Composition and synchronization of real-time components upon one processor

DOI:
10.6100/IR755346

Document status and date:
Published: 01/01/2013

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Composition and synchronization of real-time components upon one processor

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 12 juni 2013 om 16.00 uur

door

Martijn Marianus Henricus Petrus van den Heuvel

geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. J.J. Lukkien

Copromotor:
dr.ir. R.J. Bril

Composition and synchronization of real-time components upon one processor

A DISSERTATION

by

Martijn Marianus Henricus Petrus van den Heuvel

June 2013

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA Dissertation Series 2013-08

The work in this thesis is supported by Agentschap NL under the VERIFIED project (grant number: HTASI10003).

Printed by: Gildeprint Drukkerijen – The Netherlands

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-94-6108-443-9

Preface

This is a book of many colors, I could not have written just on my own. In the first place, I am extremely grateful to my co-promoter, dr.ir. Reinder J. Bril, for his guidance, discussions and his patience over the last years. We have spent many hours to exchange ideas and I have got to know you as a very demanding supervisor who pays back his student's efforts with exponential amounts of energy and constructive feedback to improve the quality of research. Some may experience this as overwhelming; I can just say: I have enjoyed the thrill. Secondly, I would like to thank my promoter, prof.dr. Johan J. Lukkien, for his support. Although our technical discussions were scheduled less frequently, their impact were significant and they made me re-think problems to the core. I am also grateful for the opportunities I was given by both of you to collaborate with other researchers, within the research group as well as abroad.

When I made my first steps into research (four years ago, in 2009), I had counsel of a team of experienced researchers. As a legacy of my master project, I could immediately practice writing my first research articles under the supervision of Reinder, who also supervised me during my master project, and under the long-distance coaching of prof. Dr.-Ing. habil. Christian Hentschel and Dipl.-Ing. Stefan Schiemenz from Brandenburg University of Technology (BTU), Germany. Christian and Stefan hosted me as a master student at BTU during the spring semester of 2009, which was very kind of you. It is thereafter, when I was coaching my master students, that I appreciated even more Stefan's non-exhaustive patience with his students and his lessons regarding reporting science to other researchers. Another remarkable moment was the joint conference presentation with Christian in Las Vegas (January 2011), graced by an award for the best runner-up research paper in a leading conference on consumer electronics. It was my pleasure to work with you and learn from you.

This doctorate dissertation takes a slightly changed research direction, targeted at safety critical embedded systems (such as in-vehicle systems). I would like to thank the expertise group, Systems Architecture and Networking (SAN), for creating a great working atmosphere and my (former) office mates, Tseesuren Batsuuri and Milosh Stolikj, for sharing a great time. My closest cooperations within SAN have been with my office mates dr.ir. Michał J. Holenderski (also known as "dr. Mike") and Uğur Keskin. It is astonishing how such different personalities conducted research together. We had many enjoyable moments and you were memorable company at our conference travels. I would also like to thank my other co-authors in the SAN group for sharing your ideas and for the interesting discussions: Ionut David, Sunder Aditya B. Rao, Wim Cools, dr.ir. Rudolf H. Mak, dr. Tanır Özçelebi and dr.ir. Richard Verhoeven. A special thanks goes to dr.ir. Pieter J.L. Cuijpers for your unselfish devotion of time for discussions – especially during Reinder's sabbatical – and for twisting every (scientific) problem. Your unique view on things makes discussions with you interesting. It certainly had impact – I think a positive one – on this dissertation. Last, and certainly not least, many thanks to dr.ir. Richard Verhoeven and to dr. Erik J. Luit for their support with technical exhibitions. You divested me of lots of project engineering, which allowed me to focus on my research niche.

The research results in this dissertation have also been influenced by a collaboration with various researchers from Mälardalen University (MDH). I would like to thank their dean, dr. Damir Isović, for the nice discussions with him and for trusting me as a coach for his selection of excellent exchange students. It was my pleasure to work with you and your students. I have also visited MDH for one week (March 2012); it was prof.dr. Thomas Nolte and dr. Moris Behnam who have kindly hosted me during my visit and they have visited us several times as well. I experienced our mutual visits as extensive, enjoyable weeks. I am therefore proud to present some of our joint research efforts as a chapter of this dissertation. I would also like to thank other researchers from MDH for sharing their ideas and discussing technical aspects, especially Mikael Åsberg and Rafia Inam. However, special thanks goes to Moris Behnam. During my visit to MDH, he spent his whole working week to warm (sometimes heated) discussions with me. Back home from our visits, we organized several conference calls to continue our discussions. Thank you, Moris, for your inspiring discussions and sharing your valuable ideas.

