we perform a frame-rate evaluation.

6.1 Introduction

The increase in global competitiveness and requirement of shorter time-to-market has increased the need for rapid development of embedded software systems. A crucial characteristic, in being fast and reliable in the development of embedded software systems, is to do analysis and prototyping early in the development process, in order to decrease the load, complexity and cost in the integration phase.

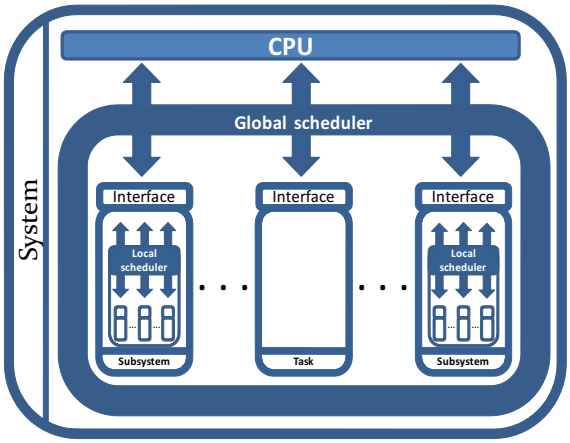

Figure 6.1: Hierarchical scheduling.

Recently, the technique of hierarchical scheduling (HS) has been introduced in order to simplify parallel development of embedded systems. HS facilitates integration of such systems, by providing mechanisms for temporal isolation of system parts, called subsystems. Essentially, a system consists of a number of subsystems that typically represents a particular function/feature of the whole system. For example, a car could have one subsystem implementing a engine control system, and another being the anti-lock braking system. These two subsystems should ideally be developed in parallel, and at the integration phase, no integration related problems should occur [1]. One such integration related problem is software that turn out to require more time to execute than originally intended, and therefore causing unforseen interference with the rest of the system. Another integration problem is the introduction of new subsystems, not apparent at early design. Integration of unforseen subsystems should not cause too much interference, i.e., the entire system should not be required

to be verified/validated again. HS insures that no unpredictable interference will occur, related to timing, hence by allowing for timing analysis of subsystems in isolation before the integration. Figure 8.1 illustrates HS. The top node is defined as the *Global scheduler*. It is responsible for multiplexing the entire *CPU* resource to the second layer of the scheduling tree. A node can be either a *Subsystem*, or a *Task* (except for the top node which is a scheduler). In this way, a node schedules its child nodes with its *Local scheduler*. All nodes have an *Interface* (set of scheduling parameters) which specifies the amount of CPU that the node may access. The schedulers uses these interfaces to schedule its nodes.

It is desirable to be able to conduct analysis of a subsystem's functional and non-functional properties in isolation, i.e., without requiring details of the rest of the system. It is hard to get access to all details of other subsystems, especially at an early stage in the construction of a system. Our proposed technique makes it possible to perform schedulability analysis of tasks, with respect to its subsystem interface. Also, the subsystem can be realized by generating source code (for our target platforms VxWorks and Linux) that will emulate the subsystem (under development) executing together with other subsystems/tasks. The subsystem's schedule will look like it is executing together with the other subsystems in the tree (early prototyping). What is required are the interfaces of the other subsystems/tasks, i.e., no subsystem internal data such as task source code, execution time, period etc. are needed. Also, there is no need to implement any scheduler. The internal scheduler of the Times tool is responsible for the schedulability analysis, and the generated source code will emulate the scheduler(s) in the system.

Recently, automata based techniques have been proposed as a generic way to describe and analyze a broad variety of real-time scheduling algorithms. One of the strengths of these techniques is the possibility to encode general release patterns of tasks. In the task automata model [2], release patterns are modeled using timed automata [3]. The schedulability analysis problem has shown to be decidable for both fixed and dynamic priority scheduling algorithms. Further, this approach has the possibility to perform simulation and formal verification of timing and functional safety properties, as well as code-synthesis [4]. For the model of task automata, the Times tool provides this support [5].

In this paper our overall goal is to provide a technique for analysis and synthesis of hierarchically scheduled real-time systems, at an early stage in the development process. Our main contributions are[1]:

[1] This work is an extension of our previous work [6]

1. We have enabled timing analysis of hierarchically scheduled, fixed-priority preemptive systems, in the Times tool.

2. We have transformed and made extensions to the generated source code (from Times) for VxWorks and Linux, allowing for early prototyping/testing of hierarchically scheduled, fixed-priority, preemptive systems.

3. Related to the above contribution (2), we have conducted experiments on the generated code (for both VxWorks and Linux). We have included response time measurements, overhead measurements of both the generated scheduler, and a manually coded scheduler, and we have compared these. Also, we have been running a video processing application (VLC) in Linux, and conducted frame-rate performance comparisons using a 2-level hierarchical scheduler, as well as task tracing.

The outline of the paper is as follows: in Section 9.2 we outline preliminaries on hierarchical scheduling, task automata and Times. In Section 6.3 we outline the problem statement including its limitations, and in Section 6.4 we show our solution. Section 6.5 shows two case-studies, including an example system, code generation and a performance evaluation. Section 9.6 presents related work, and finally, Section 9.7 concludes.

6.2 Preliminaries

6.2.1 Hierarchical scheduling

Hierarchical scheduling has been introduced to facilitate resource sharing among applications under different scheduling policies. Hierarchical scheduling can be represented as a tree of nodes (Figure 8.1), where each node corresponds to an application, equipped with a scheduler that schedules internal workloads. Looking at the tree-structure representation of HS, CPU resources are reserved from a parent node to its children nodes (Shin and Lee [7]). One of the advantages of HS is that it provides a way to decompose a complex system into well-defined parts (subsystems). HS provides the mechanism for predictable composition (in the time domain) of coarse-grained subsystems. This makes it possible for subsystems to be developed independently and later integrated, without introducing timing errors. Also, HS makes it easy to reuse subsystems, since their computational demands are characterized by well defined interfaces.

Subsystems and tasks are scheduled according to the scheduling scheme of the above scheduler and the parameters in the interface of the subsystem.

In this paper, we assume that the schedulers follow the fixed-priority preemptive scheduling policy. Subsystems can be viewed as ”virtual tasks”, where the interface parameters corresponds to those in the periodic task model [8]. At runtime, subsystems reserve a defined time (*budget*) at every *period* and the execution order is based on their *priority*. This is similar to a traditional periodic task, scheduled preemptively with a fixed-priority scheduler. When a subsystem is selected for execution by the overlaying scheduler, the subsystem's tasks are executed and scheduled according to the scheduling policy of the subsystem local scheduler. In the general case, the schedulers in HS may all have different scheduling schemes.

6.2.2 Task automata and TIMES

Timed automata [3] is a modeling language that is widely used for formal modeling and analysis of real-time systems. Essentially, a timed automaton is a finite state automaton to which clocks, that can be tested and reset, are added. Timed automata has shown to be suitable for a wide range of real-time systems.

More recently, the model of timed automata has been extended with a notion of real-time tasks. *Task automata* (of *timed automata with tasks*), associates asynchronous tasks with the states of a timed automaton. It assumes that tasks are executed with static or dynamic priorities by a preemptive or non-preemptive scheduling algorithm. Task automata is supported by the Times tool [5][2], it facilitates schedulability analysis, formal verification by model-checking and code synthesis.

An input system to the Times tool can consist of a task table in which the following parameters are defined for each task: name, computation time, (relative) deadline, priority (in case of static priority scheduling), offset and period (if applicable), interface, semaphore usage, and its C-code. Alternatively, a task can be of type *controlled* which means that its release pattern is defined by a user defined timed automata.

6.3 Problem statement

The aim of this paper is to consider a subsystem (potentially with tasks and a fixed-priority scheduler), residing in a scheduling tree, and to perform schedulability analysis of it. The analysis is done by the Times tool, although it does

[2]For more information about Times, see `http://www.times-tool.com/`.

not support schedulability analysis of hierarchically scheduled systems. The solution to this is to map the rest of the tasks and subsystems in the tree to a small amount of interference tasks. Also, for the sake of prototyping, we generate executable code (that emulates the scheduling of a scheduling tree) of hierarchically scheduled systems. In this section, we first outline the system model used, followed by some limitations and a description of our approach.

6.3.1 System model

A system $\mathcal{S}$ consists of a root S_0 and n subsystems $S_1, ..., S_n$. We assume independent tasks, i.e., there is no synchronization between tasks in the scheduling tree. Each subsystem S_i is defined as a tuple $\langle P_i, Q_i, \mathcal{T}_i, p_i, pr_i \rangle$, where P_i is the subsystem period, Q_i is the amount of CPU (or computation time) provided to the subsystem in each P_i, $\mathcal{T}_i$ is the set of subsystems (S) and tasks (τ) residing in subsystem S_i, $p_i \in [0..n]$ is the index of the parent of S_i, and pr_i is the fixed priority of S_i (higher value means higher priority). Each task τ_j is defined as a tuple $\langle T_j, C_j, D_j, pr_j \rangle$, where T_j is the task period, C_j is the task worst case execution time, D_j is the relative deadline and pr_j is the task priority (higher value means higher priority). The root S_0 is defined by the tuple $\langle \mathcal{T}_0 \rangle$, i.e., just a set of subsystems and tasks.

An example system with root S_0, subsystems S_1 and S_2 (of S_0), and subsubsystems S_3 and S_4 (subsystems of S_2), is illustrated in Figure 6.2.

Limitations: We assume that the whole system and all subsystems are scheduled by fully preemptive fixed-priority schedulers. Generalizing the considered scheduling policy is deferred to future work. Given the system model defined above, we also impose the following two limitations on the relationship between task and subsystem periods:

- $\{\forall\, S_{i,i\in[1,n]} : P_i \geq P_{p_i}\}$, i.e., all subsystem periods are greater or equal to their respective parent's subsystem period and
- $\{\forall\, S_{i,i\in[1,n]}, \forall\, \tau_k \in \mathcal{T}_i : T_k \geq P_{p_i}\}$, i.e., all task periods are greater or equal to its corresponding subsystem's period.

The main reasons for these assumptions are twofold: (1) the inequalities are recommended in order to have a resource efficient system, (2) analysis of the system is simplified given the fulfillment of the above 2 inequalities.

6.3.2 Approach

The objective is to perform schedulability analysis of the contents (tasks/subsystems resident in $\mathcal{T}_i$) of a subsystem S_i, with respect to its interface and the interference from the rest of the tree. This analysis is intended to assist engineers in the development of a subsystem. In doing the analysis, we create a set of interference tasks $\mathcal{I}_i$, representing (and consuming the computation time of) the rest of the system, i.e., the whole system excluding the subsystem under analysis. Hence, the interference from $\mathcal{I}_i$ represents the interference from the whole tree (excluding the subsystem under analysis). Each interference task is described by period a T, an offset O, and a computation time C. Given the interference tasks and the contents of the subsystem under analysis (i.e. the subsystem tasks), the Times tool is used to calculate timing properties (worst case response time) of the task set in S_i. Moreover, the Times tool is used for code synthesis, allowing for early prototyping of hierarchically scheduled subsystems.

In order to perform analysis of a complete system, i.e., for each subsystems in a system, the approach outlined above can be repeated for each subsystem in the system. If the analysis shows that the scheduling of each subsystem is successful, then we can conclude that the whole system is schedulable. Traversing the system tree and analysing each subproblem can be performed automatically, either encoded as an automata in Times, or using an external script program. In this paper however, we leave the details of how to analyze a whole system, and focus on the analysis of one subsystem.

6.4 Analysis of hierarchical systems

In order to analyze the tasks and subsystems, residing inside a subsystem (i.e., the subsystem under analysis), we create a set of interference tasks $\mathcal{I}_i$. Tasks and subsystems residing in the subsystem under analysis are then, together with the interference tasks $\mathcal{I}_i$, used as input to a tool for timing analysis. In this paper, we use the Times tool because it supports analysis of several properties, as well as code synthesis (see Section 6.5).

In the following, we outline how to obtain the set $\mathcal{I}_i$, a procedure with the following three main steps:

Step 1: First we create a partial schedule s_i, i.e., execution sequence (an example can be found in Figure 6.3). This schedule includes all subsystems and

tasks interfering with the subsystem under analysis, including the subsystem itself (S_i). The set of subsystems and tasks influencing the execution of a given subsystem is computed by the function HEP.

We define the recursive function $HEP(S_i)$ for a given system $\mathcal{S}$ in the following way. $HEP(S_i)$ is the set of subsystems (including S_i itself), on the same level of the scheduling tree as S_i (with the same parent as S_i), that have higher priority than subsystem S_i. The recursiveness is defined in that HEP must also be calculated for the parent of S_i (Eq. 6.1). However, the HEP set of the root node is empty (Eq. 6.2).

$$HEP(S_i) = HEP(S_{p_i}) \cup \{\forall\, S_k \in \mathcal{T}_{p_i} : pr_k \geq pr_i\} \cup S_i \tag{6.1}$$

$$HEP(S_0) = \{\} \tag{6.2}$$

For the set of tasks $HEP(S_i)$, we compute the schedule s_i for the time interval $[0, l_i]$, where

$$l_i = \text{LCM}(\{\forall\, k \in HEP(S_i) : P_k\})$$

,i.e., upto the least common multiple of the periods in the set $HEP(S_i)$.

Example: To show how the procedure works, we use a simple example of a hierarchical scheduled system consisting of 4 subsystems with the following parameters:

$$S_1 = \langle 4, 1, \mathcal{T}_1, 0, 3\rangle$$
$$S_2 = \langle 3, 2, \mathcal{T}_2, 0, 4\rangle$$
$$S_3 = \langle 5, 1, \mathcal{T}_3, 2, 2\rangle$$
$$S_4 = \langle 6, 2, \mathcal{T}_4, 2, 1\rangle$$

The example system is outlined in Figure 6.2. Suppose that subsystem S_3 is the subsystem that we are analyzing. Looking at S_3, HEP(S_3) = $\{S_2, S_3\}$ (highlighted in Figure 6.2) and l_3 = LCM(HEP(S_3)) = 15.

Scheduling the example system, for the interval 0 to $l_3 = 15$, gives the schedule s_3, depicted in Figure 6.3.

Step 2: In this step, we take schedule s_i as input and create an ordered set of time points ϕ_i. The first element is 0, the last is l_i = LCM($\{\forall\, k \in HEP(S_i) : P_k\}$), and the intermediate are the time-points when subsystem S_i is scheduled for execution, and is started, preempted or finished, in the time interval [0, l_i].

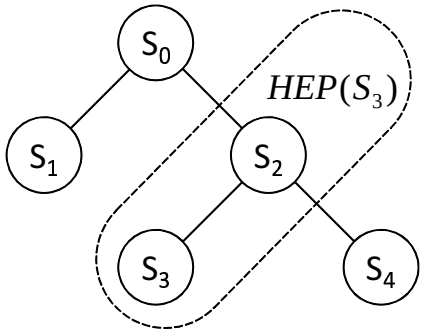

Figure 6.2: Example hierarchical system.

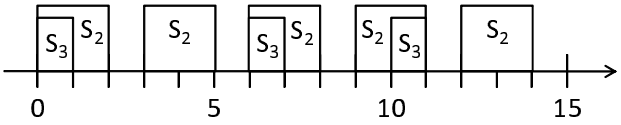

Figure 6.3: Schedule s_3 given S_3 and $l_3 = 15$.

Example (continued): Given the example system above, ϕ_3 is as follows:

$$\phi_3 = \{0, 0, 1, 6, 7, 10, 11, 15\}$$

representing a schedule starting at time 0, where the subsystem under analysis is scheduled initially at time 0, finished at time 1, scheduled again at time 6, finished at time 7, scheduled again at time 10, finished at time 11, and LCM is 15.

Step 3: In this step, given ϕ_i as input, we create a set of interference tasks $\mathcal{I}_i$. Let $|\phi_i|$ denote the number of elements in ϕ_i. We have to create $m = \frac{|\phi_i|}{2}$ interference tasks, $\partial_0, ..., \partial_{m-1}$. The task parameters are $\partial_j = \langle T_j, O_j, C_j, pr_j \rangle$, where T_j is the period of the task (set to T_j = LCM($\{\forall\ k \in HEP(S_i) : P_k\}$) for all interference tasks), O_j is the offset of the interference tasks given by $O_j = \phi_i[j * 2]$, given that $\phi_i[x]$ returns the value stored in ϕ_i at position x (given that positions are indexed starting with 0 and finishing with $|\phi_i| - 1$),

$C_j = \phi_i[1 + j * 2] - \phi_i[j * 2]$, and for pr_j the following holds: $pr_j > pr_k$, where index k is defined by the set $\forall\ (\tau_k \wedge S_k) \in \mathcal{T}_i$.

Example (continued): Looking at the example system again, $m = \frac{|\phi_3|}{2} = 4$, hence $\mathcal{I}_3$ hosting the set of 4 interference tasks is $\mathcal{I}_3 = \{\partial_0, \partial_1, \partial_2, \partial_3\}$ with

$$\begin{aligned}
\partial_0 &= \langle 15, 0\ \ , 0, pr_0 \rangle \\
\partial_1 &= \langle 15, 1\ \ , 5, pr_1 \rangle \\
\partial_2 &= \langle 15, 7\ \ , 3, pr_2 \rangle \\
\partial_3 &= \langle 15, 11, 4, pr_3 \rangle
\end{aligned}$$

Once the above three steps are finished, all interference tasks stored in $\mathcal{I}_i$, together with the tasks and subsystems ($\mathcal{T}_i$) in the subsystem under analysis, are taken as input to Times, giving detailed analysis of all tasks in $\mathcal{T}_i$.

6.5 Modeling example

In order to illustrate our solution, we have modeled an example system consisting of 4 subsystems, arranged in a hierarchical tree, depicted in Figure 6.4. The engineering challenge, highlighted in this example, is how a development team (given a scheduling tree and a dedicated subsystem within it) can develop an application, consisting of real-time tasks, and be able to perform schedulability analysis of these tasks, in order to verify whether or not they meet their respective deadlines. Such a verification should be possible when specifying and allocating task parameters, preferably early during the development and testing phase, allowing for early prototyping. The latter requires a way to execute the tasks, on a given platform, within their corresponding time slots, determined by the actual scheduling of the whole system (of subsystems). This will be shown in section 6.5.2 and 6.5.3.

Recall, in this paper it is assumed that tasks within one subsystem do not need to synchronize/communicate with tasks residing in other subsystems. Given this assumption, we do not need to consider detailed scheduling of tasks in other subsystems, since their exact scheduling does not affect the scheduling of the subsystem under analysis.

To summarize the above, in this example, we want to:

1. conduct schedulability analysis of a subsystems content (subsystem **A** and **C**'s content in this example), with respect to the interface(s) of subsystem **A**, respectively **C**, and the rest of the subsystems, and

2. generate executable code, a scheduler to be precise, that execute subsystem **A** and **C**'s content, within its precise time slots, as if the whole system of subsystems was executing (even though we only have source code and task parameters of subsystem **A** and **C**).

An assumption is that the subsystems in the tree are schedulable (for which they are in this example) and that the scheduling tree is pre-determined by the system description or similar. As a development team, you are given the timing parameters of your subsystem (i.e., subsystem **A** or **C** in this case), which is the period and capacity of these subsystems. The responsibility of the development team is to develop an application consisting of a set of tasks that are schedulable given the timing parameters of their subsystem. The issue for the development team to solve, is to assure that their application is schedulable considering that their application will (in the future and final system) be scheduled together with other subsystems in the hierarchical scheduling tree. Hence, the development team cannot assume that their subsystem, **C** for example, will get 1 time slot exactly every 10 time units because subsystems, at the same or higher level in the scheduling tree, might interfere (as they may have higher priority than subsystem **C**). The timing analysis of a subsystem (and its tasks) must consider all subsystem (of the same or higher level and with higher priority) parameters, including its own.

The first step is to analyze whether the chosen task parameters are sufficient in order for the tasks to meet their deadlines. What should be done is to add these tasks to the scheduling tree, like the one in Figure 6.4, under their subsystem, and check if they are schedulable with relation to the interfaces of the subsystems in the tree. This can be done with a schedulability test such as Response Time Analysis (RTA) [9] for hierarchical systems [10]. However, we want to show how this can be done in Times, by generating interference tasks (called dummy tasks in this section). These tasks emulate correct execution of the subsystem under analysis by blocking out time representing higher priority subsystem execution time, as well as time when the system should be idle. By laying out the schedule of all subsystems, one can identify the time-slots when the subsystem under analysis should be executed, and thereby also the inverse of this time. This inverse time represents the time that should be ”blocked out” in order to simulate interference from higher priority subsystems, as well as idle time. We achieve this ”blocking out” (interference) by creating dummy tasks with higher priority than that of the tasks in the subsystem under analysis (as described in Section 6.4). Once the dummy tasks are generated (which can be done following the steps in Section 6.4), they can be inserted into the Times

tool. The dummy tasks' release pattern can either be described (in Times) in a task-parameter table (e.g. by setting offset, priority, period etc.) or by constructing an automata. The latter has an advantage when generating code (this will be covered in more detail in Section 6.5.2). However, for schedulability analysis of tasks in Times, the easier approach is to specify the dummy tasks in the task-parameter table. After entering the dummy task parameters together with the subsystem tasks in Times, it can simulate the system and do response-time analysis as shown in Figure 6.6 and 6.14. Times will output whether or not the system is schedulable, and if schedulable, it will also give the Worst Case Response Time (WCRT) of all tasks.

In conclusion, the schedulability analysis performed in Times, is a simulation which will produce the WCRT of each task. So we have actually simplified the problem into a response time analysis of a set of periodic tasks (belonging to the subsystem under analysis), together with a set of periodic tasks with offsets (the dummy tasks). The WCRT value will include the interference from subsystems (that can reside at different levels of the scheduling tree), which is actually modeled as interference from higher priority tasks, as well as the execution time of the task itself. Hence, for the sake of timing analysis, timing analysis tools other than Times can be used. However, we are not only interested in timing analysis, but also in generating code for early prototyping of the subsystem under analysis.

6.5.1 Code synthesis

The Times tool is equipped with an automatic code generator which can generate C-code of the modeled system to the platform brickOS[3], as well as a simulator for Linux. We have used this code generator to generate code of our example system. We show two examples, where we synthesize code for a scheduler for VxWorks (section 6.5.2) and Linux (section 6.5.3). The generated code is then transformed (extended) to fit the new software platform, i.e., VxWorks or Linux. This transformation was done manually but could also be done automatically.

The reason for choosing VxWorks is that we are well familiar with task scheduling, execution tracing etc. in this platform, it provides an industry standard task scheduler, and it is a preferred platform of several of our industrial partners. Having knowledge of scheduling is specially important since we need to map brickOS scheduling to VxWorks (since the code generator generates brickOS code).