Following the lines of research in which I have been involved, I have (co-) supervised various master students during their graduation project or internship project: Coen Tempelaars, Alok Lele and Ashwini Moily (in cooperation with BTU) – Gowri Sankar Ramachandran, Syed M.J. Abdullah and Xiaodi Zhang (in cooperation with MDH) – Chidiebere G.U. Okwudire and Ivan Raul Lopez Guadarrama (within SAN). I can just hope that you enjoyed the cooperation as much as I did; it was my pleasure to work with you.

Thanks to the collaborations with a variety of researchers, as mentioned above, I co-authored several research articles, which allowed me to travel to international conferences in order to present and publish the articles. The conversations at those conferences with other researchers further inspired me and brought me new ideas. I would like to highlight a few persons: dr. Stefan M. Petters (CISTER, Portugal), dr. Nathan W. Fisher (Wayne State University, USA), dr. Marko Bertogna (University of Modena, Italy) and dr. Rob Davis (University of York, UK). Thank you for your useful discussions, feedback and suggestions which helped us to identify challenges and solutions to problems in our research domain. I would also like to thank the organizers of the summer school that I have visited in Pisa (June 2010), prof.dr. Giorgio Buttazzo (Scuola Superiore Sant'Anna, Italy) and his enthusiastic team, for sharing their knowledge and for the nice experience. The school brought me in touch with interesting fellow researchers I would not have met otherwise.

As time flies, we arrive at a moment of having actually composed this doctorate dissertation, sewed together from many pieces of collaborative work. I would like to thank the reviewers of this dissertation, prof.dr. Kees G.W. Goossens (Eindhoven University of Technology, The Netherlands), prof.dr. Thomas Nolte (Mälardalen University, Sweden) and dr. Giuseppe Lipari (Scuola Superiore Sant'Anna, Italy), for their useful feedback and their help to further improve the quality of this dissertation. Furthermore, I would like to thank prof.dr.ir. Kees C.H. van Berkel (Eindhoven University of Technology, The Netherlands) for participating in the doctorate committee.

Working towards a doctorate dissertation takes a significant period of time, sporadically occupied by stressful moments. I am grateful for the opportunities I had to meet many people with diverse backgrounds and interests and I want to thank them for spending spare time in an enjoyable way. In particular, I would like to thank the squash squad (led by Hrishikesh L. Salunkhe, SAN member) and E.S.B.V. Panache for keeping me fit physically as well as mentally. I also want to thank other friends for leisure and sharing insights. For example, the yearly March-April (MaPril) weekends were the herald of a relax spring. Finally, I owe many thanks to my family for supporting me unconditionally. Thank you all for showing your interest and coloring this book.

Martijn van den Heuvel
Eindhoven, June 2013

Abstract

Many industrial systems have various hardware and software functions for controlling mechanics. If these functions act independently, as they do in legacy situations, their overall performance is not optimal. There is a trend towards optimizing the overall system performance and creating a synergy between the different functions in a system, which is achieved by replacing more and more dedicated, single-function hardware by software components running on programmable platforms. This increases the re-usability of the functions, but their synergy requires also that (parts of) the multiple software functions share the same embedded platform.

In this work, we look at the composition of inter-dependent software functions on a shared platform from a timing perspective. We consider platforms comprised of one preemptive processor resource and, optionally, multiple non-preemptive resources. Each function is implemented by a set of tasks; the group of tasks of a function that executes on the same processor, along with its scheduler, is called a component. The tasks of a component typically have hard timing constraints. Fulfilling these timing constraints of a component requires analysis. Looking at a single function, co-operative scheduling of the tasks within a component has already proven to be a powerful tool to make the implementation of a function more predictable. For example, co-operative scheduling can accelerate the execution of a task (making it easier to satisfy timing constraints), it can reduce the cost of arbitrary preemptions (leading to more realistic execution-time estimates) and it can guarantee access to other resources without the need for arbitration by other protocols. Since timeliness is an important functional requirement, (re-)use of a component for composition and integration on a platform must deal with timing.