[3] `http://brickos.sourceforge.net/`

For Linux, we generate the Linux simulator code from Times, then we remove the simulator code manually (could be done automatically). What is left is the actual automata code (i.e., the scheduler). The automata code in turn is extended to fit in the Linux kernel, such that it can schedule tasks. This is a manual step (which can be automated).

6.5.2 Subsystem C

In this example, the global scheduler and all local schedulers (i.e. the internal scheduler of each subsystem) schedule their tasks/subsystems according to fixed-priority preemptive scheduling. The priority assignment is done according to Rate Monotonic [8], i.e, the shorter the period, the higher the priority. Subsystem **C** resides in the tree represented in Figure 6.4.

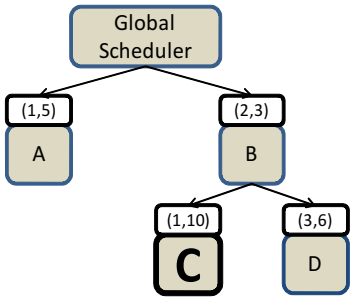

Figure 6.4: Subsystem **C**.

In doing schedulability and response-time calculations, we need a detailed description of the task set resident in subsystem **C**; these details are represented in Table 6.1.

Name	T	C	D	pr
task1 (τ_1)	40	1	40	5
task2 (τ_2)	50	1	50	4
task3 (τ_3)	80	1	80	3
task4 (τ_4)	90	1	90	2
task5 (τ_5)	250	7	250	1

Table 6.1: Task set of subsystem **C**.

Schedulability analysis

The corresponding schedule for s_C, executing in the example system, is illustrated in Figure 6.5.

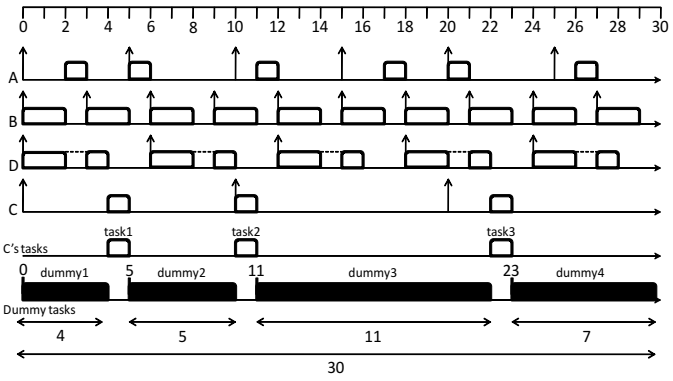

Figure 6.5: Schedule for subsystem **C**.

From this schedule we can conclude which dummy tasks that we need (∂_1-∂_4), as shown in Table 6.2.

Name	T	O	C	pr
dummy1 (∂_1)	30	0	4	6
dummy2 (∂_2)	30	5	5	6
dummy3 (∂_3)	30	11	11	6
dummy4 (∂_4)	30	23	7	6

Table 6.2: Generated dummy tasks for subsystem **C**.

The last step is to input all tasks in the Times tool and let it perform a simulation. Figure 6.6 shows that subsystem **C**'s tasks are schedulable with the 4 dummy tasks, i.e., the other three subsystems in the system.

Code synthesis to VxWorks (kernel version 6.6)

In the analysis part (Section 6.5.2), we analyzed the system based on dummy tasks (with offsets). We created periodic tasks and assigned the offsets through the task parameter table (all other tasks were also created in this manner). Creating tasks with offsets can also be done by creating an automata. This has the

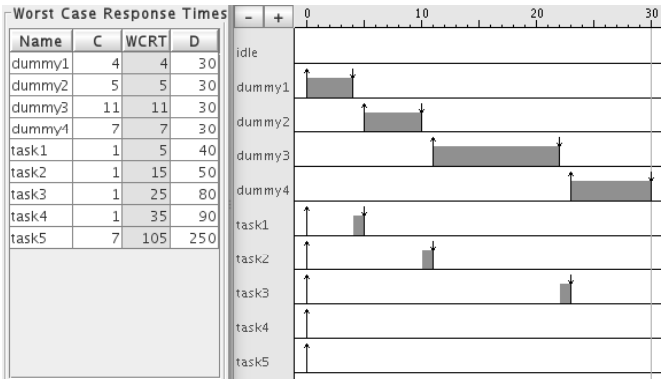

Figure 6.6: Times schedulability analysis (for subsystem **C**).

advantage that we can specify that only one dummy task is released at all offset instances and thereby replacing all dummy tasks with only one. This is good when generating code, since most RTOSs have an upper limit on the amount of tasks. At code level, the execution time of this dummy task must be set to be dynamic, since it is replacing tasks which most probably have different execution times. The two automata in Figure 6.7 models the releasing of dummy tasks (a similar automata, but with other release times, is used for the example in section 6.5.3).

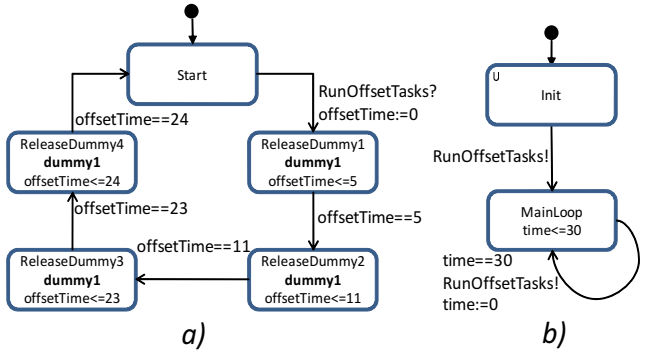

Figure 6.7: Task automata.

The automata in Figure 6.7**b)**, releases the second automata (Figure 6.7**a)**)

every 30 time units by calling a synchronization function **RunOffsetTasks!** which starts a transition in the edge where **RunOffsetTasks?** is located. The second automata releases the dummy tasks according to the calculated offsets (with relation to the period). **time** and **offsetTime** are two clocks that progresses in discrete time. An invariant such as **offsetTime**<**=5** (located inside a state) means that the automata may only be in that state until this condition does not hold. A condition at an edge such as **offsetTime==5** means that the transition can be made only when this condition holds. A statement such as **time:=0** means that the variable (in this case a clock) is assigned a value. Whenever there is a transition to a state with a task name, such as **dummy1**, this task is released for execution.

```
1: task( ) {
2.     while(TRUE) {
3.             wait_event(task_release, release_flag)
4.             // Task code here
5.     }
6. }
7: controller( ) {
8:       wait_event(check_trans, 0)
9: }
```

Figure 6.8: Function task() and controller().

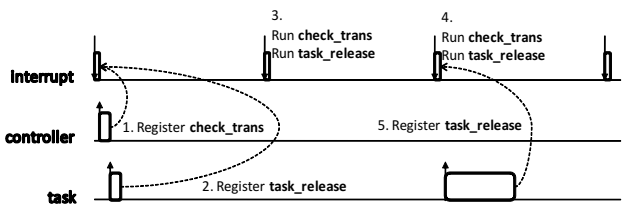

Figure 6.9: brickOS scheduling.

The mapping from the C-code (generated by Times) to VxWorks consists mostly of changing the way the task is suspended and released. In the brickOS generated code, an initializer task called **controller** (Figure 6.8, lines 7-9) calls **wait_event** in order to register a function **check_trans** that will be executed at every system tick by an interrupt routine. This will stop when the function

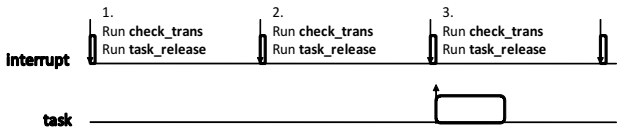

Figure 6.10: VxWorks scheduling.

returns a non-zero value (which is not the case for **check_trans**). This function traverses the automata (both user defined automata and Times default generated automata) and sets a flag whenever there should be a task release. Each task (Figure 6.8, lines 1-6) registers a function **task_release** at the beginning of its execution, before it suspends. This function checks whether the flag is set, if so, it will return a non-zero value that in turn will release the corresponding task. Figure 6.9 illustrates how the scheduling is done in the generated code for brickOS. The mapping of this scheduling to VxWorks is illustrated in Figure 6.10. We create an interrupt routine that is executed at every system tick. This routine executes both the **check_trans** function and each tasks **task_release** function. Whenever **check_trans** sets the task flag, i.e. that is when **task_release** returns a non-zero value, the corresponding task is inserted into the VxWorks ready queue.

We have successfully generated C-code for the example system in Figure 6.4, that is comprised of the tasks in Table 6.1 and Table 6.2. We transformed the generated code and ran the system in VxWorks 6.6 on a Intel Pentium4 platform. Further, we recorded and visualized the execution trace with the Tracealyzer tool[4].

Figure 6.11 shows the graphical representation of the running tasks (note that tasks 'dummy1' etc. from Figure 6.6 are named 'idle1' etc. in Figure 6.11) at critical instant and the recorded data is shown in Table 6.3. Figure 6.6 shows the WCRT of the simulation, corresponding to **Max. Response time** in Table 6.3, note that the time-base is 1000 times bigger in Table 6.3. The maximum response times in Table 6.3 are significantly higher than the simulation values because of overhead (scheduling, context switches etc.). This prolonged response time is illustrated in Figure 6.11. **task2** does not finish its entire execution before **idle3** starts, leading to that **task2** has to wait for it to finish (which will take 11 time units), and then execute the final part (it is a very small amount so it does not show in this resolution). This kind of execution

[4] `http://www.tracealyzer.se/`.

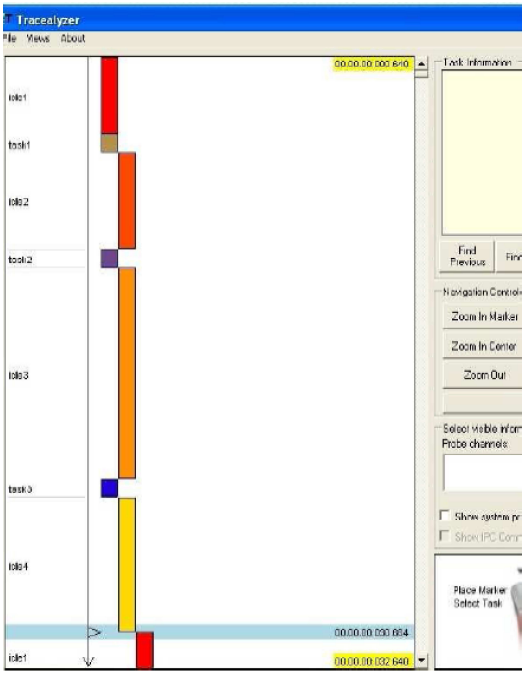

Figure 6.11: Tracealyzer screenshot.

scenario is valuable for a development team and can only be discovered in time, in the development process, through early prototyping/testing.
Table 8.3 shows the scheduling overhead (from running the tasks in Table 6.1 and 6.2) from the generated scheduler (Times) and a manually coded scheduler; the Hierarchical Scheduling Framework (HSF) [11]. We measured the schedulers execution times with micro-second resolution, 10 times each (Table 8.3 shows the average values), between time zero (when the system started) and LCM of all tasks (18000000 μs). The HSF scheduler only executes at task release and task deadline (in the latter case it checks if the task has finished), while the Times scheduler executes at every system tick (i.e. every milli-second), and releases tasks if necessary. VxWorks itself handles task switching due to that a task has finished. The conclusion is that even though Times runs more frequently than HSF, HSF still produces more overhead (the majority of it comes from queue-management [11]).

Task	Execution time (μs)		Response time (μs)	
	Avg.	Max.	Avg.	Max.
task1	996	999	11999	14003
task2	996	999	16998	24000
task3	995	997	27994	33996
task4	995	997	32042	63982
task5	6267	6973	228643	291888
idle1	3995	4004	3995	4004
idle2	5000	5001	5000	5001
idle3	11000	11001	11000	11001
idle4	6999	7007	10994	11004

Table 6.3: Tracealyzer result.

Scheduler	Avg. overhead/Duration (μs)	Avg. overhead (%)
Times	1952/18000000	0.01084
HSF	3283/18000000	0.01824

Table 6.4: Scheduling overhead.

6.5.3 Subsystem A

This example also assumes fixed-priority preemptive scheduling of periodic tasks/subsystems, as well as rate monotonic priority assignment.

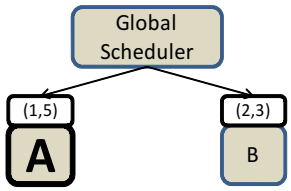

Figure 6.12: Subsystem **A**.

The content of subsystem **A** is one task (Table 6.5), which correspond to the parameters of its subsystem. Subsystem **A**'s position in the scheduling tree is shown in Figure 6.12.

Schedulability analysis

By laying out the schedule for subsystem **A** (Figure 6.13), we have generated the necessary dummy tasks (Table 6.6).

Name	T	C	D	pr
taskA	5	1	5	1

Table 6.5: Task set of subsystem **A**.

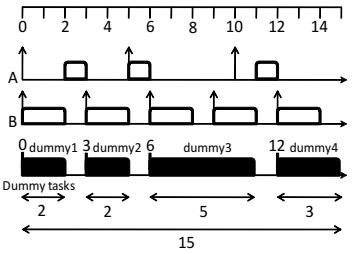

Figure 6.13: Schedule for subsystem **A**.

By inserting all tasks (Table 6.5 and 6.6) into Times and running its simulation, we can get the schedulability analysis for subsystem **A**'s task. This is shown in Figure 6.14, the tool will output the worst case response times of all tasks if the system is schedulable.

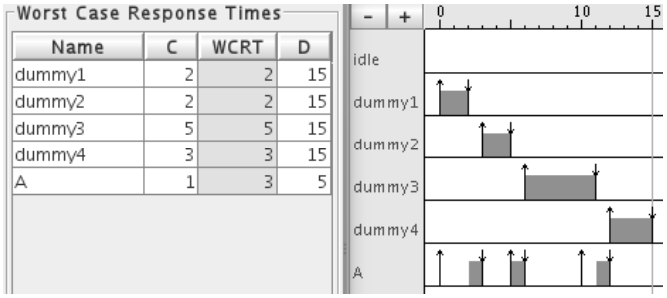

Figure 6.14: Times schedulability analysis (for subsystem **A**).

Code synthesis to Linux (kernel version 2.6.31-9)

The subsystem (**A**) execution trace is illustrated in Figure 6.13, as illustrated, the four dummy tasks replace subsystem **B**. We let a video processing appli-

Name	T	O	C	pr
dummy1 (∂_1)	15	0	2	2
dummy2 (∂_2)	15	3	2	2
dummy3 (∂_3)	15	6	5	2
dummy4 (∂_4)	15	12	3	2

Table 6.6: Generated dummy tasks for subsystem **A**.

cation (VLC[5]) replace task $taskA$ in subsystem **A** in our experiments. The release of subsystem **A** and dummy tasks 1-4 is done with two automata similar to the ones in Figure 6.7. We generate code, using the Times code generator for generating a Linux simulator. The simulator will run the automata, which is also generated by Times. We then replace the simulator with Linux kernel scheduling functions, which are exported by the scheduling framework Resch [12]. Resch is unique in that it does not require the user to make any changes in the Linux kernel, when implementing a scheduler in Resch. It runs as a kernel module, and the user implemented scheduler will act as a plugin kernel module to Resch (hence no kernel patches are required). The automata code generated from Times, is wrapped with Resch scheduling primitives, and it is executed as a kernel module in Linux. In the experiments, all tasks, i.e., the VLC application and the dummy tasks, are running as Linux real-time tasks.

We also ran the VLC application in a 2-level hierarchical scheduling framework, which is able to run a global scheduler, scheduling an arbitrary number of subsystems in one level. The subsystems themselves may have their own local scheduler. All schedulers (local and global) schedule with fixed-priority preemptive scheduling of periodic tasks/subsystems. The framework is implemented by the authors of the paper, and it runs as a plugin scheduler in Resch, i.e., as a kernel module. We executed subsystem **A** and **B** (Figure 6.12) with corresponding parameters, including rate monotonic priorities, in the hierarchical scheduling framework. Subsystem **B** corresponds to B in Figure 9.16 and subsystem **A** maps to A ($Idle$ is the idle subsystem). The VLC application (referred to as vlc_A in Figure 9.16) was running in subsystem A, the dummy task $task\_B$ was running in B and dummy task $idle$ was running in subsystem $Idle$ (which has lowest priority among the subsystems). Task $linux$ is the Linux idle task which will run whenever $task\_B$, vlc_A or $idle$ does not run.

Figure 6.16 shows the execution trace when running the Times automata in Resch, as a plugin scheduler. The dummy tasks ($dummy1$, $dummy2$, $dummy3$ and $dummy4$) in Figure 6.16 corresponds to our generated dummy

[5]VLC `http://www.videolan.org/vlc`

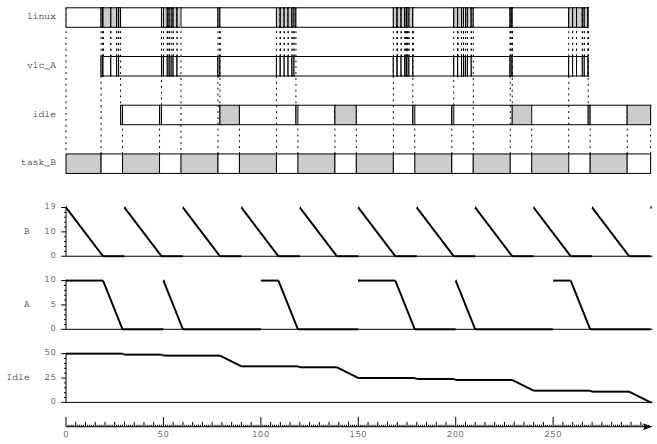

Figure 6.15: Execution recording from the HSF scheduler.

tasks in Figure 6.13 (these tasks have highest priority). The VLC application was running as task vlc (intermediate priority) and the Linux idle task $linux$ was running with lowest task priority.

We ran all the experiments on an Intel Pentium Dual-Core (E5300 2,6GHz) platform, equipped with a Linux kernel version 2.6.31.9, running with load balancing disabled (no automatic task migration) for simplicity. The task execution recording was done with the tool Ftrace [13], and the recording of subsystem scheduling events were done by our own recorder [14] (which is integrated in HSF). The recordings were visualized (Figure 9.16 and 6.16) with the tool Grasp [15].

Scheduler	fps (average)
Times scheduler	25.3174938
HSF scheduler	25.3582266
Linux scheduler	30

Table 6.7: Frames per second (fps) measurements of VLC.

We measured the execution time of the VLC application, while it processed a 91 frame long video, with corresponding audio. The measurements were done 10 times for each scheduler, and the data presented (Table 6.7) represents the average values. The resulting data is presented as the number of frames dis-

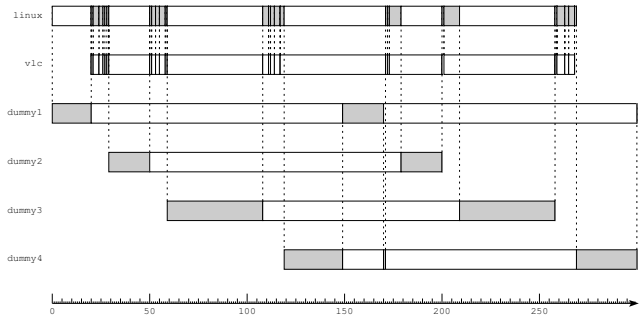

Figure 6.16: Execution recording from the Times scheduler.

played per second (Table 6.7). The measurements were done while scheduling the VLC application with the HSF scheduler, and the Times scheduler. When running VLC with only the native Linux scheduler, the video processing reached approximately 30 fps. The presented fps values shows that both schedulers (HSF and Times) gives almost the same amount of CPU power (approximately 20%) to the VLC application. However, VLC does not use all of its allocated CPU time (Figure 9.16 and 6.16) because its internal clock will decide when to process and display frames, which is dependant on the intended frame-rate of the application (which is 30 fps).

The time points when the Times scheduler allocates CPU time to VLC (Figure 6.16), matches the points that are generated by the scheduling framework HSF (Figure 9.16), which implements the scheduler that is intended to be used in the final system. However, HSF "leaks" CPU time, as can be seen in Figure 9.16. This is due to that we set the budget of subsystem **B** to less than 20, so that the budget does not deplete at the same time as the other subsystem is released (which may cause our scheduler to execute the scheduling events in wrong order).

6.6 Related work

Related work in the area of hierarchical scheduling originated in open systems [16] in the late 1990's, and it has been receiving an increasing research attention ever since. Since Deng and Liu [16] introduced a two-level hierarchical scheduling framework, its schedulability has been analyzed under fixed-

priority global scheduling [17] and under EDF-based global scheduling [18, 19]. Mok *et al.* [20] proposed the bounded-delay resource model so as to achieve a clean separation in a multi-level hierarchical scheduling framework, and schedulability analysis techniques [21, 22] have been introduced for this resource model. In addition, Shin and Lee [7] introduced the periodic resource model (to characterize the periodic resource allocation behavior), and many studies have been proposed on schedulability analysis with this resource model under fixed-priority scheduling [10, 23, 24] and under EDF scheduling [7].

Looking at the kind of analysis possible with these hierarchical scheduling approaches, typically only timing is considered. In this paper, we are also interested in code synthesis, as well as analysis using task automata. This is similar to [25], where the authors show how modeling and schedulability analysis of two-level hierarchical scheduling, with timed automata, can be accomplished in the simulation tool Cheddar. Lime *et al.* [26] model fixed and dynamic priority scheduling using time petri nets, which is similar to the work in [27]. Scheduler modeling is showed in [28] using the controller paradigm.

6.7 Conclusion

We have shown how to perform schedulability analysis in the Times tool, where a subsystem within fixed-priority preemptive hierarchical scheduling is the system under analysis. The concept we present simplifies the analysis of the whole system by analysing one subsystem and abstracting the rest of the system (black-boxing). Iterating through all subsystems in this manner results in analysing the whole system. In each step, the black-boxing is done by replacing interfering subsystems with a small set of high priority tasks (which we refer to as dummy tasks). The procedure is described with an algorithm in the paper, and the output of the algorithm is a set of dummy tasks that are periodic with offsets. These tasks, and the tasks of the subsystem to be analyzed, are then modeled in the Times tool (with a task-table or timed automata). The last step is to run a simulation in Times which will generate the worst case response time of each task, thereby deciding if the subsystem is schedulable or not. The Times tool could traverse the scheduling tree and analyze each subsystem, resulting in a complete analysis of the whole tree. The simulation itself is essentially a response time analysis of tasks that are periodic, whereas some of them will also have offsets (the dummy tasks).

We have used the Times code synthesis and shown how to generate C-code of two example systems. The code has been extended to execute on an in-

dustrial platform (i.e. VxWorks), and also on a PC desktop platform (Linux). Hence, our proposed method has shown to be practical. After the code generation, a subsystem can be executed as if it would be running within a hierarchically scheduled system. Hence, our proposed approach supports early prototyping of hierarchically scheduled systems, by using our dummy-task algorithm together with our code synthesis for VxWorks and Linux.

Our example in VxWorks shows that response times can vary significantly when moved from simulation to a real platform, even though a very small amount of overhead is introduced. The overhead measurements show that the scheduler, generated from Times, produces less overhead compared to a manually coded scheduler. Our other example in Linux shows how a video processing application (VLC) is affected when running it in a prototyped subsystem. We have measured the frame-rate and compared the results from the same example system running in a 2-level hierarchical scheduling framework.

As future work, we plan to optimize the code synthesis (in order to minimize scheduler overhead) as well as to model and generate code for hierarchical scheduling frameworks. This is interesting in the context of proving the correctness of scheduling, since model checking could be used to verify the schedulers. As a last step of the contribution of this paper we plan to implement the concept in a tool, which will provide graphical modeling of systems, automatic generation of dummy tasks as well as automatic synthesis for various platforms (such as VxWorks, Linux and FreeRTOS).

Bibliography

[1] Mikael Åsberg, Moris Behnam, Farhang Nemati, and Thomas Nolte. Towards Hierarchical Scheduling in AUTOSAR. In *14th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1181–1188, Sep 2009.

[2] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task Automata: Schedulability, Decidability and Undecidability. *Journal of Information and Computation*, 205(8):1149–1172, Aug 2007.

[3] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Journal of Theoretical Computer Science*, 126(2):183–235, April 1994.

[4] Tobias Amnell, Elena Fersman, Paul Pettersson, Wang Yi, and Hongyan Sun. Code Synthesis for Timed Automata. *Nordic Journal of Computing*, 9(4):269–300, Dec 2002.

[5] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A Tool for Modelling and Implementation of Embedded Systems. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464, April 2002.

[6] Mikael Åsberg, Thomas Nolte, and Paul Pettersson. Prototyping Hierarchically Scheduled Systems using Task Automata and TIMES. In *5th IEEE International Conference on Embedded and Multimedia Computing*, pages 1–8, Aug 2010.

[7] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *24th IEEE International Real-Time Systems Symposium*, pages 2–13, Dec 2003.

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

[9] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Journal of Software Engineering*, 8:284–292, 1993.

[10] Rob Davis and Allan Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *26th IEEE International Real-Time Systems Symposium*, pages 389–398, Dec 2005.

[11] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards Hierarchical Scheduling in VxWorks. In *4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 67–76, July 2008.

[12] Mikael Åsberg, Shinpei Kato, Thomas Nolte, and Ragunathan Rajkumar. ExSched: An External CPU Scheduler Framework for Real-Time Systems. In *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2012.

[13] Tim Bird. Measuring Function Duration with Ftrace. In *1st Annual Japan Linux Symposium*, Oct 2009.

[14] Mikael Åsberg, Thomas Nolte, and Shinpei Kato. A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 380–387, Aug 2011.

[15] Mike Holenderski, Martijn Heuvel, Reinder Bril, and Johan Lukkien. Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 37–42, July 2010.

[16] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *18th IEEE International Real-Time Systems Symposium*, pages 308–319, Dec 1997.

[17] T.-W. Kuo and C.H. Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *20th IEEE International Real-Time Systems Symposium*, pages 256–267, Dec 1999.

[18] G. Lipari and S. Baruah. Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems. In *6th IEEE Real Time Technology and Applications Symposium*, pages 166–175, May 2000.

[19] G. Lipari, J. Carpenter, and S. Baruah. A Framework for Achieving Inter-Application Isolation in Multiprogrammed Hard-Real-Time Environments. In *21th IEEE International Real-Time Systems Symposium*, pages 217–226, Nov 2000.

[20] A. Mok, X. Feng, and D. Chen. Resource Partition for Real-Time Systems. In *7th Real-Time Technology and Applications Symposium*, pages 75–84, May 2001.

[21] X. Feng and A. Mok. A Model of Hierarchical Real-Time Virtual Resources. In *23rd IEEE International Real-Time Systems Symposium*, pages 26–35, Dec 2002.

[22] Insik Shin and Insup Lee. Compositional Real-Time Scheduling Framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, Dec 2004.

[23] G. Lipari and E. Bini. Resource Partitioning Among Real-Time Applications. In *15th IEEE Euromicro Conference on Real-Time Systems*, pages 151–158, July 2003.

[24] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of Hierarchical Fixed-Priority Scheduling. In *14th IEEE Euromicro Conference on Real-Time Systems*, pages 152–160, July 2002.

[25] Frank Singhoff and Alain Plantec. AADL Modeling and Analysis of Hierarchical Schedulers. In *ACM International Conference on SIGAda*, pages 41–50, Nov 2007.

[26] Didier Lime and Olivier H. Roux. Formal Verification of Real-Time Systems with Preemptive Scheduling. *Journal of Real-Time Systems*, 41(2):118–151, Feb 2009.

[27] K. Altisen, G. Gosler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A Framework for Scheduler Synthesis. In *20th IEEE International Real-Time Systems Symposium*, pages 154–163, Dec 1999.

[28] Joseph Sifakis. Scheduler Modeling Based on the Controller Synthesis Paradigm. *Journal of Real-Time Systems*, 23(1/2):55–84, July 2002.

Chapter 7

Paper B: Towards Hierarchical Scheduling in VxWorks

Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg and Reinder J. Bril
In OSPERT'08 Workshop, pages 67–76, July, 2008

Abstract

Over the years, we have worked on hierarchical scheduling frameworks from a theoretical point of view. In this paper we present our initial results of the implementation of our hierarchical scheduling framework in a commercial operating system VxWorks. The purpose of the implementation is twofold: (1) we would like to demonstrate feasibility of its implementation in a commercial operating system, without having to modify the kernel source code, and (2) we would like to present detailed figures of various key properties with respect to the overhead of the implementation. During the implementation of the hierarchical scheduler, we have also developed a number of simple task schedulers. We present details of the implementation of Rate-Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Finally, we present the design of our hierarchical scheduling framework, and we discuss our current status in the project.

7.1 Introduction

Correctness of today's embedded software systems generally relies not only on functional correctness, but also on extra-functional correctness, such as satisfying timing constraints. System development (including software development) can be substantially facilitated if (1) the system can be decomposed into a number of parts such that parts are developed and validated in isolation and (2) the temporal correctness of the system can be established by composing the correctness of its individual parts. For large-scale embedded real-time systems, in particular, advanced methodologies and techniques are required for temporal and spatial isolation all through design, development, and analysis, simplifying the development and evolution of complex industrial embedded software systems.

Hierarchical scheduling has shown to be a useful mechanism in supporting modularity of real-time software by providing temporal partitioning among applications. In hierarchical scheduling, a system can be hierarchically divided into a number of subsystems that are scheduled by a global (system-level) scheduler. Each subsystem contains a set of tasks that are scheduled by a local (subsystem-level) scheduler. The Hierarchical Scheduling Framework (HSF) allows for a subsystem to be developed and analyzed in isolation, with its own local scheduler, and then at a later stage, using an arbitrary global scheduler, it allows for the integration of multiple subsystems without violating the temporal properties of the individual subsystems analyzed in isolation. The integration involves a system-level schedulability test, verifying that all timing requirements are met. Hence, hierarchical scheduling frameworks naturally support *concurrent development* of subsystems. Our overall goal is to make hierarchical scheduling a cost-efficient approach applicable for a wide domain of applications, including automotive, automation, aerospace and consumer electronics.

Over the years, there has been a growing attention to HSFs for real-time systems. Since a two-level HSF [1] has been introduced for open environments, many studies have been proposed for its schedulability analysis of HSFs [2, 3]. Various processor models, such as bounded-delay [4] and periodic [5], have been proposed for multi-level HSFs, and schedulability analysis techniques have been developed for the proposed processor models [6, 7, 8, 9, 10, 5, 11]. Recent studies have been introduced for supporting logical resource sharing in HSFs [12, 13, 14].

Up until now, those studies have worked on various aspects of HSFs from a theoretical point of view. This paper presents our work towards a full im-

plementation of a hierarchical scheduling framework. We have chosen to implement it in a commercial operating system already used by several of our industrial partners. We selected the VxWorks operating system, since there is plenty of industrial embedded software available, which can run in the hierarchical scheduling framework.

The outline of this paper is as follows: Section 9.6 presents related work on implementations of schedulers. Section 7.3 present our system model. Section 7.4 gives an overview of VxWorks, including how it supports the implementation of arbitrary schedulers. Section 7.5 presents our scheduler for VxWorks, including the implementation of Rate Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Section 7.6 presents the design, implementation and evaluation of the hierarchical scheduler, and finally Section 8.6 summarizes the paper.

7.2 Related work

Looking at related work, recently a few works have implemented different schedulers in commercial real-time operating systems, where it is not feasible to implement the scheduler directly inside the kernel (as the kernel source code is not available). Also, some work related to efficient implementations of schedulers are outlined.

Buttazzo and Gai [15] present an implementation of the EDF scheduler for the ERIKA Enterprise kernel [16]. The paper discusses the effect of time representation on the efficiency of the scheduler and the required storage. They use the Implicit Circular Timer's Overflow Handler (ICTOH) algorithm which allows for an efficient representation of absolute deadlines in a circular time model.

Diederichs and Margull [17] present an EDF scheduler plug-in for OSEK/VDX based real-time operating systems, widely used by automotive industry. The EDF scheduling algorithm is implemented by assigning priorities to tasks according to their relative deadlines. Then, during the execution, a task is released only if its absolute deadline is less than the one of the currently running task. Otherwise, the task will be delayed until the time when the running task finishes its execution.

Kim *et al.* [18] propose the SPIRIT uKernel that is based on a two-level hierarchical scheduling framework simplifying integration of real-time applications. The SPIRIT uKernel provides a separation between real-time applications by using partitions. Each partition executes an application, and uses

the Fixed Priority Scheduling (FPS) policy as a local scheduler to schedule the application's tasks. An offline scheduler (timetable) is used to schedule the partitions (the applications) on a global level. Each partition provides kernel services for its application and the execution is in user mode to provide stronger protection.

Parkinson [19] uses the same principle and describes the VxWorks 653 operating system which was designed to support ARINC653. The architecture of VxWorks 653 is based on partitions, where a Module OS provides global resource and scheduling for partitions and a Partition OS implemented using VxWorks microkernel provides scheduling for application tasks.

The work presented in this paper differs from the last two works in the sense that it implements a hierarchical scheduling framework in a commercial operating system without changing the OS kernel. Furthermore, the work differs from the above approaches in the sense that it implements a hierarchical scheduling framework intended for open environments [1], where real-time applications may be developed independently and unaware of each other and still there should be no problems in the integration of these applications into one environment. A key here is the use of well defined *interfaces* representing the collective resource requirements by an application, rich enough to allow for integration with an arbitrary set of other applications without having to redo any kind of application internal analysis.

7.3 System model

In this paper, we only consider a simple periodic task model $\tau_i(T_i, C_i, D_i)$ where T_i is the task period, C_i is a worst-case execution time requirement, and D_i is a relative deadline ($0 < C_i \leq D_i \leq T_i$). The set of all tasks is denoted by Γ ($\Gamma = \{\tau_i |$ `for all` $i = 1, .., n\}$ where n is the number of tasks).

We assume that all tasks are independent of each other, i.e., there is no sharing of logical resources between tasks and tasks do no suspend themselves.

The HSF schedules subsystems $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the set representing the whole system of subsystems. Each subsystem S_s consists of a set of tasks and a local scheduler (RM or EDF), and the global (system) scheduler (RM or EDF). The collective real-time requirements of S_s is referred to as a *timing-interface*. The subsystem interface is defined as (P_s, Q_s), where P_s is a subsystem period, and Q_s is a budget that represents an execution time requirement that will be provided to the subsystem S_s every period P_s.

7.4 VxWorks

VxWorks is a commercial real-time operating system developed by Wind River with a focus on performance, scalability and footprint. Many interesting features are provided with VxWorks, which make it widely used in industry, such as; Wind micro-kernel, efficient task management and multitasking, deterministic context switching, efficient interrupt and exception handling, POSIX pipes, counting semaphores, message queues, signals, and scheduling, preemptive and round-robin scheduling etc.

The VxWorks micro-kernel supports the priority preemptive scheduling policy with up to 256 different priority levels and a large number of tasks, and it also supports the round robin scheduling policy.

VxWorks offers two different modes for application-tasks to execute; either kernel mode or user mode. In kernel mode, application-tasks can access the hardware resources directly. In user mode, on the other hand, tasks can not directly access hardware resources, which provides greater protection (e.g., in user mode, tasks can not crash the kernel). Kernel mode is provided in all versions of VxWorks while user mode was provided as a part of the Real Time Process (RTP) model, and it has been introduced with VxWorks version 6.0 and beyond.

In this paper, we are considering kernel mode tasks since such a design would be compatible with all versions of VxWorks and our application domains include systems with a large legacy in terms of existing source codes. We are also considering fixed priority preemptive scheduling policy for the kernel scheduler (not the round robin scheduler). A task's priority should be set when the task is created, and the task's priority can be changed during the execution. Then, during runtime, the highest priority ready task will always execute. If a task with priority higher than that of the running task becomes ready to execute, then the scheduler stops the execution of the running task and instead executes the one with higher priority. When the running task finishes its execution, the task with the highest priority among the ready tasks will execute.

When a task is created, an associated Task Control Block (TCB) is created to save the task's context (e.g., CPU environment and system resources, during the context switch). Then, during the life-cycle of a task the task can be in one or a combination of the following states (see Figure 7.1):

- **Ready state**, the task is waiting for CPU resources.
- **Suspended state**, the task is unavailable for execution but not delayed

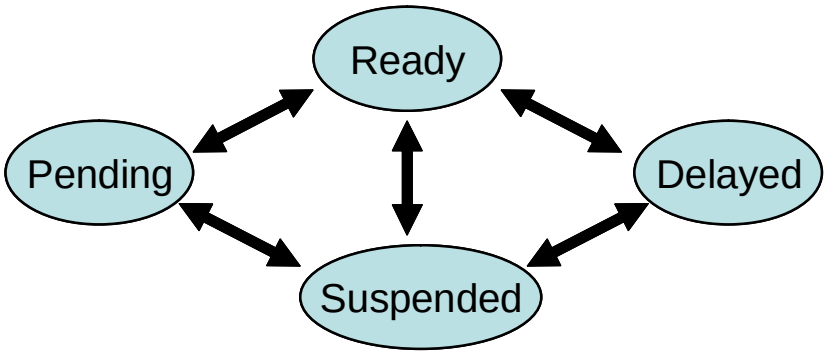

Figure 7.1: The application task state.

or pending.

- **Pending state**, the task is blocked waiting for some resource other than the CPU.
- **Delayed state**, the task is sleeping for some time.

Note that the kernel scheduler sorts all tasks that are ready to execute in a queue called the *ready queue*.

7.4.1 Scheduling of time-triggered periodic tasks

A periodic task is a task that becomes ready for execution periodically once every n-th time unit, i.e., a new instant of the task is executed every constant period of time. Most commercial operating systems, including VxWorks, do not directly support the periodic task model [20]. To implement a periodic task, when a task finishes its execution, it sleeps until the beginning of its next period. Such periodic behaviour can be implemented in the task by the usage of timers. Note that a task typically does not finish its execution at the same time always, as execution times and response times vary from one period to another. Hence, using timers may not be easy and accurate as the task needs to evaluate the time for next period relative to the current time, whenever it finishes its execution. This is because preemption may happen between the time measurement and calling the sleep function.

In this project we need to support periodic activation of *servers* in order to implement the hierarchical scheduling framework. The reason for this is that we base our hierarchical scheduling framework around the periodic resource model [5], and a suitable implementation of the periodic resource model is achieved by the usage of a server based approach similar to the periodic servers [21, 22] that replenish their budget every constant period, i.e., the servers behave like periodic tasks.

7.4.2 Supporting arbitrary schedulers

There are two ways to support arbitrary schedulers in VxWorks:

1. Using the VxWorks custom kernel scheduler.

2. Using the original kernel scheduler and manipulating the ready queue by changing the priority of tasks and/or activating and suspending tasks.

In this paper, we are using the second approach since implementing the custom kernel scheduler is a relatively complex task compared with manipulating the ready queue. However, it will be interesting to compare between the two methods in terms of CPU overhead, and we leave this as a future work.

In the implementation of the second solution, we have used an Interrupt Service Routine (ISR) to manipulate the tasks in the ready queue. The ISR is responsible for adding tasks in the ready queue as well as changing their priorities according to the hierarchical scheduling policy in use. In the remainder of this paper, we refer to the ISR as the User Scheduling Routine (USR). By using the USR, we can implement any desired scheduling policy, including common ones such as Rate Monotonic (RM) and Earliest Deadline First (EDF).

7.5 The USR custom VxWorks scheduler

This section presents how to schedule periodic tasks using our scheduler, the User Scheduling Routine (USR).

7.5.1 Scheduling periodic tasks

When a periodic task finishes its execution, it changes its state to suspended by explicitly calling the suspend function. Then, to implement a periodic task, a

timer could be used to trigger the USR once every new task activation time to release the task (to put it in the ready queue).

The solution to use a timer triggering the USR once every new period can be suitable for systems with a low number of periodic tasks. However, if we have a system with n periodic tasks such a solution would require the use of n timers, which could be very costly or not even possible. In this paper we have used a scalable way to solve the problem of having to use too many timers. By multiplexing a single timer, we have used a single timer to serve n periodic tasks.

The USR stores the next activation time of all tasks (absolute times) in a sorted (according to the closest time event) queue called Time Event Queue (TEQ). Then, it sets a timer to invoke the USR at the time equal to the shortest time among the activation times stored in the TEQ. Also, the USR checks if a task misses its deadline by inserting the deadline in the TEQ. When the USR is invoked, it checks all task states to see if any task has missed its deadline. Hence, an element in the TEQ contains (1) the absolute time, (2) the id of task that the time belongs to, and (3) the event type (task next activation time or absolute deadline). Note that the size of the TEQ will be $2 * n * B$ bytes (where B is the size in bytes of one element in the TEQ) since we need to save the task's next period time and deadline time.

When the USR is triggered, it checks the cause of the triggering. There are two causes for the USR to be triggered: (1) a task is released, and (2) the USR will check for deadline misses. For both cases, the USR will do the following:

- Update the next activation and/or the absolute deadline time associated with the task that caused triggering of the USR in the TEQ and re-insert it in the TEQ according to the updated times.
- Set the timer equal to the shortest time in the TEQ so that the USR will be triggered at that time.
- For task release, the USR changes the state of the task to Ready. Also, it changes priorities of tasks if required depending on the scheduler (EDF or RM). For deadline miss checking, the USR checks the state of the task to see if it is Ready. If so, the task missed its deadline, and the deadline miss function will be activated.

Updating the next activation time and absolute deadline of a task in the TEQ is done by adding the period of the task that caused the USR invocation to the current absolute time. The USR does not use the system time as a time

reference. Instead it uses a time variable as a time reference. The reason for using a time variable is that we can, in a flexible manner, select the size of variables that save absolute time in bits. The benefits of such an approach is that we can control the size of the TEQ since it saves the absolute times, and it also minimizes the overhead of implementing 64 bits operations on 32 bit microprocessor [15], as an example. The reference time variable t_s used to indicate the time of the next activation, is initialized (i.e., $t_s = 0$) at the first execution of the USR. The value of t_s is updated every time that the USR executes and it will be equal to the time given by the TEQ that triggered the USR.

When a task τ_i is released for the first time, the absolute next activation time is equal to $t_s + T_i$ and its absolute deadline is equal to $t_s + D_i$.

To avoid time consuming operations, e.g., multiplications and divisions, that increase the system overhead inherent in the execution of the USR, all absolute times (task periods and relative deadlines) are saved in system tick unit (system tick is the interval between two consecutive system timer interrupts). However, depending on the number of bits used to store the absolute times, there is a maximum value that can be saved safely. Hence, saving absolute times in the TEQ may cause problems related to overrun of time, i.e., the absolute times become too large such that the value can not be stored using the available number of bits. To avoid this problem, we apply a wrapping algorithm which wraps the absolute times at some point in time, so the time will restart again. Periods and deadlines should not exceed the wrap-around value.

The input of the timer should be in a relative time, so evaluating the time at which to trigger the USR again (next time) is done by $TEQ[1] - t_s$ where $TEQ[1]$ is the first element in the queue after updating the TEQ as well as sorting it, i.e., the closest time in the TEQ. The USR checks to see if there are more than one task that have the same current activation time and absolute deadline. If so, the USR serves all these tasks to minimize the unnecessary overhead of executing the USR several times.

7.5.2 RM scheduling policy

Each task will have a fixed priority during run-time when Rate Monotonic (RM) is used, and the priorities are assigned according to the RM scheduling policy. If only RM is used in the system, no additional operations are required to be added to the USR since the kernel scheduler schedules all tasks directly according to their priorities, and the higher priority tasks can preempt the execution of the lower priority task. Hence, the implementation overhead for RM

will be limited to the overhead of adding a task in the ready queue and managing the timer for the next period (saving the absolute time of the new period and finding the shortest next time in the TEQ) for periodic tasks.

The schedulability analysis for each task is as follows [23];

$$\forall \tau_i \in \Gamma, 0 < \exists t \leq T_i \;\; \texttt{dbf}(i,t) \leq t. \tag{7.1}$$

And $\texttt{dbf}(i,t)$ is evaluated as follows

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

where $\texttt{HP}(i)$ is the set of tasks with priority higher than that of τ_i.

Eq. (7.2) can be easily modified to include the effect of using the USR on the schedulability analysis. Note that the USR will be triggered at the beginning of each task to release the task, so it behaves like a periodic task with priority equal to the maximum possible priority (the USR can preempt all application tasks). Checking the deadlines for tasks by using the USR will add more overhead, however, also this overhead has a periodic nature as the task release presented previously.

Eq. (7.3) includes the deadline and task release overhead caused by the USR in the response time analysis,

$$\begin{aligned} \texttt{dbf}(i,t) \;=\; C_i \;+ & \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k \;+ \sum_{\tau_j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil X_R \\ & + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t + T_j - D_j}{T_j} \right\rceil X_D \end{aligned} \tag{7.3}$$

where X_R is the worst-case execution time of the USR when a task is released and X_D is the worst-case execution time of the USR when it checks for deadline misses (currently, in case of deadline misses, the USR will only log this event into a log file).