To enable us to analyze and specify the timing requirements of a particular component in isolation from other components, we reserve and enforce the availability of all its specified resources during run-time. The real-time systems community has proposed hierarchical scheduling frameworks (HSFs) to implement this isolation between components. After admitting a component on a shared platform, a component in an HSF keeps meeting its timing constraints as long as it behaves as specified. If it violates its specification, it may be penalized, but other components are temporally isolated from the malignant effects. A component in an HSF is said to execute on a virtual platform with a dedicated processor at a speed proportional to its reserved processor supply. Three effects disturb this point of view. Firstly, processor time is supplied discontinuously. Secondly, the actual processor is faster. Thirdly, the HSF no longer guarantees the isolation of an individual component when two arbitrary components violate their specification during access to non-preemptive resources, even when access is arbitrated via well-defined real-time protocols.

The scientific contributions of this work focus on these three issues. Our solutions to these issues cover the system design from component requirements to run-time allocation. Firstly, we present a novel scheduling method that enables us to integrate the component into an HSF. It guarantees that each integrated component executes its tasks exactly in the same order regardless of a continuous or a discontinuous supply of processor time. Using our method, the component executes on a virtual platform and it only experiences that the processor speed is different from the actual processor speed. As a result, we can focus on the traditional scheduling problem of meeting deadline constraints of tasks on a uni-processor platform. For such platforms, we show how scheduling tasks co-operatively within a component helps to meet the deadlines of this component. We compare the strength of these cooperative scheduling techniques to theoretically optimal schedulers.

Secondly, we standardize the way of computing the resource requirements of a component, even in the presence of non-preemptive resources. We can therefore apply the same timing analysis to the components in an HSF as to the tasks inside, regardless of their scheduling or their protocol being used for non-preemptive resources. This increases the re-usability of the timing analysis of components. We also make non-preemptive resources transparent during the development cycle of a component, i.e., the developer of a component can be unaware of the actual protocol being used in an HSF. Components can therefore be unaware that access to non-preemptive resources requires arbitration.

Finally, we complement the existing real-time protocols for arbitrating access to non-preemptive resources with mechanisms to confine temporal faults to those components in the HSF that share the same non-preemptive resources. We compare the overheads of sharing non-preemptive resources between components with and without mechanisms for confinement of temporal faults. We do this by means of experiments within an HSF-enabled real-time operating system.

Note to the reader

This dissertation considers the modeling of timing constraints of complex embedded systems in a way that enables us to compose such systems from independently analyzed (software) components. Since an important functional metric of many complex systems is timing, the desired timing properties of a system and its components should be modeled and analyzed carefully. This dissertation focusses on the underlying techniques in two book parts.

In the first part of this dissertation, we survey existing techniques for modeling timing properties of so-called *real-time systems* and we extend those techniques with our best practices to model synchronization between components. This part includes an introductory background in the research field, a discussion of our most important research results and directions for future research. The second part of this dissertation presents a selection of research papers. These papers describe our research contributions in detail and they complement our modeling techniques with programming techniques and algorithms to optimize the processing resources required by a real-time system, while respecting its timing constraints.

Contents

Part I

Composition and synchronization of real-time components upon one processor

1 INTRODUCTION

Many industrial systems have various hardware and software functions for controlling the real physical world through electronics and mechanics. Examples can be found in a wide range of application domains, including manufacturing machines, highly dependable medical devices, intelligent transportation and in-vehicle control. The traditional approach in the design and development of these embedded systems makes use of an architecture that simply composes sets of self-contained hardware platforms. These hardware platforms are connected by several network buses. Each set of inter-connected platforms typically hosts a single, independently developed functional unit [1]. The last few years have shown, however, that this federated architectural approach can no longer stand. Firstly, since the number of electronic functions in embedded systems is ever increasing, it becomes too expensive to make dedicated hardware units for each function. Secondly, there is a trend towards optimizing the overall system performance and creating a synergy between the different functions in a system.

In short, more and more dedicated, single-function hardware will continue to be replaced by increasingly complex software functions running on programmable platforms. If these software functions act independently from other functions, as they do in legacy and early development situations, their overall performance is not optimal. A revolutionary performance increase of embedded systems comes from extensive networking between embedded electronic computers and from the composition and the integration of independently developed software functions. The aim of composition in here is to increase the re-usability of the functions, regardless of their inter-dependence.