7.5.3 EDF scheduling policy

For EDF, the priority of a task changes dynamically during run-time. At any time t, the task with shorter deadline will execute first, i.e., will have the highest priority. To implement EDF in the USR, the USR should update the priorities of all tasks that are in the Ready Queue when a task is added to the Ready

Queue, which can be costly in terms of overhead. Hence, on one hand, using EDF on top of commercial operating systems may not be efficient depending on the number of tasks, due to this sorting. However, the EDF scheduling policy provides, on other hand, better CPU utilization compared with RM, and it also has a lower number of context switches which minimizes context switch related overhead [24].

In the approach presented in this paper, tasks are already sorted in the TEQ according to their absolute times due to the timer multiplexing explained earlier. Hence, as the TEQ is already sorted according to the absolute deadlines, the USR can easily decide the priorities of the tasks according to EDF without causing too much extra overhead for evaluating the proper priority for each task.

The schedulability test for a set of tasks that use EDF is shown in Eq. (7.4) [25] which includes the case when task deadlines are allowed to be less than or equal to task periods.

$$\forall t > 0, \quad \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i \leq t \tag{7.4}$$

The overhead of implementing EDF can also be added to Eq. (7.4). Hence, Eq. (7.5) includes the overhead of releasing tasks as well as the overhead of checking for deadline misses.

$$\begin{aligned} \forall t > 0, \quad & \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil X_R \\ & + \sum_{\tau_j \in \Gamma} \left\lceil \frac{t + T_j - D_j}{T_j} \right\rceil X_D \leq t \end{aligned} \tag{7.5}$$

7.5.4 Implementation and overheads of the USR

To implement the USR, we have used the following VxWorks service functions;

- Q_PUT - insert a node into a multi-way queue (ready queue).
- Q_REMOVE - remove a node from a multi-way queue (ready queue).
- taskCreat - create a task.
- taskPrioritySet - set a tasks priority.

We present our initial results inherent in the implementation of the USR, implementing both the Rate Monotonic (RM) scheduler as well as the Earliest Deadline First (EDF) scheduler. The implementations were performed on a ABB robot controller with a Pentium 200 MHz processor running the VxWorks operating system version 5.2. To trigger the USR for periodic tasks, we have used watchdog timers where the next expiration time is given in number of ticks. The watchdog uses the system clock interrupt routine to count the time to the next expiration. The platform provides system clock with resolution equal to $4500 ticks/s$. The measurement of the execution time of the USR is done by reading a timestamp value at the start as well as at the end of the USR's execution. Note that the timestamp is connected to a special hardware timer with resolution $12000000 ticks/s$.

Table 7.1 shows the execution time of the USR when it performs RM and EDF scheduling, as well as deadline miss checking, as a function of the number of tasks in the system. The worst case execution time for USR will happen when USR deletes and then inserts all tasks from and to TEQ and to capture this, we have selected a same period for all tasks. The table shows the minimum, maximum and average out of 50 measured values. Comparing between the results of the three cases (EDF, RM, deadline miss), we can see that there is no big difference in the execution time of the USR. The reason for this result is that the execution of the USR for EDF, RM and deadline miss checking all includes the overhead of deletion and re-inserting the tasks in the TEQ, which is the dominating part of the overhead. As expected, EDF causes the largest overhead because it changes the priority of all tasks in the ready queue during run-time. Figures 7.2-7.3 show that EDF imposes between $6 - 14\%$ extra overhead compared with RM.

7.6 Hierarchical scheduling

A Hierarchical Scheduling Framework (HSF) supports CPU sharing among subsystems under different scheduling policies. Here, we consider a two-level scheduling framework consisting of a global scheduler and a number of local schedulers. Under global scheduling, the operating system (global) scheduler allocates the CPU to subsystems. Under local scheduling, a local scheduler inside each subsystem allocates a share of the CPU (given to the subsystem by the global scheduler) to its own internal tasks (threads).

We consider that each subsystem is capable of exporting its own interface that specifies its collective real-time CPU requirements. We assume that such a

Number of tasks	X_R (RM)			X_R (EDF)			X_D (Deadline miss check)		
	Max	Average	Min	Max	Average	Min	Max	Average	Min
10	71	65	63	74	70	68	70	60	57
20	119	110	106	131	118	115	111	100	95
30	172	158	155	187	172	169	151	141	137
40	214	202	197	241	228	220	192	180	175
50	266	256	249	296	280	275	236	225	219
60	318	305	299	359	338	331	282	268	262
70	367	352	341	415	396	390	324	309	304
80	422	404	397	476	453	444	371	354	349
90	473	459	453	539	523	515	415	398	393
100	527	516	511	600	589	583	459	442	436

Table 7.1: USR execution time in μs, the maximum, average and minimum execution time of 45 measured values for each case.

subsystem interface is in the form of the periodic resource model (P_s, Q_s) [5]. Here, P_s represents a *period*, and Q_s represents a *budget*, or an execution time requirement within the period $(Q_s < P_s)$. By using the periodic resource model in hierarchical scheduling frameworks, it is guaranteed [5] that all timing constraints of internal tasks within a subsystem can be satisfied, if the global scheduler provides the subsystem with CPU resources according to the timing requirements imposed by its subsystem interface. We refer interested readers to [5] for how to derive an interface (P_s, Q_s) of a subsystem, when the subsystem contains a set of internal independent periodic tasks and the local scheduler follows the RM or EDF scheduling policy. Note that for the derivation of the subsystem interface (P_s, Q_s), we use the demand bound functions that take into account the overhead imposed by the execution of USR (see Eq. (7.3) and (7.5)).

7.6.1 Hierarchical scheduling implementation

Global scheduler: A subsystem is implemented as a periodic server, and periodic servers can be scheduled in a similar way as scheduling normal periodic tasks. We can use the same procedure described in Section 7.5 with some modifications in order to schedule servers. Each server should include the following information to be scheduled: (1) server period, (2) server budget, (3) remaining

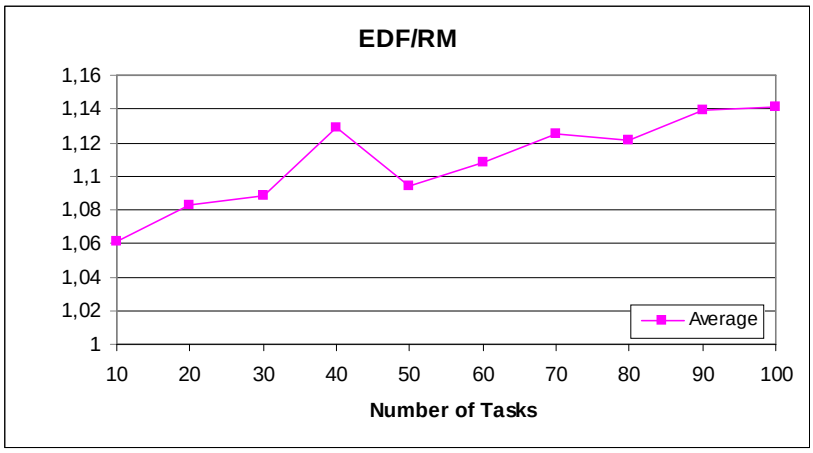

Figure 7.2: EDF normalized against RM, for average USR execution time.

budget, (4) pointer to the tasks that belong to this server, and (5) the type of the local scheduler (RM or EDF) (6) local TEQ. Moreover, to schedule servers we need:

- **Server Ready Queue** to store all servers that have non zero remaining budget. When a server is released at the beginning of its period, its budget will be charged to the maximum budget Q, and the server will be added to the Server Ready Queue. When a server executes its internal tasks for some time x, then the remaining budget of the server will be deceased with x, i.e., reduced by the time that the server execute. If the remaining budget becomes zero, then the server will hand over the control to the global scheduler to select and remove the highest priority server from Server Ready Queue.

- **Server TEQ** to release the server at its next absolute periodic time since we are using periodic servers and also track their remaining budgets.

Figures 7.4 illustrates the implementation of HSF in VxWorks. The Server Ready Queue is managed by the routine that is responsible for scheduling the servers. Tracking the remaining budget of a server is solved as follows; whenever a server starts running, it sets an absolute time at which the server budget

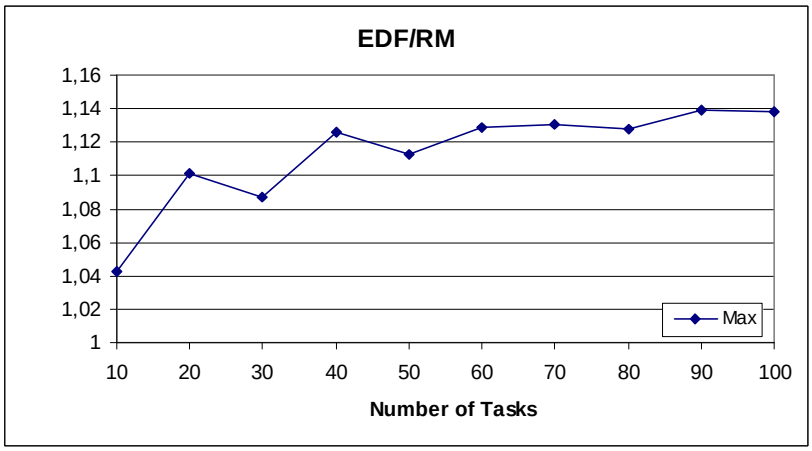

Figure 7.3: EDF normalized against RM, for maximum USR execution time.

expire and it equals to the current time plus its remaining budget. This time is added to the server event Queue to be used by the timer to trigger an event when the server budget expires. When a server is preempted by another server, it updates the remaining budget by subtracting the time that has passed since the last release. When the server executes its internal tasks until the time when the server budget expiry event triggers, it will set its remaining budget to zero, and the scheduling routine removes the server from the Server Ready Queue.

Local scheduler: When a server is given the CPU resources, the ready tasks that belong to the server will be able to execute. We have investigated two approaches to deal with the tasks in the Ready Queue when a server is given CPU resources:

- All tasks that belong to the server that was previously running will be removed from the Ready Queue, and all ready tasks that belong to the new running server will be added to the Ready Queue, i.e., swapping of the servers' task sets. To remove tasks from the Ready Queue, the state of the tasks is changed to suspend state. However, this will cause a problem since the state of the tasks that finish their execution is also changed to suspend and when the server run again it will add non-ready

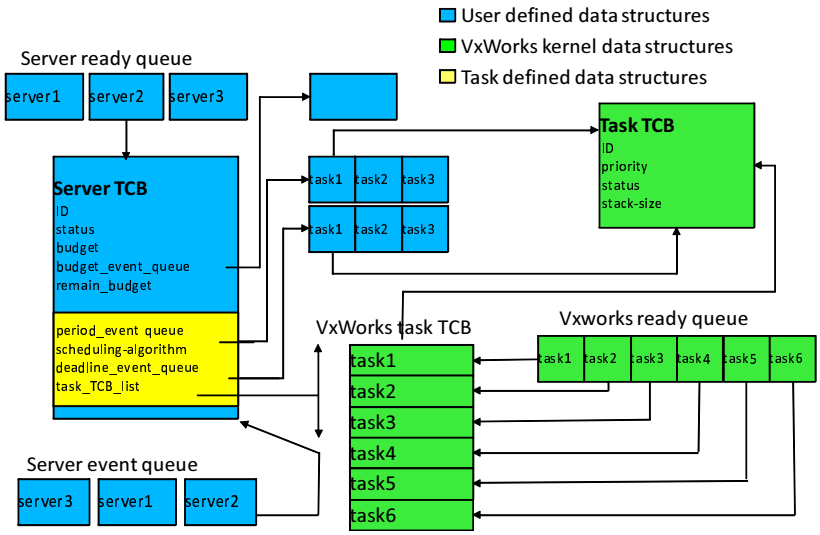

Figure 7.4: The implementation of HSF in VxWorks.

tasks to the Ready Queue. To solve this problem, an additional flag is used in the task's TCB to denote whether the task was removed from Ready Queue and enter to suspend state due to budget expiration of its server or due to finishing its execution.

- The priority of all tasks that belong to the preempted server will be set to a lower (the lowest) priority, and the priority of all tasks that belong to the new running server will be raised as if they were executing exclusively on the CPU, scheduled according to the local scheduling policy in use by the subsystem.

The advantage of the second approach is that it can give the unused CPU resources to tasks that belong to other servers. However, the disadvantage of this approach is that the kernel scheduler always sorts the tasks in the Ready Queue and the number of tasks inside Ready Queue using the second approach will be higher which may impose more overhead for sorting tasks. In this paper, we consider the first approach since we support only periodic tasks. When a server is running, all interrupts that are caused by the local TEQ, e.g.,

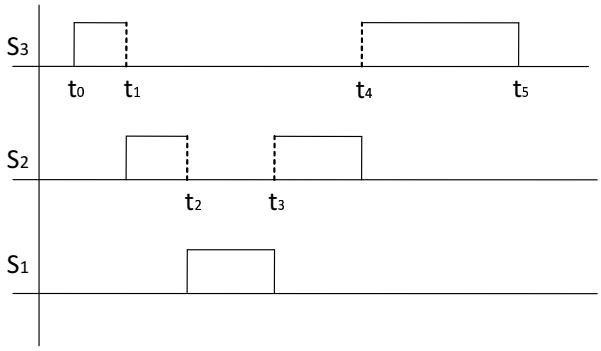

Figure 7.5: Simple servers execution example.

releasing tasks and checking deadline misses, can be served without problem. However, if a task is released or its deadline occurs during the execution of another server, the server that includes the task, may miss this event. To solve this problem, when the server starts running after server preemption or when it finishes its budget, it will check for all past events (including task release and deadline miss check events) in the local TEQ that have absolute time less than the current time, and serve them.

Note that the time wrapping algorithm described in section 7.5.1 should take into account all local TEQ's for all servers and the server event queue, because all these event queues share the same absolute time.

Figure 7.5 illustrates the implementation of hierarchical scheduling framework which includes an example with three servers S_1, S_2, S_3 with global and local RM schedulers, the priority of S_1 is the highest and the priority of S_3 is the lowest. Suppose a new period of S_3 starts at time t_0 with a budget equal to Q_3. Then, the USR will change the state of S_3 to Ready, and since it is the only server that is ready to execute, the USR will;

- add the time at which the budget will expire, which equals to $t_0 + Q_3$, into the server event queue and also add the next period event in the server event queue.
- check all previous events that have occurred while the server was not active by checking if there are task releases or deadline checks in the time interval of $[t^*, t_0]$, where t^* is the latest time at which the budget of S_3 has been expired.

- start the local scheduler.

At time t_1 the server S_2 becomes Ready and it has higher priority than S_3. So S_2 will preempt S_3 and in addition to the previously explained action, the USR will remove all tasks that belong to S_3 from the ready queue and save the remaining budget which equals to $Q_3 - (t_1 - t_0)$. Also the USR will remove the budget expiration event from the server event queue. Note that when S_3 executes next time it will use the remaining budget to calculate the budget expiration event.

Number of servers	Max	Average	Min
10	91	89	85
20	149	146	139
30	212	205	189
40	274	267	243
50	344	333	318
60	412	400	388
70	483	466	417
80	548	543	509
90	630	604	525
100	689	667	570

Table 7.2: Maximum, average and minimum execution time of the USR with 100 measured values as a function of the number of servers.

The USR execution time depends on the number of the servers, and the worst case happens when all servers are released at the same time. In addition, the execution time of the USR also depends on the number of ready tasks in both the currently running server to be preempted as well as the server to preempt. The USR removes all ready tasks that belong to the preempted server from ready queue and adds all ready tasks that belong to the preempting server with highest priority into the ready queue. Here, the worst case scenario is that all tasks of both servers are ready at that time. Table 7.2 shows the execution time of the USR (when a server is released) as a function of the number of servers using RM as a global scheduler at the worst case, where all the servers are released at the same time, just like the case shown in the previous section. Here, we consider that each server has a single task in order to purely investigate the effect of the number of servers on the execution time of the USR.

7.6.2 Example

In this section, we will show the overall effect of implementing the HSF using a simple example, however, the results from the following example are specific for this example because, as we showed in the previous section, the overhead is a function of many parameters affect the number of preemptions such as number of servers, number of tasks, servers periods and budgets. In this example we use RM as both local and global scheduler, and the servers and associated tasks parameters are shown in Table 7.3. Note that $T_i = D_i$ for all tasks.

$S_1(P_1 = 5, Q_1 = 1)$			$S_2(P_2 = 6, Q_2 = 1)$			$S_3(P_3 = 70, Q_3 = 20)$		
τ_i	T_i	C_i	τ_i	T_i	C_i	τ_i	T_i	C_i
τ_1	20	1	τ_1	25	1	τ_1	140	7
τ_2	25	1	τ_2	35	1	τ_2	150	7
τ_3	30	1	τ_3	45	1	τ_3	300	30
τ_4	35	1	τ_4	50	1			
τ_5	40	7	τ_5	55	7			
-	-	-	τ_6	60	7			

Table 7.3: System parameters in μs.

The measured overhead utilization is about 2.85% and the measured release jitter for task τ_3 in server S_3 (which is the lowest priority task in the lowest priority server) is about $49ms$. The measured worst case response time is $208.5ms$ and the finishing time jitter is $60ms$. These results indicate that the overhead and performance of the implementation are acceptable for further development in future project.

7.7 Summary

This paper has presented our work on the implementation of our hierarchical scheduling framework in a commercial operating system, VxWorks. We have chosen to implement it in VxWorks so that it can easily be tested in an industrial setting, as we have a number of industrial partners with applications running on VxWorks and we intend to use them as case studies for an industrial deployment of the hierarchical scheduling framework.

This paper demonstrates the feasibility of implementing the hierarchical scheduling framework through its implementation over VxWorks. In partic-

ular, it presents several measurements of overheads that its implementation imposes. It shows that a hierarchical scheduling framework can effectively achieve the clean separation of subsystems in terms of timing interference (i.e., without requiring any temporal parameters of other subsystems) with reasonable implementation overheads.

In the next stage of this implementation project, we intend to implement synchronization protocols in hierarchical scheduling frameworks, e.g., [12]. In addition, our future work includes supporting sporadic tasks in response to specific events such as external interrupts. Instead of allowing them to directly add their tasks into the ready queue, we consider triggering the USR to take care of such additions. We also plan to support aperiodic tasks while bounding their interference to periodic tasks by the use of some server-based mechanisms. Moreover, we intend to extend the implementation to make it suitable for more advanced architectures including multicore processors.

Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers for their helpful comments, as well as to Clara Maria Otero Pérez for detailed information regarding the implementation of hierarchical scheduling as a dedicated layer on top of pSoSystem.

Bibliography

[1] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *18th IEEE International Real-Time Systems Symposium*, pages 308–319, Dec 1997.

[2] T.-W. Kuo and C.H. Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *20th IEEE International Real-Time Systems Symposium*, pages 256–267, Dec 1999.

[3] G. Lipari and S. Baruah. Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems. In *6th IEEE Real Time Technology and Applications Symposium*, pages 166–175, May 2000.

[4] A. Mok, X. Feng, and D. Chen. Resource Partition for Real-Time Systems. In *7th Real-Time Technology and Applications Symposium*, pages 75–84, May 2001.

[5] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *24th IEEE International Real-Time Systems Symposium*, pages 2–13, Dec 2003.

[6] L. Almeida and P. Pedreiras. Scheduling within Temporal Partitions: Response-Time Analysis and Server Design. In *4th ACM International Conference On Embedded Software*, pages 95–103, Sep 2004.

[7] Rob Davis and Allan Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *26th IEEE International Real-Time Systems Symposium*, pages 389–398, Dec 2005.

[8] X. Feng and A. Mok. A Model of Hierarchical Real-Time Virtual Resources. In *23rd IEEE International Real-Time Systems Symposium*, pages 26–35, Dec 2002.

[9] G. Lipari and E. Bini. Resource Partitioning Among Real-Time Applications. In *15th IEEE Euromicro Conference on Real-Time Systems*, pages 151–158, July 2003.

[10] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of Hierarchical Fixed-Priority Scheduling. In *14th IEEE Euromicro Conference on Real-Time Systems*, pages 152–160, July 2002.

[11] Insik Shin and Insup Lee. Compositional Real-Time Scheduling Framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, Dec 2004.

[12] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems. In *7th ACM International Conference On Embedded Software*, pages 279–288, Oct 2007.

[13] R. I. Davis and A. Burns. Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems. In *27th IEEE International Real-Time Systems Symposium*, pages 257–270, Dec 2006.

[14] Nathan Fisher, Marko Bertogna, and Sanjoy Baruah. The Design of an EDF-Scheduled Resource-Sharing Open Environment. In *28th IEEE International Real-Time Systems Symposium*, pages 83–92, Dec 2007.

[15] G. Buttazzo and P. Gai. Efficient Implementation of an EDF Scheduler for Small Embedded Systems. In *2nd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2006.

[16] Evidence Srl. ERIKA Enterprise RTOS.

[17] C. Diederichs, U. Margull, F. Slomka, and G. Wirrer. An Application-Based EDF scheduler for OSEK/VDX. In *11th Conference on Design, Automation and Test in Europe*, pages 1045–1050, Aug 2008.

[18] D. Kim, Y. Lee, and M. Younis. SPIRIT-uKernel for Strongly Partitioned Real-Time Systems. In *7th International Workshop on Real-Time Computing and Applications Symposium*, pages 73–80, Dec 2000.

[19] L. Kinnan P. Parkinson. Safety Critical Software Development for Integrated Modular Avionics. In *Wind River white paper*, 2007.

[20] Jane W.S. Liu. Real-time Systems. *Prentice Hall*, page 610, 2000.

[21] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *8th IEEE International Real-Time Systems Symposium*, pages 261–270, Dec 1987.

[22] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Journal of Real-Time Systems*, 1(1):27–60, June 1989.

[23] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *10th IEEE International Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[24] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgement day. *Journal of Real-Time Systems*, 29(1):5–26, Jan 2005.

[25] S. Baruah, R. Howell, and L. Rosier. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Journal of Real-Time Systems*, 2(4):301–324, Nov 1990.

Chapter 8

Paper C: A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux

Mikael Åsberg, Thomas Nolte and Shinpei Kato
In RTCSA'11 Conference, pages 380–387, August, 2011

Abstract

This paper presents a Hierarchical Scheduling Framework (HSF) recorder for Linux-based operating systems. The HSF recorder is a loadable kernel module that is capable of recording tasks and servers without requiring any kernel modifications. Hence, it complies with the reliability and stability requirements in the area of embedded systems where proven versions of Linux are preferred. The recorder is built upon the loadable real-time scheduler framework RESCH (REal-time SCHeduler). We evaluate our recorder by comparing the overhead of this solution against another (patched) recorder. Also, the tracing accuracy of the HSF recorder is tested by running a media-processing task together with periodic real-time Linux tasks in combination with servers. The tests are recorded with the HSF recorder, and the Ftrace recorder, in order to show the correctness of the experiments and the HSF recorder itself.