Various industrial standards have been developed over the years to promote the exchange and the integration of certified software components from multiple vendors. Each function is implemented by a set of tasks; the group of tasks of a function that executes on the same processor, along with a scheduler for those tasks, is called a component. For example, a consortium of automotive suppliers and manufacturers has developed the AUTOSAR standard for the exchange of such component software. As another example, the ARINC specification 653-2 describes the interface between component software and underlying middleware in a distributed avionics system. However, many standards, including AUTOSAR and ARINC 653-2, lack explicit information about timing requirements of components in their meta-model. Since timeliness is an important requirement for many embedded systems, re-use of a software component for composition in an integrated platform must deal with timing.

In this work, we look at the composition of software components on a shared platform from a timing perspective. We consider platforms comprised of one preemptive processor resource and, optionally, multiple non-preemptive resources. Since the tasks of a component typically have hard timing constraints, it requires analysis to verify that the timing constraints of a component are fulfilled. Our ultimate goal is to develop tools to reduce development time of embedded software and to prove that integrating independently developed components upon a shared platform is functional, safe and cost effective.

1.1 Motivating example

Today's vehicles contain an ever increasing number of electronic functions, see Figure 1. These electronic control functions consist of various components that replace mechanical control systems (also called X-by-wire), so that some of the traditional mechanics, e.g., the steering column, can be left out. Each function is also a cyber-physical system by itself [2], which holistically integrates sensing, actuation, computation, networking and physical processes. These X-by-wire functions are complex, distributed and interdependent.

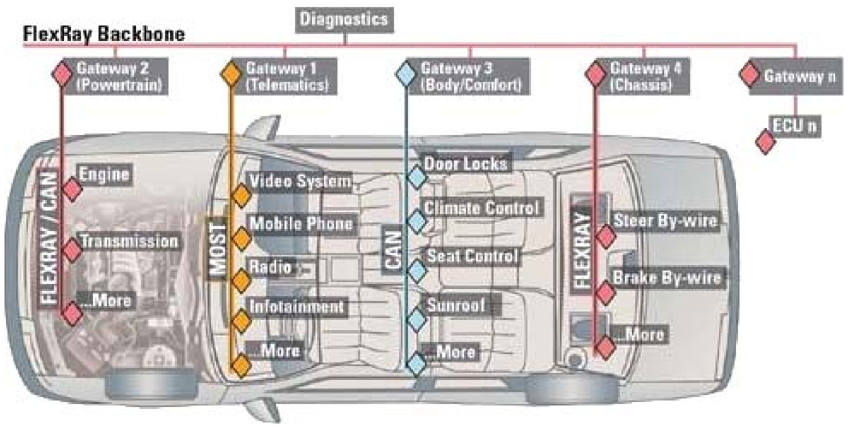

Fig. 1. A modern car is equipped with many embedded computers interconnected via different kinds of communication busses. This infrastructure facilitates more and more electronic, X-by-wire applications.

For example, a function, called integrated vehicle dynamics control (IVDC), is meant to optimize the coordination of multiple chassis control functions in a vehicle [3]. The main goal of IVDC is to stabilize a vehicle in critical situations and to improve its handling performance by reducing oversteer or understeer. This type of control when applied to braking actuators only is sometimes referred to as electronic stability program (ESP). ESP has already proven to have a positive impact on the accident rates. Recent literature indicates that there is still room for further improvement by adding more X-by-wire functions, such as steer-by-wire, to the stability control [3]. However, combining different control functions is not trivial and their synergy requires also that the multiple software components of different functions share the same embedded platform. The development of electronic functions such as IVDC brings many technical challenges, ranging from distributed programming to control theory and from timing analysis to the composition of components.

The current market situation reinforces these challenges, because adding a new function into a vehicle often means purchasing pre-manufactured hardware and software with little information about the internal behavior of the function [1]. The AUTOSAR consortium recently attempted to introduce a standard in which suppliers provide only the functional software components associated with well-defined behavioral component models and interfaces. Industrial standards like AUTOSAR thereby aim at giving system integrators better control over the functional cooperation, communication and synchronization between different components of arbitrary vendors. The underlying operating system and middleware – responsible for standardized application programming interfaces (APIs) with underlying support for multi-tasking and synchronization – are also subject to AUTOSAR certification. These layers should protect a component against propagation of faults caused by other components.

The challenge of the system integrator is then to satisfy the resource requirements of the composed components. Many components, especially those that implement control functionality, are sensitive to timing and fluctuations in actuation delays [4]. Furthermore, some components must share several actuation devices, such as brakes and the steering wheel. Software components may also share multiple input and output (I/O) devices and software pieces [1], e.g., object detection. These forms of resource sharing inherently require interrupt handling or contiguous mutual-exclusive execution of components, which may further impact I/O delays experienced by the control tasks. Failing to meet timing constraints of a component may lead to an unstable vehicle and unsafe situations.

Because we consider software components that are sensitive to timing, the individual components and the composed system require timing analysis. This work investigates the composition of inter-dependent software components, including the underlying operating-system support, from a timing perspective.

1.2 Problem description

Embedded systems are increasingly taking advantage of the opportunities offered by better programmable and more powerful hardware platforms. This leads to system designs with new software functions and solutions that share hardware resources and that have hard real-time constraints associated with them. The design of embedded systems requires the development of cost-effective platforms, i.e., a selection of off-the-shelf hardware, real-time operating system and programming framework supported by novel tools and technologies.

To cope with the ever increasing complexity of embedded systems, we revisit hierarchical scheduling frameworks (HSFs). These frameworks enable composition and isolation of independently developed, real-time components upon a single processor through a hierarchy of schedulers and reservations. If these real-time components also act independently, as they do in legacy and development situations, their overall performance is not optimal. There is a trend towards optimizing the overall system performance and creating a synergy between the different components in a system. The emerging synergy between different components calls for innovative technologies, methodologies and algorithms to establish predictable software synchronization.

We focus on two types of synchronization between tasks located in arbitrary components:

1) **a-synchronous resource sharing and communication (e.g., event or interrupt driven).** A component, that shares the processor with other components, may cause unpredictable overheads to those other components due to the handling of external events such as I/O interrupts, software signals or timer interrupts. This is the result of propagating an event immediately, i.e., without delaying the delivery of an a-synchronous event until the destined task has the highest priority in the composed system. Moreover, a component may receive a continuous processor supply in a federated architecture, because it has the entire processor at its disposal. In an integrated system, however, the processor needs to be shared and it is inherently available for execution discontinuously. If the tasks of a component are triggered by external events, an integrated component may execute its tasks in a different order under a continuous than under a discontinuous supply of processor time.
2) **synchronous resource sharing and communication (via so-called non-preemptive resources).** The HSF no longer guarantees isolation of an individual component when an arbitrary other component violates its specification during the execution of its tasks on non-preemptive resources that require mutually exclusive access, even when access is arbitrated via well-defined real-time synchronization protocols. This problem becomes even more apparent when a component developer (consciously or unconsciously) exploits protocol-specific timing glitches. In addition, a component that has been developed with the explicit support of one particular synchronization protocol cannot be re-used in an HSF when the synchronization protocol is changed.

We focus on the problems associated with these two types of resource sharing and communication between tasks in arbitrary components.

1.3 Contributions

In this work we revisit frameworks for designing hierarchically scheduled systems. We cover the system design from component requirements to run-time resource allocation and arbitration of components upon a single processor and other resources.