8.1 Introduction

The research that we conduct is primarily focused on the development of hierarchical scheduling [1, 2, 3]. Our previous and ongoing work within hierarchical scheduling includes practical (implementation) aspects of this kind of scheduling [4, 5], the applicability/usage [6, 7] of it, as well as applying formal methods [8] on it. In server-based scheduling (the predecessor of hierarchical scheduling), tasks (a sequence of instructions) are only allowed to execute whenever their server (the virtual task which they belong to) runs. The server itself executes according to some scheduling scheme (global scheduling) which is independent of the tasks. The advantage is that it can improve the response time (the time length between task activation and completion) of event triggered tasks, and still keep the scheduling deterministic since the server scheduling parameters are known and included in the schedulability analysis. Further, introducing a scheduler within each server (local scheduling) makes it more general since it supports time triggered tasks as well. This can be generalized even further by representing a task as a set of tasks together with a scheduler. When we have separate scheduling inside a server, i.e. both global and local scheduling, then we refer to hierarchical scheduling or a Hierarchical Scheduling Framework (HSF), this is illustrated in Figure 8.1.

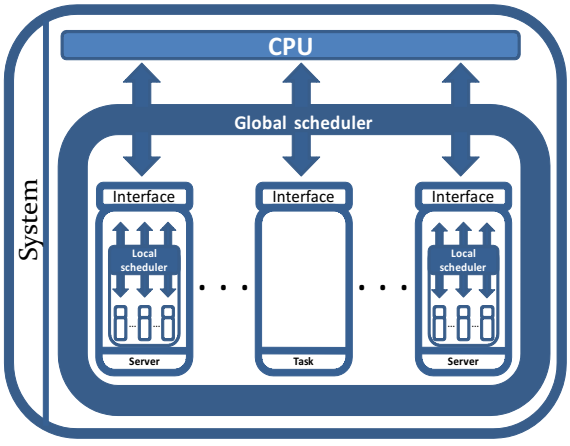

Figure 8.1: Hierarchical scheduling framework.

Hierarchical scheduling has several advantages, besides improving response time of event triggered tasks. It enables parallel development of system parts

(subsystems), simplifies integration of subsystems (analysis), supports runtime temporal partitioning and safe execution of tasks etc. However, except for ARINC653 [9, 10] compliant operating systems that are commonly found in avionics applications, hierarchical scheduling is rarely an integrated part of an operating system (OS). Indeed, there is a need to develop/implement new scheduling algorithms, such as hierarchical scheduling, in the area of embedded and/or real-time systems [6]. A motivation of this can be found in our scheduling example in the evaluation (Section 8.4), where we let a media-processing task (which does a movie playback) execute within a server (server-based scheduling). The server executes with a certain frequency, giving (guaranteeing) the media task an even amount of CPU power which improves the playback quality of the movie, even though it executes among other time triggered tasks. The media task has an unknown execution pattern, i.e., the releases are undefined. Still, we get predictability (since we can analyze the behavior) from both the media tasks point of view, and the time triggered tasks. Also, we avoid (temporal) interference at runtime, meaning that we get a safe execution environment for the tasks because temporal errors do not propagate between the media task and the time triggered tasks.

From a practical point of view, it is an advantage if hierarchical scheduling can be implemented easily/efficiently and without modifying the kernel. The latter makes it easier for both developers and users since there is no need to maintain/apply kernel modifications every time the kernel is replaced or updated. Moreover, keeping the scheduler isolated in a kernel module, without modifying the kernel, simplifies debugging and potential certification of its correctness (component-based development advantages). We see that the RESCH scheduling framework [11] is useful because it has the advantages mentioned, since it does not need any kernel modifications. Also, it makes scheduler development easier because it simplifies the scheduling interface to the user and it supports the development of schedulers (plugins) which run as independent kernel modules. However, while the development of schedulers are simplified with this framework, it lacks support for debugging the schedulers. That is why we have developed a HSF recorder, which can easily be plugged in to a server-based/hierarchical scheduler, developed in RESCH. The recorder does not require kernel modifications and it is of course also suitable for analyzing the runtime behavior of tasks/servers since the recorded trace can be visualized graphically with the Tracealyzer [12] or Grasp [13] visualization tools. In turn, these tools can present valueable trace data such as execution- and response-time.

The HSF recorder is able to record the following scheduling events during

run-time:

1. The time instance when a task/server is released (even though it might not start to execute).
2. The time instance when a task/server starts to execute.
3. When there is a task/server context switch, the recorder distinguishes between preemption and non-preemption.
4. The time instance when a task/server finishes its execution.

Contribution The main contributions of this paper are:

1. We have implemented a task/server recorder with the use of RESCH, i.e., it does not require any kernel modifications. The recorder enables debugging at task and server level, in Linux based real-time/general-purpose OSs.
2. We have evaluated our HSF recorder by implementing yet another recorder (Section 8.2.3), using the technique presented in [14], and compared the overhead of this recorder, with the HSF recorder.
3. We have tested our recorder by running a media-processing task together with time triggered tasks and servers. The example shows how the play-back quality gets improved by putting the media-processing task in a server. The HSF recorder is used in this example to debug and display the runtime behavior.

Outline The outline of this paper is as follows: Section 8.2 presents preliminary background, in Section 8.3 we describe the HSF-recorder implementation. Section 8.4 evaluates the overhead and tracing accuracy of the HSF recorder. Section 8.5 presents related work, and finally, Section 8.6 concludes.

8.2 Preliminaries

8.2.1 System model

We assume fixed-priority, preemptive, scheduling of periodic tasks, according to the periodic task model [15]. A task i is presumed to have the following parameters, $\langle T_i, WCET_i, D_i, pr_i \rangle$, where the period T_i represents the frequency

in which the task is released for execution, $WCET_i$ is the worst case execution time of the task, the relative deadline D_i (within the period) is when the task must complete its execution (RESCH monitors this) and pr_i is the task priority (lower value represents higher priority). Also, all tasks are assumed to execute independently of eachother and on the same core, i.e., single core.

The servers are also assumed to have fixed priority and they are scheduled preemptively and periodic. A server j has similar parameters as tasks, i.e. $\langle P_j, Q_j, pr_j \rangle$, where P_j is the server period, Q_j is defined as a budget (which is the time given at each period P_j to the tasks within the server) and pr_j is the server priority (lower value represents higher priority).

8.2.2 RESCH

We have been developing a loadable real-time scheduler framework, RESCH [11], designed to work with the POSIX-compliant SCHED_FIFO scheduling policy implementation. RESCH has previously been used as the basis for another scheduler called AIRS [16] - a multi-core CPU scheduler for interactive real-time applications. As mentioned previously, RESCH is a modification-free scheduling framework for Linux. It supports periodic tasks which can be scheduled in a fixed-priority preemptive manner. RESCH is simply composed of external kernel modules and user-space libraries for easy installation. It gives both an interface to the users in user space (e.g. a task specific interface like `rt_wait_for_period()`) as well as in the kernel space. The kernel space API (Application Programming Interface) has the interface shown below:

1. **task_run_plugin( )**
2. **task_exit_plugin( )**
3. **job_release_plugin( )**
4. **job_complete_plugin( )**

These functions can be implemented by a **RESCH plugin** (Figure 8.3), i.e., a kernel module that has access to the RESCH kernel API. These functions are called in the **RESCH core** at certain events which are illustrated in Figure 8.2. Functions 1) and 2) are executed every time a task registers/unregisters to RESCH. With register we mean that the task does a RESCH API call, transforming it to a **RESCH task**, which creates a RESCH TCB (Task Control Block) and puts it in the RESCH ready-queue etc. A RESCH TCB has, among

other real-time specific data, a reference to its corresponding Linux task TCB (`task_struct`). Once the task is registered in RESCH, it will be scheduled periodically (and preemptive) according to its real-time priority. The primitives 3) and 4) are called whenever a RESCH task is released for execution or when it has finished its execution. The plugins get these scheduling notifications and can thereby affect scheduling, trace tasks etc. The plugin notifications are shown in Figure 8.2. When a task notifies RESCH that it has finished its execution in its current period, the RESCH core will inform any plugin about this event and set a timer for the release of the tasks next period. As a last step, it will call the Linux kernel to re-schedule another task. The next running task might be a RESCH task or any other Linux process.

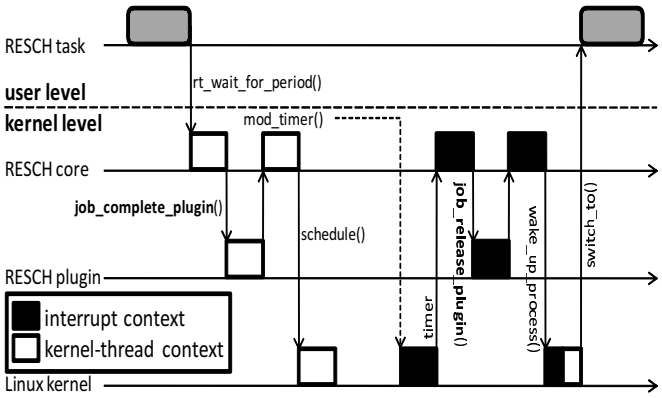

Figure 8.2: RESCH control flow.

When the kernel responds to the corresponding timeout (task release), a handler in the RESCH core will get notified about this event. The handler will notify any plugin about the task release and then call the kernel to wake up the task.

In Linux, since kernel version 2.6.23 (October of 2007), tasks can be either a $fair$ or a $real\text{-}time$ task. The latter group has higher priority (0-99 where 0 is highest) than fair tasks (100-140). A task that registers to RESCH is automatically transformed to a real-time task. RESCH is responsible for releasing tasks, and tasks registered to RESCH must notify when they have finished their execution in the current period. In this way, RESCH can control the scheduling. RESCH uses an absolute-time clock, i.e., it does not wrap around. Also, release times are stored as absolute values, so release patterns are exact.

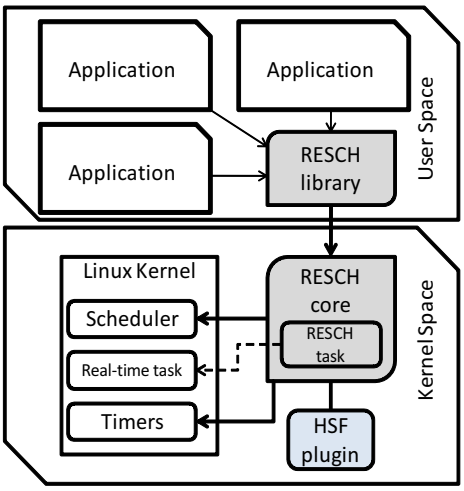

Figure 8.3: RESCH framework.

The cost of having a modification-free solution is that RESCH can only see scheduling events related to its registered tasks. Real-time tasks with higher priority than RESCH tasks (i.e. tasks that are not registered in RESCH) can thereby interfere with RESCH tasks without the RESCH core being able to detect it. A simple solution to this problem is to schedule all real-time tasks with the RESCH framework.

8.2.3 Task-switch hook patch

Our previous work [14] includes an implementation of a `task_switch_hook` function (Figure 8.4), residing in a kernel module, which is called by the Linux scheduler at every scheduler tick. In this way, it is possible to record task scheduling events. This solution requires modification of two code lines in two separate kernel source files (`sched_rt.c` and `sched_fair.c`). The modification of file `sched_rt.c` is illustrated in Figure 8.4 (a similar change is done in `sched_fair.c`). Linux has (since kernel version 2.6.23) two scheduling classes, namely the $fair$ and the $real\text{-}time$ scheduling classes. When a new task should be released, the Linux scheduler iterates through its scheduling classes (first the $real\text{-}time$ class, secondly the $fair$ class) in order to find the next task to release.

The modification (Figure 8.4) makes it possible to re-direct a scheduling class' function pointer `.pick_next_task` to point to a user defined function (i.e., our function `task_switch_hook`), instead of the original function `pick_next_task_rt`. Our function will instead point to `pick_next_task_rt`, in this way, we do not alter the kernel functionality other than executing our function `task_switch_hook` (which contains user defined code) just before `pick_next_task_rt` starts to execute. Our function (hook) can be inserted and removed during runtime. A task recorder can easily be implemented (as a kernel module) and use the `task_switch_hook` function to register task context switches, however, the kernel must be modified.

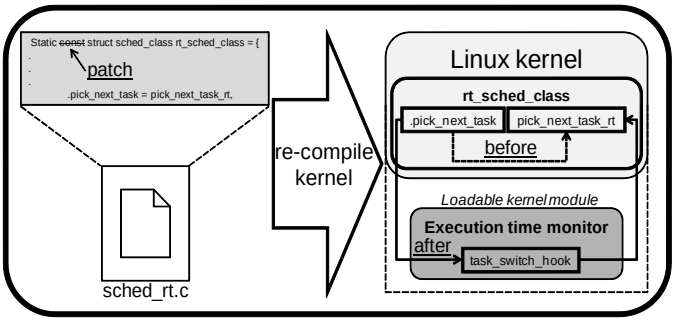

Figure 8.4: Hook patch.

8.3 Implementation

The implementation of the HSF recorder is based on the scheduler plugin HSF which in turn is based on the scheduling framework RESCH. Figure 8.5 shows that the HSF scheduler uses primitives exported by RESCH and exports these, as well as server specific primitives, to the recorder. These primitives are used to register server and task context switches. Note that the flexible structure allows for new scheduler plugins to reuse the recorder as long as they export the same primitives.

For the recording to work correctly, it is assumed that no higher priority $real\text{-}time$ Linux tasks, which are not registered by RESCH, are executed.

The current implementation does not support $load\ balancing$ (a function in Linux that migrates tasks to other CPUs based on load). This is because the

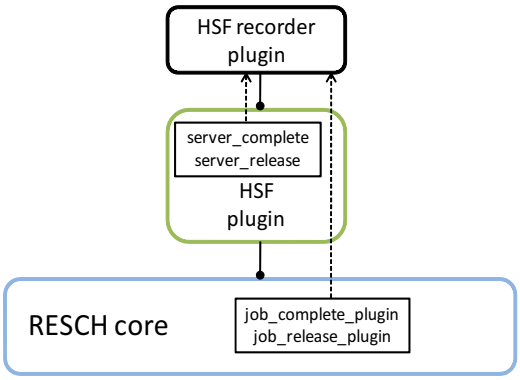

Figure 8.5: HSF-recorder plugin.

RESCH scheduler cannot detect task migrations made by the Linux scheduler.

Each recorded event has 2 records:

- ID of the next task/server to execute.
- Timestamp of the event.

The ID of the next task/server is used to calculate the previous task/server. The 4 hook functions (Figure 8.5) are used by the recorder to save scheduling records in memory (this is a circular implementation). The recorder flushes the recorded data to disk when it gets unloaded by the user. The recording format can easily be converted to match any visualization tool. We have successfully converted the format to fit with the Tracealyzer [12] and the Grasp [13] visualization tools. We use Grasp in the evaluation (Section 8.4) in order to visualize the trace of the HSF recorder since it also supports hierarchical scheduling in addition to regular (flat) scheduling.

Figure 8.6 illustrates how the HSF recorder gets triggered. As can be seen, the HSF scheduler gets triggered by its own timers as well as by the RESCH core. The HSF scheduler relays task releases and completions to the HSF recorder when the HSF scheduler itself is triggered by the RESCH core. Whenever the HSF scheduler gets triggered by a timer, it automatically calls its server release/completion plugin, which in turn starts the recorder. The figure also shows that the HSF recorder executes mostly in interrupt context. This makes it less expensive in terms of context-switch overhead.

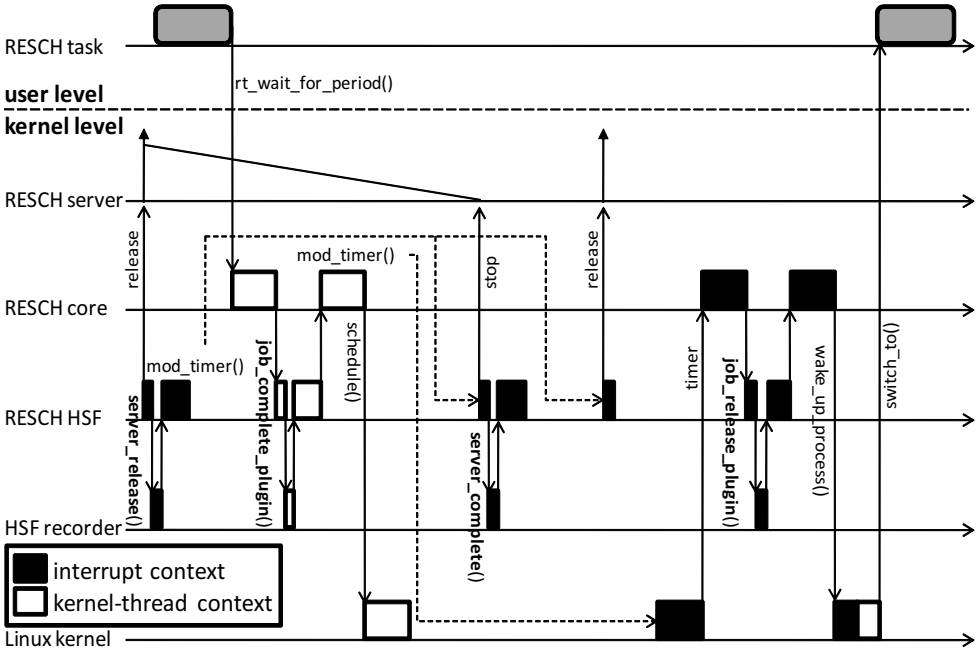

Figure 8.6: HSF recorder control flow.

8.4 Evaluation

We have evaluated our HSF recorder by recording a set of tasks and servers (Table 8.1 and 8.2). In our example, task `rt_task1` belongs to server `Server0`, `rt_task2` and `rt_task3` does not belong to any server while `rt_task4` belong to server `Server1` and `rt_task5` to `Server2`.

The evaluation shows two aspects: the measured overhead (section 8.4.1) of the HSF recorder compared to the patched recorder [14], and an example of how the Quality of Service (QoS) of multimedia tasks can be improved with hierarchical scheduling as well as how our HSF recorder can assist in this work (section 8.4.2). In the multimedia example we used our HSF recorder and the Ftrace [17] recorder.

During our experiments, the two recorders were recording the tasks and servers simultaneously.

Task-name	T	$WCET$	D	$Prio$	$Server$
rt_task1	80	9	80	0	Server0
rt_task2	200	75	200	1	-
rt_task3	105	9	105	2	-
rt_task4	500	100	500	3	Server1
rt_task5	-	-	-	4	Server2

Table 8.1: Tasks used in the evaluation.

Server-name	P	Q	$Prio$
Server0	40	6	1
Server1	90	23	2
Server2	25	8	0

Table 8.2: Servers used in the evaluation.

8.4.1 Overhead measurements

In order to estimate the overhead impact, we measured the execution time of the patched and the HSF recorder, running simultaneously and recording the same trace. We also noted the amount of data (in kilo bytes) that the two recorders produced (out of curiosity we also measured Ftrace). We implemented an optimized version of the patched recorder, **Patch** (Table 8.3) so that it only saved recorded data of the tasks that we were interested in recording. In this way, the comparison to the HSF recorder became fair since it is only triggered at task/server events related to the tasks/servers we are interested in recording (RESCH related task and servers).

Recorder	Exec. time (μs)	Rec. data (KB)
HSF	45	10.5
Patch	1246	17.4
Ftrace	-	888.6

Table 8.3: Measurements of the recorders.

The values listed in Table 8.3 are the average measured values of 10 runs and the recorders recorded about 4 seconds at each run. We see that the HSF

recorder has a ratio of 4.3 µ*s*/KB while Patch has 71.6 µ*s*/KB. The conclusion is that the HSF recorder produces less overhead than the patched recorder, comparing the execution-time/data ratio. The small amount of recorded data compared to Ftrace suggests that our recorder might be a better option if the user is only interested in a subset of tasks. Having a small amount of overhead is attractive for recorders since they can remain active in shipped products (without wasting too much resources), and thereby eliminating the probe effect.

8.4.2 Multimedia example

The purpose of this example is to show how a multimedia task (processing a movie) can benefit from hierarchical scheduling in such a way that the movie playback runs more smoothly. The HSF scheduler has never been evaluated (and debugged) as properly as the example we are about to show, so this is a good case study for the HSF recorder. We run the multimedia task in different setups (with and without hierarchical scheduling), and measure its performance. The hierarchical scheduling gives the multimedia task an even amount of CPU power, and thereby improves the movie playback. Note that all of this is done, including the recording, without modifying the kernel. The HSF recorder plays a key role since knowledge of the scheduling behavior is important in order for the result of this evaluation to be correct. For example, the recorder shows that the tasks and servers get the amount of CPU that we specify (i.e., that tasks run within their servers) and that the tasks/servers run according to the specified frequency and $WCET/Q$. During our experiments, the recording showed that the HSF cannot keep tasks within their server if they do a lot of blocking (e.g. multimedia tasks). Therefore, we set lowest priority to the multimedia task and add idle tasks with higher priority than the multimedia task. This will keep the multimedia task within its server, thereby guaranteeing the upper limit on its resource supply. This was confirmed by the recording of our HSF recorder. A second recorder (Ftrace) was also used in order to show that the HSF recorder recorded correctly. We used the Grasp tool [13] to visualize our recordings (for both the HSF recorder and Ftrace), since it can display both tasks and servers.

In this example, we have 5 tasks, i.e., `rt_task1` to `rt_task5` (Table 8.1). Tasks `rt_task1` to `rt_task4` are dummy tasks, i.e., they just loop (`rt_task1` in Figure 8.7). `rt_task5` does a movie playback, its task body is shown in Figure 8.7.

```
// rt_task1
int main(int argc, char *argv[ ])
{
        .
        .
        for (i = 0; i < NR_OF_JOBS; i++) {
            for (j = 0; j < USEC_UNIT; j++) {
            }
            if (!rt_wait_for_period()) {
                printf("deadline is missed!\n");
            }
        }
        .
        .
}
// rt_task5
int main (int argc, char *argv[ ])
{
        .
        .
        libvlc_media_player_play(player);
        .
        .
}
```

Figure 8.7: Task bodies.

`rt_task5` used the libVLC[1] for movie playback and the library itself has the nice property that the movie processing can be executed by a task running in $real\text{-}time$ mode. We executed `rt_task5` in 4 different setups:

1. `rt_task5` with lowest priority and tasks `rt_task1` to `rt_task4` with priority order as in Table 8.1.