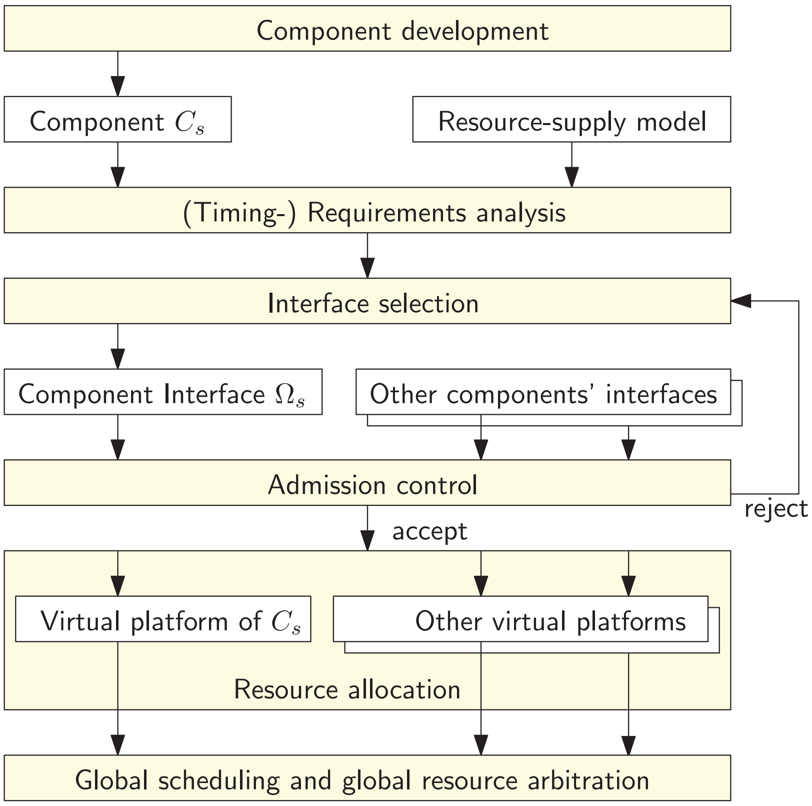
Component development
Component C_s
Resource-supply model
(Timing-) Requirements analysis
Interface selection
Component Interface Ω_s
Other components' interfaces
Admission control
reject
accept
Virtual platform of C_s
Other virtual platforms
Resource allocation
Global scheduling and global resource arbitration

Fig. 2. The design stages of an hierarchically scheduled system.

The composition of components into an hierarchically scheduled system involves several design phases, see Figure 2. Our main objectives are: (*i*) independent development of components, (*ii*) independent (timing) analysis of components and (*iii*) abstraction of resource sharing between component. The latter objective especially calls for enhancements of the earlier proposed design phases of a compositional system. In short, our contributions are as follows.

Component development. During the development of a component, a designer has to split a piece of software into concurrently executing units called tasks. Part of this work is to protect *critical sections*, requiring contiguous mutual-exclusive execution, through a synchronization protocol. When a task executes in a critical section it is said to execute on a non-preemptive resource.

We make the API to non-preemptive resources transparent during the development cycle of a component (Paper D), i.e., the developer of a component can be unaware of the actual protocol being used in an HSF to realize mutual-exclusive execution on that resource. Components can therefore be unaware that access to non-preemptive resources requires global arbitration (beyond their virtual platforms). This supports the independent development of components.

Requirements analysis. At design time, the component designer must characterize the timing requirements of the component and derive an appropriate timing interface that summarizes these requirements. We distinguish two aspects here (see Paper C): (*i*) the choice of an interface representation, i.e., determined by the choice of a resource-supply model, and (*ii*) the selection of the interface parameters under the auspices of the chosen resource-supply model.

Furthermore, we standardize the way of computing the interface parameters of a component, even in the presence of non-preemptive resources. We can therefore apply the same timing analysis to the components in an HSF as to the tasks inside, regardless of their scheduling or their protocol being used for non-preemptive resources. The latter property, i.e., that the component's timing analysis is independent of the global synchronization protocol, is called *opacity*. This property increases the re-usability of the timing analysis of components.

We also show how scheduling tasks co-operatively within a component, i.e., by a limited-preemptive local task scheduler, helps to meet the deadlines of this component (Paper A). We compare the strength of these scheduling techniques to theoretically optimal schedulers. The limited-preemptive schedules of components may result in tighter resource requirements of a component.

Interface selection. Given a resource-supply model, the designer needs to compute an interface, so that the component is guaranteed to satisfy its deadline constraints on any platform that satisfies its interface. The optimal interface for a component is subject to different trade offs. For example, Lipari and Bini [5] derived an interface for an independent component based on the bounded-delay model while trading off the maximum service delay versus the maximum processor bandwidth.

In the presence of non-preemptive resources that need to be shared between tasks of different components, the selection of interface parameters becomes even more complicated. On the one hand, the more preemptive the tasks in a component execute, the better their responsiveness. On the other hand, the more preemptive the tasks in a component execute, the longer the component as a whole may keep control over a non-preemptive resource; this may lead to long blocking durations between components. We propose a method to trade off these interface parameters of a component in a computationally tractable way (Paper C).

Admission control. When a component needs to execute on a shared platform with other components, it presents its timing interface to the admission-control algorithm. The admission controller determines whether or not the component can enter the system without compromising the schedule of the other components. If a component is admitted successfully, the component is allocated a *virtual platform* dimensioned according to its specification in the timing interface. If the component is rejected, the designer must go back to the interface specification.