2. `rt_task5` with medium priority (in between `rt_task2` and `rt_task3`) and tasks `rt_task1` to `rt_task4` with priority order as in Table 8.1.

3. `rt_task5` with highest priority and tasks `rt_task1` to `rt_task4` with priority order as in Table 8.1.

4. `rt_task5` executed in server `Server2`, and `rt_task1` and `rt_task4`

[1] libVLC `http://wiki.videolan.org/Libvlc`

in server `Server0` and `Server1` respectively (`rt_task2` and `rt_task3` was not included in this setup).

Given these 4 setups, task `rt_task5` will get different amount/distribution of CPU power and the processing of movie images (frames) will therefore also be affected. The movie processing is measured in amount of produced frames per second (FPS). The CPU utilization (percantage of CPU time) of task `rt_task5` is shown in Table IV as well as the frame rate of which `rt_task5` is processing a movie. We measured the FPS by timestamping the beginning and end of the movie playback system call and dividing the amount of frames of the movie with the measured time. The amount of frames is 91 and this value was generated by Mplayer[2] (using the `benchmark` flag). It is important to note that the CPU utilization given in Table IV is the $available$ CPU time, it does not mean that task `rt_task5` uses this CPU time. The FPS values may not considered to be 100% accurate, but it shows the approximate efficiency. For example, running `rt_task5` with 100% CPU should of course not give worse FPS value than running it with 32% CPU. These values are of course affected by overhead from the Linux kernel etc. We ran the the experiments on an Intel Pentium Dual-Core (E5300 2,6GHz) platform, equipped with a Linux kernel version 2.6.31.9, running with $load\ balancing$ disabled. The recorded tasks (and servers) ran on the same core, i.e., all tasks were migrated to CPU #0 at initialization phase.

Setup	CPU utilization (%)	FPS
Lowest prio	22.65	22.55
Medium prio	51.25	23.57
Highest prio	100	25.48
HSF	32	25.66

Table 8.4: FPS of task `rt_task5`.

The conclusion based on Table IV is that the distribution of CPU power influences the frame frequency a lot and that utilization alone is not sufficient for determining this. For example, giving task `rt_task5` 51.25% of the CPU produces less FPS than giving it 32%. The 32% CPU is guaranteed (no more no less) and it is distributed evenly as can be seen by the recording of HSF recorder in Figure 8.8 (visualized with the Grasp tool [13]).

[2]Mplayer `http://www.mplayerhq.hu/design7/news.html`

Apparently, (during our experiments) task `rt_task5` must have been active when other higher priority tasks were occupying the CPU, thereby temporarely getting less than 51.25% CPU. This is not the case when running the multimedia task in its server, since it is always supplied 32%.

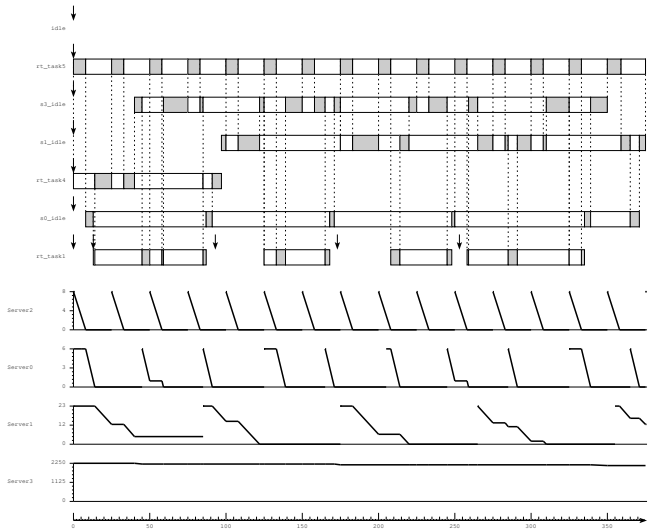

Figure 8.8: Tasks and servers recorded with the HSF recorder.

Figure 8.9 shows the same trace as in Figure 8.8, but recorded with the Ftrace recorder. As can be seen, the HSF recorder records correctly, also, it shows that task `rt_task5` does not consume CPU continuously (i.e., it blocks often).

Figure 8.10 shows a trace by our HSF recorder when task `rt_task5` was running with lowest priority, without HSF. As can be seen, the CPU availability for task `rt_task5` is highly dependant on when higher priority tasks execute.

Our example shows that it is difficult to fine tune the CPU supply for a multimedia task, i.e., we can only do it by changing the priority of the task since it is not periodic. However, it is possible to do tuning by setting server period, budget and priority, when using HSF. The main contribution of this example is the trace (Figure 8.8) made by the HSF recorder which shows the correctness of the CPU distribution, made by HSF, to real-time tasks (with media processing). We have also tested the correctness of the HSF recorder by

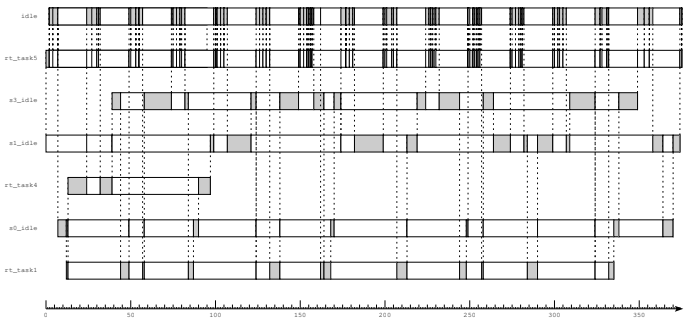

Figure 8.9: Tasks recorded with the Ftrace recorder.

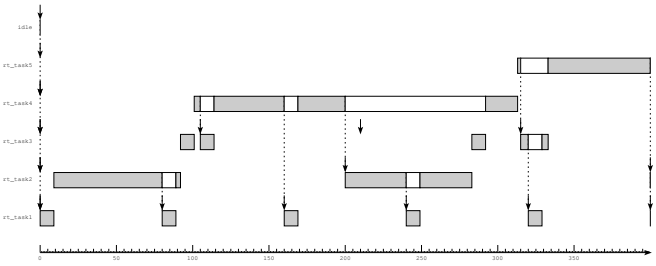

Figure 8.10: Tasks recorded with the HSF recorder.

comparing its trace results with the Ftrace recorder, i.e., the trace in Figure 8.8 is identical with the trace in Figure 8.9, which shows that it records correctly. Also, the trace in Figure 8.8 shows the amount of unused CPU time (slack time) at both server level and within each server, since the different idle tasks represent this. For example, server `Server3` (which has lowest priority) and its task `s3_idle` represent slack time at server level, while `s0_idle` represent unused time in `Server0`. The conclusion is that the HSF recorder can be a good tool for debugging hierarchical schedulers in RESCH, since it records accurately and with low overhead. Further, this example shows that our (HSF) recorder and scheduler records (and schedules) correctly, even though we do not modify the kernel.

8.5 Related work

The idea of our solution is based on the replay debugging approach [18], which records system events online and replays them offline. In later work [19], the replay debugging has been extended to be compiler- and OS-independent. While the replay debugging works with off-the-shelf compilers for application-level debugging, our solution is self-contained software using Grasp [13] for OS-level debugging, and it is primarily focused on real-time scheduler debugging.

The SCHED_DEADLINE project [20], which is in charge of the EDF scheduler implementation for Linux, has used the sched_switch tracer provided by the Ftrace toolkit [17] to output the recordings of context switches. The output logs are later converted to the Value Change Dump (VCD) format so that GtkWave can visualize the task execution traces. The trace can of course be converted to other trace formats, such as the Tracealyzer [12] or the Grasp [13] format. Given that Ftrace is supported by the Linux community, it is reasonable to use this toolkit to trace task executions for kernel debugging, but it is dedicated to the Linux kernel, so it is not necessarily suitable for real-time scheduler debugging in general. For instance, sched_switch does not catch job releases, however, context switches are precisely traced, and it can distinguish between task completions and task preemptions. Our solution is more flexible and integrated in that it is available not only for the Linux kernel, but also for other OSs, once the RESCH framework is ported to other platforms.

Our previous work [21] includes a simple task recorder in Linux (based on RESCH) which supports the Tracealyzer [12] and the Grasp [13] format. Further, we have also implemented a task recorder [14] (in Linux) which is able to record all task scheduling events, but it requires modifications to the kernel.

DTrace [22], SystemTrap [23], LTT [24], and LTTng [25] are advanced tools for OS debugging. They are oriented for tracing entire kernel events, so it is required that the developers understand how to use them. Meanwhile, our solution is more simplified by focusing on real-time scheduler debugging, and it is very easy to use in practice.

Real-Time Application Interface for Linux (RTAI) [26] is a collection of loadable kernel modules and a kernel patch which together provides a rich real-time API to the user. It gives the possibility to add/delete hooks for every task-start, task-switch and task-delete. These hooks give the possibility to monitor task execution in a detailed level.

Tracealyzer [12] is a visualization and analysis tool for embedded systems.

It can visualize task traces as well as task communication. Recorders implemented in the OSs VxWorks, OSE, Rubus and RTXC support the Tracealyzer format.

8.6 Conclusion

We have presented the implementation and evaluation of a task/server recorder based on the RESCH (REal-time SCHeduler) framework in Linux. RESCH is a scheduling framework for Linux which support scheduler plugins, i.e., multi-, uni-core, flat-, server-based-scheduling etc. Our recorder implementation is a plugin on top of an already existing hierarchical scheduler plugin called HSF (Hierarchical Scheduling Framework). This framework supports fixed-priority preemptive scheduling of servers as well as tasks. The HSF recorder uses scheduling primitives supported by RESCH itself, and HSF, in order to record scheduling events. The RESCH framework, the HSF scheduler plugin as well as our HSF recorder require no modification of the kernel and this is the main contribution of this approach. To the best of our knowledge, this is the first attempt to perform task tracing (within hierarchical scheduling) in Linux, without kernel modifications.

The evaluation of the HSF recorder includes two parts:

- Overhead comparison against an optimized version of our previously implemented task-switch patch [14].

- The correctness of the HSF recorder (as well as the HSF scheduler) is tested with a media processing example. The tracing capability and accuracy of the HSF recorder is compared against the main-line Linux recorder Ftrace [17].

Our HSF recorder produces very low overhead, in terms of CPU consumption, compared to the task-switch patch. The amount of recorded data is also much smaller than Ftrace, suggesting that the HSF recorder could be a better choice if only a subset of Linux tasks is of interest to monitor.

The media-processing example shows 5 real-time tasks running with, and without servers, i.e., with the HSF scheduler activated and with only RESCH. In the example, we show that one of the tasks (which is processing a movie) produces higher frame rate with theoretically lower CPU utilization (using the HSF scheduler) than with higher CPU utilization (using only RESCH). The

reason for this is that HSF gives the media-processing task better CPU resource distribution. In this example, the HSF recorder contributes by showing that the media task uses only its allocated CPU resource, thereby showing that the example is correct. It also shows a weakness with the HSF scheduler in that it has problems with keeping media tasks (and similar tasks which blocks often) within its server. However, non-blocking real-time tasks are shown to be properly contained inside their servers. All traces from the HSF recorder, in this example, are done in parallel with the Ftrace recorder, thereby showing the accuracy (and correctness) of our HSF recorder.

The conclusion is that the HSF recorder could be a good tool for debugging hierarchical schedulers in RESCH. The recorder can, together with a visualization tool, such as Grasp [13], visualize the execution of tasks and servers as well as display worst-case, best-case and average value of both execution- and response-time of tasks. In case that the Linux kernel is configured with Ftrace, then it could be useful to use also, since it complements our recorder well. Our recorder can record server events and task releases, while Ftrace can record the context switches between the RESCH real-time tasks and other Linux tasks.

Future work includes merging Ftrace and the HSF recorder to get more detailed and complete traces. We will also continue with improving the HSF scheduler plugin as well as developing new server-based schedulers (Bandwidth Sharing Server, Constant Bandwidth Server, Sporadic Server etc.) and support for multi-core scheduling (and tracing).

Bibliography

[1] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *2nd USENIX Symposium on OS Design and Implementation*, pages 107–121, Oct 1996.

[2] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *18th IEEE International Real-Time Systems Symposium*, pages 308–319, Dec 1997.

[3] John Regehr and John A. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In *22nd IEEE International Real-Time Systems Symposium*, pages 3–14, Dec 2001.

[4] Mikael Åsberg, Moris Behnam, Thomas Nolte, and Reinder J. Bril. Implementation of Overrun and Skipping in VxWorks. In *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 45–52, July 2010.

[5] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards Hierarchical Scheduling in VxWorks. In *4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 67–76, July 2008.

[6] Mikael Åsberg, Moris Behnam, Farhang Nemati, and Thomas Nolte. Towards Hierarchical Scheduling in AUTOSAR. In *14th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1181–1188, Sep 2009.

[7] Mikael Åsberg, Thomas Nolte, and Paul Pettersson. Prototyping and Code Synthesis of Hierarchically Scheduled Systems using TIMES. *Journal of Convergence*, 1(1):77–86, Dec 2010.

[8] Mikael Åsberg, Paul Pettersson, and Thomas Nolte. Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling. In *23rd IEEE Euromicro Conference on Real-Time Systems*, pages 172–181, July 2011.

[9] ARINC. *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), 1996.

[10] ARINC/RTCA-SC-182/EUROCAE-WG-48. Minimal Operational Performance Standard for Avionics Computer Resources. 1999.

[11] Mikael Åsberg, Shinpei Kato, Thomas Nolte, and Ragunathan Rajkumar. ExSched: An External CPU Scheduler Framework for Real-Time Systems. In *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2012.

[12] Anders Wall, Johan Andersson, and Christer Norström. Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings. In *1st International Symposium on Leveraging Applications of Formal Methods*, pages 130–145, Oct 2004.

[13] Mike Holenderski, Martijn Heuvel, Reinder Bril, and Johan Lukkien. Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 37–42, July 2010.

[14] Mikael Åsberg, Thomas Nolte, Clara M. Otero Perez, and Shinpei Kato. Execution Time Monitoring in Linux. In *Work-In-Progress session of 14th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1601–1604, Sep 2009.

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

[16] Shinpei Kato, Raj Rajkumar, and Yasuyuki Ishikawa. AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms. In *22nd IEEE Euromicro Conference on Real-Time Systems*, pages 47–56, July 2010.

[17] Tim Bird. Measuring Function Duration with Ftrace. In *1st Annual Japan Linux Symposium*, Oct 2009.

[18] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real Time Systems. In *12th IEEE Euromicro Conference on Real-Time Systems*, pages 265–272, June 2000.

[19] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *17th International Symposium on Parallel and Distributed Processing*, page 288, April 2003.

[20] D. Faggioli, M. Trimarchi, and F. Checconi. An implementation of the Earliest Deadline First algorithm in Linux. In *24th Annual ACM Symposium on Applied Computing*, pages 1984–1989, March 2009.

[21] Mikael Åsberg, Johan Kraft, Thomas Nolte, and Shinpei Kato. A Loadable Task Execution Recorder for Linux. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 31–36, July 2010.

[22] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal. Dynamic Instrumentation of Production Systems. In *13th USENIX Annual Technical Conference*, pages 15–28, June 2004.

[23] V. Prasad, W. Colhen, F. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating System Problems Using Dynamic Instrumentation. In *7th Ottawa Linux Symposium*, pages 49–64, July 2005.

[24] K. Yaghmour and M.R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *9th USENIX Annual Technical Conference*, pages 13–26, June 2000.

[25] M. Desnoyers and M.R. Dagenais. The LTTng Tracer: A Low Impact Performance and Behavior Monitor of GNU/Linux. In *8th Ottawa Linux Symposium*, pages 209–224, July 2006.

[26] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 29(10), April 2000.

Chapter 9

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

Mikael Åsberg, Paul Pettersson and Thomas Nolte
In ECRTS'11 Conference, pages 172–181, July, 2011

Abstract

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.

9.1 Introduction

Hierarchical scheduling [1, 2, 3] has been introduced as a means to simplify parallel development of embedded systems. It facilitates the integration of such systems by providing mechanisms for temporal isolation between software parts, called subsystems. The schedulable entity manifested by a subsystem is referred to as a *Server*. A system (a product, a large piece of software etc.) can be composed of a number of subsystems, where each of these typically implement a particular function or feature of the whole system. For example, a car has a number of features/subsystems, and two examples of these are the engine control system and the anti-lock braking system. These features/subsystems should ideally be developed in parallel and integrated smoothly [4]. Integration related problems include having to cope with different scheduling policies among subsystems, sharing the CPU resource among subsystems according to their need (and keeping that share during runtime), and ensuring that timing faults do not propagate from one subsystem to another. An example of such a fault is a piece of software that requires more time to execute than originally intended (exceeding its analysed worst-case execution time), and thereby causing unforseen interference with the rest of the system. Yet another integration problem is the introduction of new software functions, not apparent at early design.

Hierarchical scheduling allows for timing analysis of an entire system, as well as for subsystems in isolation, before they are integrated. It supports multiple scheduling policies and it has a runtime mechanism that multiplexes the CPU resource among subsystems, hence, making sure that no unpredictable interference between subsystems will occur in the time domain. Also, the size of the CPU share can easily be re-configured, allowing for ”last minute” changes when introducing new software late in the development process.

One important property of hierarchical scheduling, when it comes to hard real-time applications, is the safe execution environment for a subsystem. The scheduling entity of a subsystem, i.e., a server, should ensure (together with the scheduler) that the subsystem will get the exact CPU share that it was promised. Even though a subsystem is executed together with other (potentially faulty) subsystems, it should still get the CPU share that it is entitled to. In practice, hierarchical scheduling can prevent faulty subsystems from propagating timing faults to other subsystems. However, hierarchical scheduling cannot deal with timing faults propagating from itself, i.e., a faulty scheduler causing incorrect scheduling events, and thereby violating the contracted CPU shares that belong to the subsystems. This is of course not acceptable in applications with hard

real-time constraints.

We have experience in the implementation of two-level hierarchical scheduling frameworks in operating systems such as VxWorks [5] and Linux [6]. Our implemented frameworks operate in two levels using periodic/polling servers (PS) [7] and, inside these, fixed priority preemptive scheduling (FPPS) of periodic tasks. Even though the setup of these frameworks are quite simple (two-level, PS and FPPS), it gives rise to a large implementation complexity, since we are dealing with multiple schedulers (multiple scheduling-related timing events). From our experience, debugging/tracing of this kind of scheduling [6] is very time consuming. Also, debugging/tracing does not guarantee 100% correctness, since it can be difficult to determine wheather the schedule is correct or not. Due to this, in this paper we look at modelling, formal verification and code-synthesis of hierarchical scheduling with FPPS.

The motivation for modelling hierarchical FPPS is inherent in its wide support for schedulability analysis [8, 9, 10], as well as the evolving research in synchronisation protocols [11, 12], which need hierarchical scheduling implementations/models for its development and evaluation.

Recently, automata based approaches have been proposed to describe/analyse a broad set of real-time scheduling policies. One of the advantages of these approaches is the ability to generate generic task release patterns. In task automata models [13], task release patterns are modelled using timed automata [14]. It has been shown that the schedulability analysis problem is resolvable for both FPPS and dynamic scheduling policies such as earliest deadline first (EDF). Other benefits of such approaches are that simulation, formal verification of timing/functional safety properties, as well as code-synthesis [15] is possible. The Times tool [16] supports modelling with the task automata model, and it can perform simulation, verification, code-synthesis etc. However, hierarchically scheduled systems cannot be verified using existing solutions.

In this paper our overall goal is to model, verify and synthesise a two-level hierarchical scheduling framework. The main contributions of this paper are:

1. We have modelled two-level hierarchical scheduling, with FPPS and PS at the global level with support for an arbitrary number of servers with FPPS and periodic tasks at the local level. We have used the modelling language task automata and implemented the model using the Times tool. To the best of our knowledge, this is the first task-automata model of two-tier FPPS with PS.

2. We have extended the model with support for verification (using what

we call $observers$), allowing us to verify that the model matches the scheduler behavior (properties) that we have specified. Note that we are NOT verifying schedulability analysis, but the scheduler itself (two-level FPPS with periodic tasks/servers). The contribution to the state-of-the-art is the verification of the schedulers (scheduling policies) in a hierarchically scheduled system.

3. We have used the built-in code generator in Times to synthesise our model. However, the manual work needed includes adapting the code for our large model (which has 370 edges and 155 locations), since the Times code-generator currently supports a limited size. This work also includes removing platform (Linux simulator) dependent code, and inserting VxWorks related code. This gives us the possibility to get real overhead estimates of the modelled scheduler when executing it. The results presented are the actual execution traces of the scheduler executed in the VxWorks kernel, as well as a comparison of CPU- and memory-overhead against an equivalent manually-coded hierarchical scheduler. To the best of our knowledge, there is no prior work on synthesis (from model) for this type of scheduling.

The outline of this paper is as follows: in Section 9.2 we outline preliminaries on hierarchical scheduling, task automata and Times. In Section 9.3 we present the model of two-level hierarchical scheduling, in Section 9.4 we show how we have verified the behavior of the modelled scheduler, and finally in Section 9.5, we show the result of the synthesis. Section 9.6 presents related work, and finally, Section 9.7 concludes.

9.2 Preliminaries

9.2.1 Hierarchical scheduling

Hierarchical scheduling has been introduced to support CPU multiplexing in combination with different scheduling policies. It can generally be represented as a tree of nodes with arbitrary size, where each node represents a subsystem with its own local scheduler for scheduling internal workloads (tasks). Looking at the tree-structure representation, the CPU resource is allocated from a parent node to its children nodes. One of the main advantages of hierarchical scheduling is that it provides means for decomposing a complex system into well-defined parts (subsystems). In essence, hierarchical scheduling gives rise to time-predictable *composition* of coarse-grained subsystems.

This means that subsystems can be developed and tested independently, and at a later stage assembled without introducing unwanted temporal behavior. Hierarchical scheduling also facilitates *reusability* of subsystems, since their computational requirements are characterised by well defined *interfaces*.

Figure 9.1 illustrates two-level hierarchical scheduling. The left side illustrates the structure: the top node is defined as the *Global scheduler* and it is responsible for distributing the *CPU* capacity to the servers (the schedulable entity of a subsystem). Servers are allocated a defined time (*budget*) every predefined *period* [17] and they are executed based on their *priority*. They are scheduled according to the scheduling policy of the global scheduler (for example FPPS or EDF) and the parameters just mentioned, hence, they can be viewed as ”virtual tasks”. Each server can comprise a *Local scheduler* which schedules the workload inside it, i.e. its tasks, when its server is selected for execution by the global scheduler. Note that the local scheduling policy may differ from the global policy. The interfaces (T,C,Pr) for tasks and servers shows the allocated CPU capacity. It includes the release period, execution time (or budget in the case for a server) and priority (lower value corresponds to higher priority). The right side of the figure corresponds to the runtime behavior of the structure.

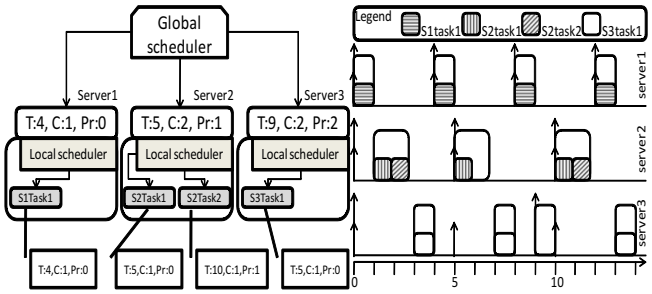

Figure 9.1: Example hierarchical FPPS.

9.2.2 Task automata and TIMES

Timed automata [14] is a widely used modelling language for formal modelling and analysis of real-time systems. A timed automaton is essentially a finite state automaton extended with real-valued clocks that can be tested and reset. The formalism has shown to be suitable for a wide range of real-time systems.

The timed automata model has been extended with an explicit notion of tasks, with parameters such as periods, priorities, execution times etc. The model, referred to as *task automata* (of *timed automata with tasks*), associates asynchronous tasks with the locations (states) of a timed automaton, and assumes that the tasks are executed using static/dynamic priorities with a preemptive or non-preemptive scheduling policy. This model is supported by the Times tool. One of the main benefits of using this tool (in the context of this paper) is that it supports task automata, which is suitable for modelling schedulers. Secondly, it can verify properties of a modelled system. Last but not least, the tool has a code-generator which gives the possibility for synthesis.

In case that tasks are released periodically (with or without offsets), or aperiodically, the input to the Times tool is merely a task table in which the following parameters are defined for a task: name, execution time, (relative) deadline, priority (in case of static priority scheduling), offset and period (if applicable), interface, semaphore usage, and its C-code. Alternatively, a task can be of type *controlled*, meaning that the release pattern of a task is defined by a given task automata. All tasks in our modelled hierarchical scheduler are of type controlled.

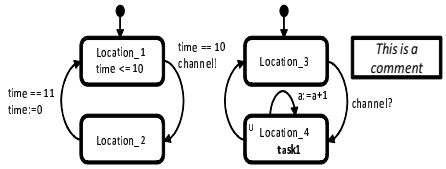

Figure 9.2: Example task automata.

Figure 9.2 shows an example of a task automata that releases a (*controlled*) task for execution, at minimum, every 10 time units. The arrows (with a dot) to state **Location_1** and **Location_3** defines that they are the start locations. The invariant **time <= 10** defines that control can only be at this state up until time 10, then a transition has to be made. The condition **time==10** defines that a transition may take place if this holds. The channel **channel!** defines that when this transition is made, the corresponding channel **channel?** must be activated, i.e., there has to be a transition between state **Location_3** and **Location_4**. The latter location has a task release statement (**task1**), and this means that upon arrival at this state, task *task1* is released for execution. State **Location_4** is flagged as *urgent* (**U**), which defines that no time will pass when computing **a:=a+1** or before the transition to state **Location_3**. A transition

from state **Location_2** to **Location_1** may take place when **time==11**, if so, the clock **time** will be reset to zero.

9.3 Model

This section will describe the hierarchical scheduler, modelled in Times. The modelling language of task automata is used for modelling the framework. This language allows task releasing, and transitions/actions can be controlled with clock constraints (as shown in Figure 9.2). However in general, in order to implement hierarchical scheduling, one either need to be able to release tasks and suspend them, **or**, release tasks and change task priorities dynamically during runtime (in order to perform a server context switch). Unfortunately, task suspension and dynamic priority (of controlled tasks) is not supported by the Times tool. In order to solve this issue, we model an executing task as a series of task releases, where each task release will execute the task 1 time unit. Hence, the minimum task execution time is 1 time unit, and the execution time is discrete, i.e., it has to be divisible by 1 (without generating a remainder). What this means in practice, is that when there is a task executing within a server and its budget depletes, then we simply stop releasing the task (and take a note of the amount of time executed so far). This is illustrated in Figure 9.3 where a task is supposed to execute 5 time units, within 2 budget instances of its server. This results in 3 task releases at the first server instance and 2 releases in the second instance. This fragmentation does not affect the task model, schedulability analysis or verification, it just makes the task automata model more complicated to implement. A more practical approach is to only model task releases and no actual task execution (hence there will be no task suspension in the model). The downside of such a non-fragmented approach is restricted verification capabilities as well as no possibilities of graphical representation during simulation (Figure 9.8). We will show verification using the fragmented task model, and we will show code-synthesis for both the fragmented and the non-fragmented model (Section 9.5).

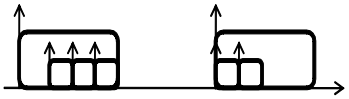

Figure 9.3: Discrete task execution.

The model structure is illustrated in Figure 9.4. The global scheduler ac-

tivates the servers with channels, through the *EventHandler* automata. The global scheduler is unchanged when adding/deleting servers, only the *EventHandler* is affected. Servers are activated periodically and they run according to their budget and priority, i.e., PS with FPPS. In our model, Server 3 has a local scheduler, scheduling periodic tasks with FPPS. Server 1 has no scheduler, i.e., it just releases a task upon activation and lets it run until budget depletion.

Each scheduler (global or local) has a ready- and a release-queue. The ready-queue contains the servers/tasks, ordered by priority. The release-queue stores the release times (in absolute time) of the servers/tasks, ordered with the earliest time first. The queues are implemented as arrays and insertion is based on a binary search algorithm.

As mentioned previously, a server is activated/deactivated through channels (where the global scheduler is the initiator). This means that a server must always be prepared to be activated/deactivated, i.e., all of its states which are not marked as *urgent* must have an activation/deactivation channel. If this is fulfilled, then the server will be in total control by the global scheduler, hence, scheduling errors will not propagate from local to global level. Also, if the global scheduler is verified, then the local scheduler can assume that it is getting its correct timeslots (according to its interface), making verification at the local level easier (the power of compositional verification). The local scheduler releases its tasks according to the model illustrated in Figure 9.3, which will prevent the tasks from executing outside of its servers budget (the Times simulation in Figure 9.8 illustrates this).

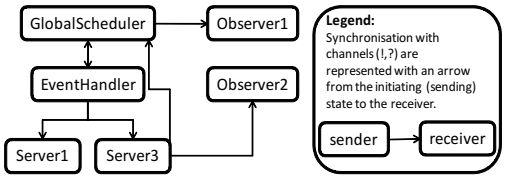

Figure 9.4: Structure of the model.

Observer1 and *Observer2* (Figure 9.4) will get notifications of scheduling events through channels. We define scheduling events as being task/server releases, server budget depletion and task suspension (due to the task finishing its current execution). The observers themselves do not initiate these synchronisations and they do not affect the clocks, hence, they do not affect the behavior of the model. The observers are mainly used for the purpose of verifying the schedulers [18], this will be elaborated in more detail in Section 9.4.

9.3.1 Global scheduler

Figure 9.5 illustrates a simplified version of the global scheduler. The excluded parts include initialisation, queue management etc. Basically, whenever there are no scheduling events, the automata waits in the main state, i.e., the one without the *urgent* symbol (**U**). This is the only state where time is allowed to pass. From the main state, there are in total three transitions possible: server budget deplete, server release and allowing for a task-event (i.e., task release etc.) that belongs to the current active server. As can be seen, the depletion transition has highest priority, followed by the release and task-event transitions. The latter is necessary since the global scheduler needs precedence over local scheduling events when they occur at the same time. As with the priority of the other two, it is simply more convenient to handle a budget-deplete event before a release event (when they occur at the same time).
As can be seen by the model, we model that scheduling events do not consume any time (hence the *urgent* symbols). The reason for this is to reduce the complexity of the model. This means that during simulation, the scheduler produces no overhead. However, running experiments would of course yield some scheduler overhead, these details will be shown in Section 9.5.

$Observer1$ is notified about server budget-deplete (**DepleteObs1!**) and server release events (**ReleaseObs1!**), this is shown in Figure 9.5.

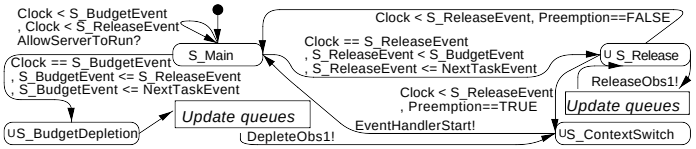

Figure 9.5: Model of the global scheduler (simplified).

9.3.2 Event handler

Figure 9.6 shows the model of the event handler. The motivation for its existence is that it abstracts the number of servers from the global scheduler, i.e., adding/removing servers only affects the number of states in the event handler and not in the global scheduler. Since *channels* cannot be declared as arrays, every server requires 2 states (activation and deactivation) in this model. As can be seen in this model, the global scheduler observer (*Observer1*) is notified if there is a server scheduling event, and which servers that are activated/deactivated.

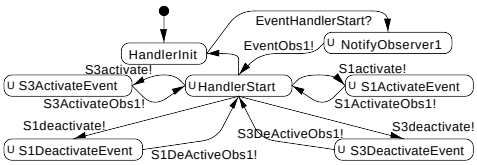

Figure 9.6: Model of the event handler (simplified).

9.3.3 Local scheduler

The local scheduler model (Figure 9.7) is similar to the global scheduler. Discretising the time is important for keeping track of events, hence the added time pass state that increments time (clocks are not allowed be read in timed automata). The time-pass state is crucial since the local scheduler has more scheduling events to keep track of, compared to the global scheduler.
Whenever the server is deactivated, it stays in the sleep state. In active mode, the server can release, stop and increment a tasks execution. The latter goes back to the statement that a tasks execution is discrete with sections of 1 time unit of execution.
Observer2 is notified of events by getting triggered by the local scheduler through a number of channels.
Each upcoming task scheduling-event must be passed to the global scheduler so that it does not schedule a server event (such as deactivating the server) without letting the local scheduler handle task scheduling events that are earlier in time. The upcoming task scheduling event is calculated in the **CalcNextEvent** state and stored in the **NextTaskEvent** variable, which is visible in the global scheduler.

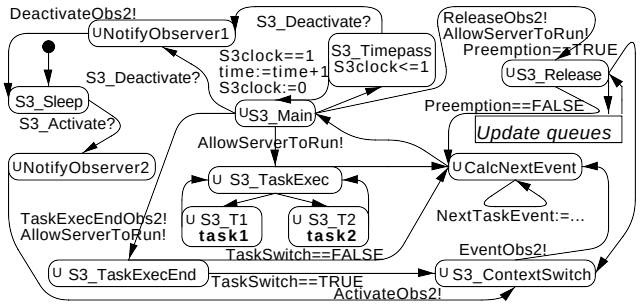

Figure 9.7: Model of the local scheduler (simplified).

All models (including schedulers, observers etc.) can be viewed in our technical report [19].

9.4 Verification

We have specified 5 respectively 4 properties for each scheduler level (global/local) that should be satisfied by our modelled schedulers. We use two so called $observer$ automata that will implement the behavior (properties) that we have specified. The next step is to use the built in verifier in Times, and simply construct logic statements (TCTL) that checks if certain states are reached in the observers. The observers will reach these states if they detect a scheduling fault that contradicts our proposed properties. $Observer1$ is used to verify the global scheduler, and $Observer2$ is used for the verification of the local scheduler. The reason for using observers, instead of only using logic statements in Times, is that the verifier cannot determine the amount of time elapsed from one location to another, which we need in order to conduct our verification. Naturally, all automata have been checked for the absence of deadlock before proceeding with the verification.

9.4.1 Task/server systems used in the verification

It is well known that model checking requires a finite model, and thus, it might cause problems when verifying schedulers [20, 21] since the tasks give rise to unknown factors such as number of tasks, task parameters etc. In essence, different task sets will give rise to different automata transitions (behavior), so the scheduler will behave different depending on task sets. Due to this, we explore the fact that the modelled scheduler has a small set of scheduling events (task/server release, task/server suspension, context switch etc.), even when including the combinations of these events (as we will see). We identify all of these events, which represents the entire behavior of the scheduler. Then we run the scheduler together with selected task/server sets that will generate all of these (combinations of) scheduling events, during the verification. Alternatively (just to be safe), since the process from modelling/verification down to synthesis is short, once the model is finished (the verification of models in this size takes just a few minutes on a standard PC), a system can be verified with scheduler and load (task/server) together before deployment.

We ran three different task/server systems (system 1, 2 and 3) during the verification of our scheduler properties. The three systems are presented in

Name	T	$Budget$	D	$Prio$	**Tasks**
Server1	19	2	19	Low	{server1}
Server3	5	3	5	High	{s3task1,s3task2}

Table 9.1: Server set (used in system 1 and 2).

Name	T	$Budget$	D	$Prio$	**Tasks**
Server1	19	2	19	Low	{server1}
Server3	10	6	10	High	{s3task1,s3task2}

Table 9.2: Server set (used in system 3).

Table 9.3, 9.4 and 9.5. The corresponding execution traces can be found in Figure 9.9, 9.10 and 9.11. The server parameters used for systems 1 and 2 (Figure 9.9 and 9.10) are listed Table 9.1, and the server parameters for system 3 (Figure 9.11) is shown in Table 9.2. Figure 9.8 shows a simulation trace (in Times) of system 1, i.e., Figure 9.9.

Name	T	C	D	$Prio$
server1	-	-	-	-
s3task1	10	3	10	Low
s3task2	11	1	11	High

Table 9.3: Task set of system 1.

Table 9.6 list all possible scheduling events at the global level. A release or suspension of a task/server can lead to a context switch (c.s.). If not (in case of suspension), then there will be a switch to an idle task/server, which is not part of our model, hence we define a context switch only when the model switches between tasks/servers that are defined in the model. A simultaneous suspension/release will always lead to a context switch. We do not differentiate if the task/server that is released is to be switched in, or, if there is another higher priority task/server ready to be switched in. We differentiate in that local scheduling events can occur when its server is active, the time when its server activates and the time when its server deactivates. Local scheduling events happen only during the time when its server is active (according to the model). Related to the undefined events in Table 9.7, a task suspension cannot happen during a server release since it cannot finish its execution at the same time as its server activates. The local scheduler does not differentiate the cause

Name	T	C	D	$Prio$
server1	-	-	-	-
s3task1	16	4	16	Low
s3task2	11	2	11	High

Table 9.4: Task set of system 2.

Name	T	C	D	$Prio$
server1	-	-	-	-
s3task1	10	3	10	High
s3task2	11	1	11	Low

Table 9.5: Task set of system 3.

of its servers activation/deactivation, e.g., there is no differentiation if the server activation is due to a release, or suspension of a higher priority server. Hence, we do not need to consider all possible cases/combinations of local and global scheduling events. All scheduling events in Table 9.6 and 9.7 are referred to the execution traces presented in systems 1, 2 and 3. These scheduling events will occur during the verification of the global (section 9.4.2) and local scheduler (section 9.4.3).

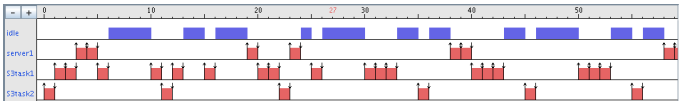

Figure 9.8: TIMES simulator (simulating system 1).

9.4.2 Global level verification

In the verification of the global scheduler, we use the server parameters shown in Table 9.1, which will generate all server scheduling events (shown in Table 9.6). The following properties are defined (and later verified):

$Property1$: A server S_i (with index i) should never get more than C_i budget at any discrete interval (non sliding) of length P_i, where the first interval starts at time 0.

$Property2$: A server S_i (with index i) should never get less than C_i budget at any discrete interval (non sliding) of length P_i, where the first interval starts

Server event	Example
Release (c.s.)	Fig. 9.9, time=20
Release (no c.s.)	Fig. 9.9, time=57
Suspend (c.s.)	Fig. 9.9, time=03
Suspend (no c.s.)	Fig. 9.9, time=08
Suspend/Release (c.s.)	Fig. 9.9, time=38

Table 9.6: Server scheduling events.

Task event	**Server event**		
	Active	Activate	Deactivate
Release (c.s.)	Fig. 9.9, t=11	Fig. 9.10, t=55	Fig. 9.10, t=33
Release (no c.s.)	Fig. 9.11, t=22	Fig. 9.10, t=00	Fig. 9.11, t=66
Suspend (c.s.)	Fig. 9.11, t=13	-	Fig. 9.9, t=23
Suspend (no c.s.)	Fig. 9.9, t=06	-	Fig. 9.10, t=08
Suspend/Release (c.s.)	Fig. 9.11, t=33	-	Fig. 9.9, t=33

Table 9.7: Task scheduling events.

at time 0, if there is unused time within this interval.

$Property3$: A server S_i (with index i) should always be released (inserted in the server ready-queue) according to its specified period P_i.

$Property4$: A server should always be removed from the server ready-queue upon server budget depletion.

$Property5$: The highest priority server in the server ready-queue should always be the current running server in the system.

We have modelled a task automata called $Observer1$ (Figure 9.13 and 9.14) that will check that each of the 5 properties are fulfilled.

Figure 9.12 shows at which server scheduling events the observer executes, the following list explains each event:

- Event **A** represents a release event.
- Event **B** represents the start/stop of a budget (not necessarily the beginning and end of a budget).
- Event **C** represents the end of a budget.
- Event **D** represents the the beginning of a budget in case it was idle previously.

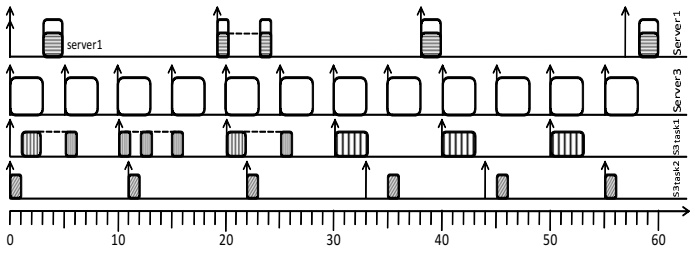

Figure 9.9: System 1.

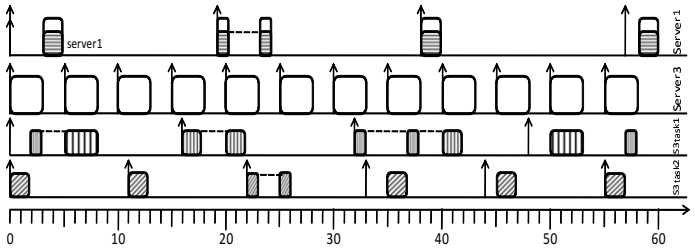

Figure 9.10: System 2.

$Property1$ and $Property2$ are checked by the observer by measuring the server budget, event **B** (Figure 9.12) illustrates these events. $Property2$ is not valid if there is no unused budget within the period, since that indicates a schedulability problem. At event **D**, a server is activated, and the observer timestamps this point if no previous server was running. This timestamp value is checked at event **A** together with the measured budget. If the timestamp is within the period, then there was unused time. At each event **A**, $Property1$ and $Property2$ are checked. In Figure 9.14, either a transition to state **LessBudget** or **MoreBudget** is made if the budget has been underused or exceeded. Event **D** corresponds to **CheckSlack** (Figure 9.14). The logical expressions (1) and (2) in Figure 9.15 checks that there is no path leading to the error states, i.e., for all paths ($\forall$), on every state along the path ($\Box$), a state is never ($\neg$) visited. The transition to these error states contradicts the requirements of $Property1$ and $Property2$. For more details on the modelling of the error states, we direct the reader to our technical report [19].

$Property3$ is checked at event **A** (Figure 9.12). State **IncorrectRelease2** (Figure 9.14) is active if the global scheduler tries to release a server at an

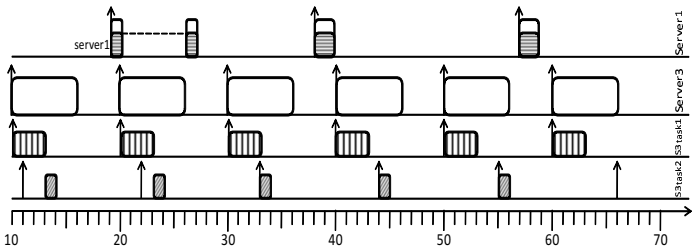

Figure 9.11: System 3.

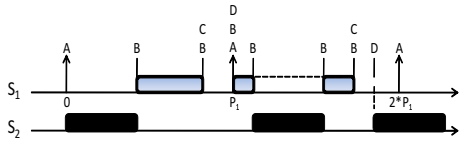

Figure 9.12: $Observer1$ events.

incorrect time. A transition to state **IncorrectRelease1** (Figure 9.14) is done if there should be an incorrect value in the server release queue, which does not match the calculated release time of the observer. We have used the logical expressions (3) and (4) in Figure 9.15 to check $Property3$ in Times.

$Property4$ is checked at event **C**, Figure 9.12. Whenever there is a server deplete event, the observer checks that the server is no longer in the server ready-queue (**IncorrectDeplete**, Figure 9.13). The logical expression (5) (Figure 9.15) verifies this property.

$Property5$ is checked at event **A** by checking the server ready-queue content and order, the logical expression used is (6) (Figure 9.15). We check that a server is in the ready queue after its release (**CheckQueueContent**) and that the queue is ordered correctly (**CheckQueueOrder**), both states are found in Figure 9.14. **Server_Event** is entered whenever there is a server context switch (Figure 9.13). It is not possible to enter this automata part if no budget depletion or server release has occurred (**CheckExecution**, Figure 9.13), this refers to expression (7) (Figure 9.15). Yet two more expressions are important to check, (8) and (9) (Figure 9.15), in order to verify $Property5$. Whenever there is a server release that affects the server ready-queue in such a way that it ends up as the head node (**ServerCS2**, Figure 9.14), then it implies that a server context switch should occur (**Server_Event**, Figure 9.13). This is checked in

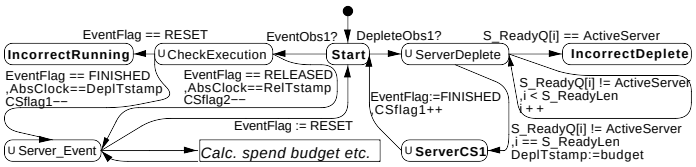

Figure 9.13: $Observer1$: Server context-switch and depletion.