We propose a method to select an interface of a component in such a way that the composition of components will return a configuration that can be scheduled successfully whenever possible[1] (Paper C). Our method selects new interface parameters when necessary and the selection procedure searches the design space in a computationally tractable way.

Resource allocation. If a set of components is admitted on a shared platform, each of the components is allocated a virtual platform. A virtual platform implements an abstraction layer, so that all resources required by the hosted component appear to be available without the need of sharing those resources with other components. For example, the virtual platform implements a service policy (i.e., a *server*, in short) for the allocation of a virtual processor.

The virtual platforms are then together scheduled on the physical resources. On a shared platform, however, a virtual platform delivers the processor time discontinuously and the actual processor is faster than the virtual processor. This disturbs the desired view of a virtual processor, because the order in which external events are handled within the component during run time may change with respect to the execution order on a continuous processor.

We present a novel scheduling method – called *virtual scheduling* – that enables us to compose a component into an HSF with other components (Paper B). It guarantees that each integrated component executes its tasks exactly in the same order regardless of a continuous or a discontinuous supply of processor time. Using our virtual-scheduling method, the component executes on a virtual platform and it only experiences that the processor speed is different from the actual processor speed. As a result, external events are delivered to a component when that component is executing on the processor. This prevents the handling of events destined for suspended components; thus, we avoid that event-handling consumes resources allocated to other components. Furthermore, we can now focus on the traditional scheduling problem of meeting deadline

1. This optimality criterion holds under the assumption that the interface parameters of a component are computed opaquely, i.e., independently of the global scheduling and resource-arbitration policies which decide when the required resources are actually delivered.

constraints of tasks on a continuous uni-processor.

Global scheduling and global resource arbitration. After we have reserved all the specified resources of the components, we must schedule and enforce the availability of these resources during run-time. In paper D we investigate the scheduling overheads in a real-time operating system that we have extended with an HSF and with various real-time synchronization protocols. Paper D ignores the enforcement of reserved resources other than the processor, however.

The promise of a virtual platform is that a component keeps meeting its timing constraints as long as it behaves as specified. If it violates its specification, it may be penalized, but other components are temporally isolated from the malignant effects. The HSF no longer guarantees the isolation of an individual component when two arbitrary components violate their specification during access to non-preemptive resources, even when access is arbitrated via well-defined real-time protocols.

We therefore complement the existing real-time protocols for arbitrating access to non-preemptive resources with mechanisms to confine temporal faults to those components in the HSF that share the same non-preemptive resources (see Paper E). We compare the overheads of sharing non-preemptive resources between components with and without mechanisms for confinement of temporal faults (see Paper F). We do this by means of experiments within the same HSF-enabled real-time operating system as used in Paper D.

1.4 Outline

The remainder of the first part of this work focusses on (*i*) timing-requirements analysis, (*ii*) the representation of component interfaces and the derivation of the corresponding interface parameters and (*iii*) the admission control of components. The programming framework for the development of a component and the algorithms for selecting the best interface of a component are advanced topics and these topics are therefore kept for the second part of this work.

The outline of this first part is as follows. Section 2 gives an overview of several real-time task models that provide means for timing-requirements analysis of an individual component. This section also introduces the notion of resource sharing between tasks of the same component and it recapitulates the corresponding analysis. Section 3 presents the current state-of-the-art techniques for composing real-time models in a hierarchically scheduled system. We present guidelines and rules how to abstract from the actual execution of the local tasks within a component on any required resource. This gives a preview of our newly contributed techniques for designing compositional systems in which we implement an admission controller for components as if components were simple tasks. Section 4 summarizes our most important research results. Finally, Section 5 proposes future research directions.

2 Real-time models, their performance and complexity

For the timing analysis of real-time components, we take a process view of a component. This means that a component has been synthesized into concurrently executing units, i.e., tasks, for which access to the processor (as well as other resources) is arbitrated by a local scheduler. The mapping from the original application's functional entities to run-time tasks may be complex, e.g., deriving local deadlines of tasks in a distributed system can be complex [6]; however, this complexity falls beyond the scope of our work.

We are therefore no longer interested in the functional models. This prevents that changes in one function (or application) causes a chain of changes in components that have to execute on the same platform. At the end, we are interested in satisfying all the timing constraints in the system by means of composing independently developed and analyzed real-time components. This means that abstraction of local scheduling and local tasks is desired, so that a component fits in the context of standard real-time scheduling models.