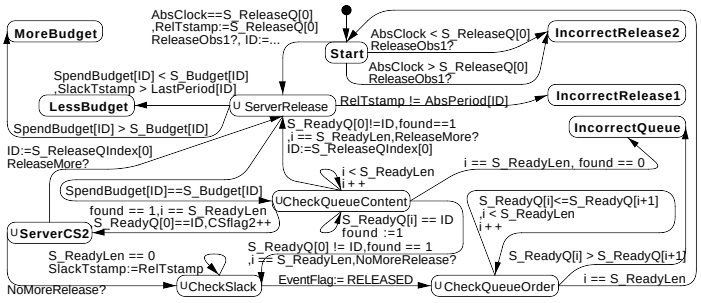

Figure 9.14: $Observer1$: Server release.

expression (8) (Figure 9.15), for all paths and states ($\forall\ \Box$), whenever state **ServerCS2** is reached, it implies ($\Longrightarrow$) that at some state in all the upcoming paths ($\forall\ \Diamond$), state **Server_Event** is reached and ($\wedge$), at the same time the condition $CSflag2 = 0$ holds. The condition is that it should happen directly, i.e., no time should pass. This is a condition in the model where the transitions between **CheckExecution** and **Server_Event** checks the elapsed time (Figure 9.13). Also, there should not be any nesting, i.e., two server releases (where both imply server context switch) followed by one context switch (hence the check $CSflag2 = 0$ in the expression). The same check is made for budget depletion, expression (9) (Figure 9.15).

9.4.3 Local level verification

During the verification of the local level we use all three task systems presented in section 9.4.1, and we use another observer called $Observer2$ (due to space restrictions we direct the reader to the technical report [19] for this figure) to verify the following 4 properties.

$Property6$: A task t_i (with index i) should always be released (inserted in

$$\forall \Box \neg LessBudget \quad (1)$$
$$\forall \Box \neg MoreBudget \quad (2)$$
$$\forall \Box \neg IncorrectRelease1 \quad (3)$$
$$\forall \Box \neg IncorrectRelease2 \quad (4)$$
$$\forall \Box \neg IncorrectDeplete \quad (5)$$
$$\forall \Box \neg IncorrectQueue \quad (6)$$
$$\forall \Box \neg IncorrectRunning \quad (7)$$
$$\forall \Box (ServerCS2 \implies (\forall \Diamond Server\_Event \wedge CSflag2 = 0)) \quad (8)$$
$$\forall \Box (ServerCS1 \implies (\forall \Diamond Server\_Event \wedge CSflag1 = 0)) \quad (9)$$

Figure 9.15: TCTL expressions.

the task ready-queue) according to its specified period P_i, OR if later, directly when its server is activated.

$Property7$: A task should always be removed from the task ready-queue upon finishing its execution.

$Property8$: The highest priority task in the task ready-queue (in each server) should always be the current running task in the server, when it is active.

$Property9$: All tasks should run within their respective server.

The only properties that are different in the local level compared to the global level are $Property6$ and $Property9$, we will explain these two briefly.

$Property6$ is checked with the same expressions as in the global level, but the local level observer will also allow task releases that coincide with its server releases.

Regarding $Property9$, $Observer2$ assumes that all context switches that happen during its observation are within the server under observation. Hence, it is only required to check that no task context switch (where the next running task belongs to the observed server) will occur during server deactivation. The property is checked by timestamping all task context switches. A transition is made to an error state if a task context switch occur at the same time as a server deactivation.

9.5 Code synthesis

We have synthesised the model into two different kernel-level implementations; the original model which has fragmented task executions and that is fully verified (Section 9.4), and the more simple model (without fragmentation) where only the global level is fully verified, and the local level is partially verified (only $Property6$ is fulfilled). In the simple model we don't use any internal task ready-queue (tasks are just released according to the release-queue), hence, the local level cannot be fully verified. We synthesised these two models for the sake of comparing the CPU overhead, further, we also included our previously manually coded hierarchical scheduler HSF [5] (as a reference point) in the comparison. The fragmented model is of course not practical, in terms of synthesis (real applications cannot have this kind of fragmentation), but still we show that it is possible to synthesise a fully verified hierarchical scheduler. Removing the fragmentation (and keeping the full verification) is just a matter of adding dynamic priority support (or the ability to suspend tasks) in the Times tool.

We measured the CPU overhead of all 3 schedulers as well as the memory consumption. The platform used for the experiments is VxWorks 6.6, running on an Intel Pentium4 (1,66 GHz, uni-core) desktop machine. The CPU overhead was measured with the sysTimestamp facility and the dynamic memory consumption was analysed with the Wind River Workbench Memory Analyzer. The tasks used in the experiments were executing empty for-loops and the execution times were estimated using the VxWorks Timex facility. During the experiments, the tick resolution was set to 1000 Hz. We let 1 time unit in the system represent 1 scheduler tick.

Scheduler	CPU (%)
Times (fragmented)	1.78
Times (non-fragmented)	1.36
HSF	0.08

Table 9.8: CPU overhead.

Table 9.8 shows the measured CPU overhead of the schedulers. The measurements were done in the first 2090 scheduler ticks, i.e., the least common multiple (LCM) of all task and server periods of system 1. The CPU overhead (%) represents the LCM of all task and server periods divided by the measured execution time of each scheduler. As can be observed, the non-fragmented version has less overhead than the fragmented, which is due to less

Scheduler	Dynamic memory		Static memory
	Max	Average	
Times (fragmented)	1646	1646	10874
Times (non-fragmented)	1646	1646	10874
HSF	11456	1692	24

Table 9.9: Memory overhead (bytes).

automaton transitions and task releases. Both generated schedulers has substantial more overhead than the manually coded scheduler, i.e., 17 respectively 22 times more CPU overhead. We experimented on the generated code with an optimisation which reduced the amount of scheduler invocations by 50% (1045 instead of 2090 scheduler invocations), however, the total CPU overhead was reduced by only 5%. We have identified more ways to optimise the code, but we defer this to future work.

Table 9.9 shows the amount of dynamic/static memory used by the schedulers. During the actual scheduling (after initialisation), the memory allocation of HSF drops down to 1692 bytes. The total memory used (during the scheduling) by HSF is 1716 bytes, and for the generated schedulers it counts up to 12520 bytes in total. The conclusion is that there is a similar worst-case memory usage (11480 vs. 12520 bytes), but less CPU overhead by HSF (0.08% vs. 1.36%).

Figure 9.16, 9.17 and 9.18 shows the actual runtime execution recording of the tasks and servers. As can be seen, our generated schedulers (Figure 9.17 and 9.18) gives the same trace as the manually coded scheduler (Figure 9.16). What can also be noted is how the fragmented model gives a slightly different execution trace than the non-fragmented since there are more task releases due to the fact that the task execution is divided into several one-time-unit sections of execution.

Our execution recorder uses the VxWorks taskHookLib and we use the visualisation tool Grasp [22] to display the recordings.

9.6 Related work

Hierarchical scheduling theory There is a growing attention in that little prior work has been done on verification of hierarchical scheduling implementations [23], as compared to the great amount of work on schedulability analysis [8, 9, 10, 24, 25, 26, 27, 28] (where there is an assumption that the

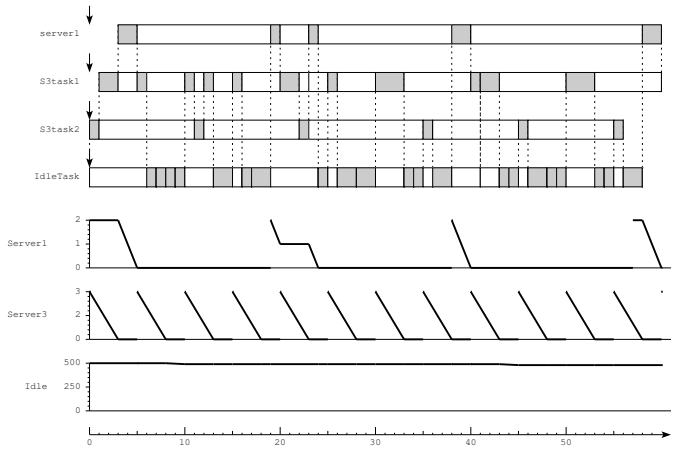

Figure 9.16: Execution trace of HSF, running system1.

scheduling policy is correctly implemented), which has originated from open systems [2] in the late 1990's.

Hierarchical scheduling implementation Among the implementation work, Kim *et al.* [29] propose the SPIRIT uKernel that is based on a two-level FPPS hierarchical scheduling framework, simplifying integration of real-time applications. A mix of theory and practice is presented in [3] where the authors reason about general scheduling trees with arbitrary scheduling policies and scheduling depths. They also present an implementation in Windows 2000. More recently, [5] and [30] implemented a two-level FPPS HSF in the commercial real-time operating systems VxWorks and μC/OS-II.

Scheduler modelling There are two main categories of scheduler modelling, either the scheduler already exists as an implementation and it is modelled (and verified) after code analysis or other techniques [31, 32, 33, 34, 35, 36], or (as in our paper) the scheduler is modelled and later verified (and perhaps also synthesised) [37, 38, 39, 40, 41].

In the area of modelling hierarchical scheduling, the authors in [42] show how modelling and schedulability analysis of two-level hierarchical scheduling, with timed automata, can be accomplished in the simulation tool Cheddar. Ha *et al.* [43] describes the verification, using theorem-proving, to verify the

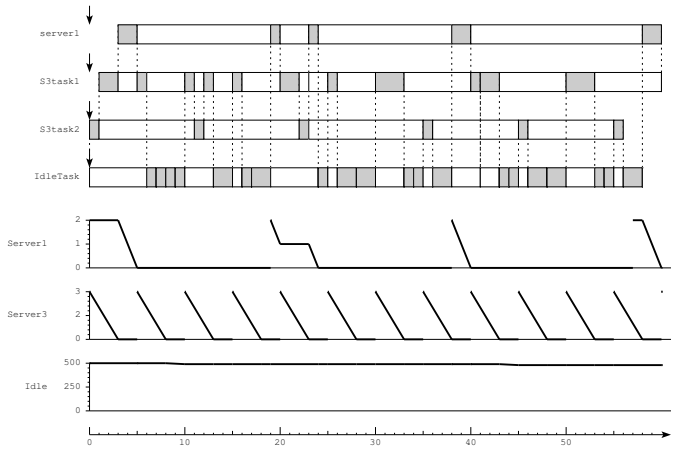

Figure 9.17: Execution trace of TIMES scheduler (non-fragmented), running system1.

IMA scheduler DEOS, used for safety critical domains such as aerospace and space. The scheduler assigns a period and budget to each thread, the scheduling policy used is RMA. The work of Muller *et al.* [44, 45] is most similar to our work. They use a domain specific language (DSL) to model schedulers (including hierarchical schedulers). The difference is that they verify that the scheduler is correct with respect to the kernel interface, and not the actual scheduling policy. Their framework support synthesis for early Linux kernel versions. Zerzelidis *et al.* [46] model a system with multiple schedulers, including resource sharing with SRP. The modelling tool UPPAAL is used, and the model is compatible with RTSJ. Each partition (local scheduler) has a priority level, but no release time or budget. The verification shows the absence of livelock/deadlock and the correctness of SRP.

Few papers touch upon the area of code-synthesis in the context of scheduler modelling. Hsiung *et al.* [47] presents a framework (VERTAF) for developing real-time embedded software. The application, as well as the scheduler is specified as UML diagrams. The framework does a transformation to extended timed automata (ETA) and model checking is used to verify properties such as livelock and deadlock. The framework supports code-synthesis for the OS's MontaVista Linux, μC/OS, Embedded Linux, and eCOS. Li *et al.* [48] introduce a meta-scheduler framework, compliant with POSIX-supported OS's.

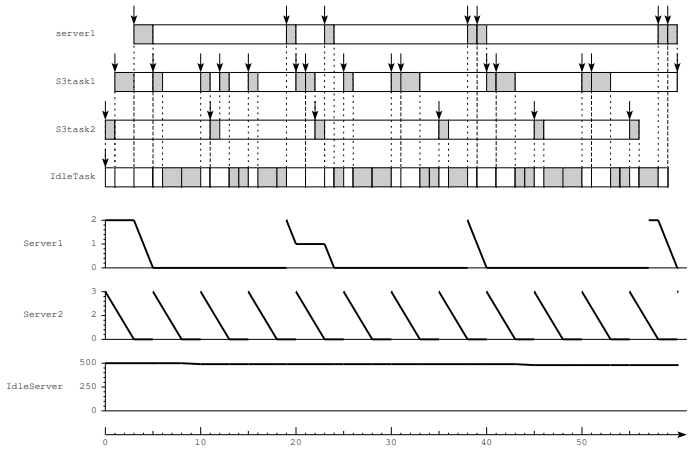

Figure 9.18: Execution trace of TIMES scheduler (fragmented), running system1.

Basically, the framework is a middleware layer which uses OS primitives, and it exports an interface to schedulers, which in turn are implemented by the users. The correctness of the framework is verified using UPPAAL. They implement several flat-schedulers in various platforms (VxWorks for example), and they measure the overhead of the schedulers.

To sum up, modelling of hierarchical scheduling has been done, but not specifically for two-tier FPPS with PS. To the best of our knowledge, there is no prior work on verification of hierarchical scheduling policies, nor code-synthesis (from model) for this type of scheduling.

9.7 Conclusion

In this paper we deal with modelling, verification and synthesis of hierarchically scheduled real-time systems. We have looked at two-level hierarchical scheduling, with fixed priority preemptive scheduling of periodic tasks/servers. The scheduler has been modelled using the task-automata language and the model was implemented in the Times tool. However, the Times tool does not support dynamic change of priorities, nor task suspension, which are two fundamental properties required when implementing hierarchical scheduling. In

the paper we show how to get around this problem through an innovative approach for how the system is modelled.

In addition we modelled $observers$ which monitored the behavior of the schedulers. We implemented rules for the observers, based on the criteria that we have specified as properties. These properties are appropriate behaviors that comply with hierarchical fixed priority preemptive scheduling of periodic tasks and servers. The observers are then modelled to enter error states if they detect a contradiction to any of our properties. We check that the observers do not enter these error states through the use of model checking. We use task/server systems that stress the schedulers to generate all combinations of scheduling events, so that we can verify the entire behavior of the hierarchical scheduler.

The code synthesis results showed a considerable difference in CPU between the generated schedulers and an equivalent manually coded scheduler. However, the worst-case memory consumption showed to be similar to each other.

To sum up, this paper presents a proof of concept, showing that we can model, verify, and generate source-code that executes a hierarchical scheduler on an industrial platform.

As future work, we plan to optimise the synthesis of the model by implementing a new (optimised) code generator. This will make the synthesis fully automated, which will open up the possibility to generate systems in a larger scale.

Bibliography

[1] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *2nd USENIX Symposium on OS Design and Implementation*, pages 107–121, Oct 1996.

[2] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *18th IEEE International Real-Time Systems Symposium*, pages 308–319, Dec 1997.

[3] John Regehr and John A. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In *22nd IEEE International Real-Time Systems Symposium*, pages 3–14, Dec 2001.

[4] Mikael Åsberg, Moris Behnam, Farhang Nemati, and Thomas Nolte. Towards Hierarchical Scheduling in AUTOSAR. In *14th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1181–1188, Sep 2009.

[5] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards Hierarchical Scheduling in VxWorks. In *4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 67–76, July 2008.

[6] Mikael Åsberg, Thomas Nolte, and Shinpei Kato. A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 380–387, Aug 2011.

[7] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some Practical Problems in Prioritized Preemptive Scheduling. In *7th IEEE International Real-Time Systems Symposium*, pages 181–191, Nov 1986.

[8] Rob Davis and Allan Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *26th IEEE International Real-Time Systems Symposium*, pages 389–398, Dec 2005.

[9] T.-W. Kuo and C.H. Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *20th IEEE International Real-Time Systems Symposium*, pages 256–267, Dec 1999.

[10] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *24th IEEE International Real-Time Systems Symposium*, pages 2–13, Dec 2003.

[11] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems. In *7th ACM International Conference On Embedded Software*, pages 279–288, Oct 2007.

[12] R. I. Davis and A. Burns. Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems. In *27th IEEE International Real-Time Systems Symposium*, pages 257–270, Dec 2006.

[13] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task Automata: Schedulability, Decidability and Undecidability. *Journal of Information and Computation*, 205(8):1149–1172, Aug 2007.

[14] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Journal of Theoretical Computer Science*, 126(2):183–235, April 1994.

[15] Tobias Amnell, Elena Fersman, Paul Pettersson, Wang Yi, and Hongyan Sun. Code Synthesis for Timed Automata. *Nordic Journal of Computing*, 9(4):269–300, Dec 2002.

[16] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A Tool for Modelling and Implementation of Embedded Systems. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464, April 2002.

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

[18] L. Andriantsiferana, Jean-Pierre Courtiat, Roberto C. de Oliveira, and L. Picci. An Experiment in using RT-LOTOS for the Formal Specification and Verification of a Distributed Scheduling Algorithm in a Nuclear Power Plant Monitoring System. In *17th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, pages 433–448, Nov 1997.

[19] Mikael Åsberg. Model of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling. Technical Report 2379, Mälardalen University, 2011.

[20] Leonard Lensink, Sjaak Smetsers, and Marko Van Eekelen. Machine Checked Formal Proof of a Scheduling Protocol for Smartcard Personalization. In *12th International Conference on Formal Methods for Industrial Critical Systems*, pages 115–132, July 2007.

[21] Gudmund Grov, Greg Michaelson, and Andrew Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *13th International Conference on Parallel and Distributed Systems*, pages 1–6, Dec 2007.

[22] Mike Holenderski, Martijn Heuvel, Reinder Bril, and Johan Lukkien. Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 37–42, July 2010.

[23] Robert Glaubius, Terry Tidwell, William D. Smart, and Christopher Gill. Scheduling Design and Verification for Open Soft Real-Time Systems. In *29th IEEE International Real-Time Systems Symposium*, pages 505–514, Nov 2008.

[24] X. Feng and A. Mok. A Model of Hierarchical Real-Time Virtual Resources. In *23rd IEEE International Real-Time Systems Symposium*, pages 26–35, Dec 2002.

[25] G. Lipari and S. Baruah. Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems. In *6th IEEE Real Time Technology and Applications Symposium*, pages 166–175, May 2000.

[26] G. Lipari and E. Bini. Resource Partitioning Among Real-Time Applications. In *15th IEEE Euromicro Conference on Real-Time Systems*, pages 151–158, July 2003.

[27] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving Real-time Systems Using Hierarchical Scheduling and Concurrency Analysis. In *24th IEEE International Real-Time Systems Symposium*, pages 25–36, Dec 2003.

[28] S. Matic and T. A. Henzinger. Trading End-to-End Latency for Composability. In *26th IEEE International Real-Time Systems Symposium*, pages 99–110, Dec 2005.

[29] D. Kim, Y. Lee, and M. Younis. SPIRIT-uKernel for Strongly Partitioned Real-Time Systems. In *7th International Workshop on Real-Time Computing and Applications Symposium*, pages 73–80, Dec 2000.

[30] Martijn Heuvel, Mike Holenderski, Wim Cools, Reinder Bril, and Johan Lukkien. Virtual Timers in Hierarchical Real-time Systems. In *Work-in-Progress session of the 30th IEEE International Real-Time Systems Symposium*, pages 37–40, Dec 2009.

[31] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *22nd International Conference on Software Engineering*, pages 488–497, June 2000.

[32] Darren Cofer, Eric Engstrom, and Nicholas Weininger. Using Model Checking for Verification of Partitioning Properties in Integrated Modular Avionics. In *19th IEEE Digital Avionics Systems Conference*, pages 1D2/1–1D210, Oct 2000.

[33] Torsten K. Iversen, Kare J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-Checking Real-Time Control Programs - Verifying LEGO MINDSTORMS Systems Using UPPAAL. In *12th IEEE Euromicro Conference on Real-Time Systems*, pages 147–155, June 2000.

[34] Matthias Daum, Jan Drrenbcher, and Burkhart Wolff. Proving Fairness and Implementation Correctness of a Microkernel Scheduler. *Journal of Automated Reasoning*, 42(2-4):349–388, April 2009.

[35] Naren Narasimhan, Elena Teica, Rajesh Radhakrishnan, Sriram Govindarajan, and Ranga Vemuri. Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis. *Journal of Formal Methods in System Design*, 19(3):237–273, Nov 2001.

[36] Moritz Kleine, Björn Bartels, Thomas Gothel, and Sabine Glesner. Verifying the Implementation of an Operating System Scheduler. In *3rd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 285–286, July 2009.

[37] Didier Lime and Olivier H. Roux. Formal Verification of Real-Time Systems with Preemptive Scheduling. *Journal of Real-Time Systems*, 41(2):118–151, Feb 2009.

[38] Pao-Ann Hsiung and Shang-Wei Lin. Model Checking Timed Systems with Priorities. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 539–544, Aug 2005.

[39] Chen Shu and Wu-Guo Qing. Modeling and Formal Analysis of Real-Time System via CCS. In *1st International Symposium on Computer Science and Computational Technology*, pages 321–324, Dec 2008.

[40] L. Durante, R. Sisto, and A. Valenzano. Formal Specification and Verification of the Real-Time Scheduler In FIP. In *1st IEEE International Workshop on Factory Communication Systems*, pages 99–106, Oct 1995.

[41] Odile Nasr, Jean-Paul Bodeveix, Mamoun Filali, and Miloud Rached Irit. Verification of a Scheduler in B Through a Timed Automata Specification. In *21st ACM Symposium on Applied Computing*, pages 1800–1801, April 2006.

[42] Frank Singhoff and Alain Plantec. AADL Modeling and Analysis of Hierarchical Schedulers. In *ACM International Conference on SIGAda*, pages 41–50, Nov 2007.

[43] Vu Ha, Murali Rangarajan, Darren Cofer, Harald Rues, and Bruno Dutertre. Feature-Based Decomposition of Inductive Proofs Applied to Real-Time Avionics Software: An Experience Report. In *26th International Conference on Software Engineering*, pages 304–313, May 2004.

[44] Luciano Porto Barreto and Gilles Muller. Bossa: A Language-Based Approach to the Design of Real-Time Schedulers. In *10th International Conference on Real-Time Systems*, pages 19–31, March 2002.

[45] Julia L. Lawall, Gilles Muller, and Hervé Duchesne. Invited Application Paper: Language Design For Implementing Process Scheduling Hierarchies. In *14th ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 80–91, Aug 2004.

[46] Alexandros Zerzelidis and Andy Wellings. Model-based Verification of a Framework for Flexible Scheduling in the Real-Time Specification for Java. In *4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 20–29, Oct 2006.

[47] Pao-Ann Hsiung, Shang-Wei Lin, and Chao-Sheng Lin. Real-Time Embedded Software Design for Mobile and Ubiquitous Systems. *Journal of Signal Processing Systems*, 59(1):13–32, April 2010.

[48] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems. *IEEE Transactions on Software Engineering*, 30(9):613–629, Sep 2004.