In this section, we look at real-time task and scheduling models. Hence, we consider the situation where a single component has the entire processor and all resources at its disposal. We look at the following triangle of metrics:

1) **performance:** in this work, we limit the performance measure of a task to its schedulability. A task is schedulable, if it can complete all its jobs prior to its deadline. The better the performance is of a real-time task model, the more task sets are schedulable.
2) **abstraction:** a model abstracts from the real timing behavior of a task. The more details that are left out of a timing model, the harder it is to give realistic and accurate performance guarantees of a task.
3) **complexity:** given a set of individual models of tasks, we compose these models in such a way that the resulting system satisfies all the deadlines of each of the tasks. We are interested in the algorithmic complexity of the composition.

In our work, we look at methods to optimize the number of task sets that can be scheduled under various abstraction levels and under various ways of composition (Paper A and Paper C). We also compare the performance of the different models in terms of scheduling overheads (Paper D and Paper F).

The remainder of this section is as follows. Firstly, we investigate different abstraction levels of modeling the timing behavior of the jobs executed by a set of independent tasks. We also investigate the impact on the complexity of the analysis when those tasks have to execute on a shared processor. Secondly, we look how synchronization between tasks changes these models. Synchronization protocols make it possible to accommodate for limited dependencies between tasks, i.e., they implement mutual-exclusive execution between tasks. For example, a task requires mutual exclusion with other tasks when it executes on a shared non-preemptive resource.

2.1 Task models

In this section, we give a brief overview of commonly used task models in real-time uni-processor scheduling. A system comprises a set of n independent tasks, $\tau_1 \dots \tau_n$, that share a single processor. Each task τ_i generates an infinite sequence of jobs, counted from $0 \dots k-1$. The k-th job $J_{i,k}$ of a task τ_i is characterized by an arrival time $a_{i,k}$, a relative deadline $D_{i,k}$ and a worst-case execution time (WCET) $E_{i,k}$. Job $J_{i,k}$ is said to be schedulable, if it is able to complete its WCET of $E_{i,k}$ time units within $D_{i,k}$ time units from its arrival.

As scheduling policies, we consider earliest deadline first (EDF) and fixed-priority scheduling. EDF is an optimal policy for scheduling independent jobs upon a preemptive processor. EDF dynamically assigns a static priority, $\pi_{i,k}$ to each job of a task, i.e., its absolute deadline $d_{i,k} = a_{i,k} + D_{i,k}$ and $\pi_{i,k} = d_{i,k}$. Fixed priority scheduling of tasks assumes the same static priority, π_i, of all jobs of the same task. At each moment in time, the scheduler selects the job with the highest priority among all the jobs that have pending execution requests on the processor[2], i.e., jobs are *fully preemptive* and the scheduler is *work conserving*.

We start with the basic model by Liu and Layland [7]. We also adopt their basic assumptions, i.e., jobs do not suspend themselves, a job of a task does not start before its previous job is completed, and the overhead of context switching and task scheduling is ignored. A real-time task model may make further assumptions or add task parameters, so that efficient analysis can be developed to determine whether or not the jobs of all tasks in a system are able to satisfy their deadline constraints. Figure 3 briefly summarizes of the relation between several task models.

Periodic task model [7]. Following the seminal model of Liu and Layland [7], each task has a period, T_i, and a WCET, $E_i = \max\{E_{i,k} \mid 0 \dots k-1\}$; these parameters are fixed for all jobs generated by task τ_i. Whereas Liu and Layland [7] assumed *implicit deadlines* of tasks (i.e., $D_i = T_i$), Leung and Whitehead [8] have explicitly added a deadline constraint to each task, i.e., each job has to complete its execution time within D_i time units from its arrival. We use the short-hand notation $\tau_i = (T_i, E_i, D_i)$ for characterizing a periodic task. If each task of a task set has a deadline at most equal to its period, then the task set is called *deadline constrained*. Many control algorithms can be developed considering only periodic actuation and, thus, they can be implemented as hard real-time periodic tasks [4].

2. Strictly spoken, jobs are the units of concurrency being scheduled. In many *restricted* task models, however, each task is assumed to have just one job with pending execution requests at each moment in time. Tasks and jobs are therefore often used interchangeably.

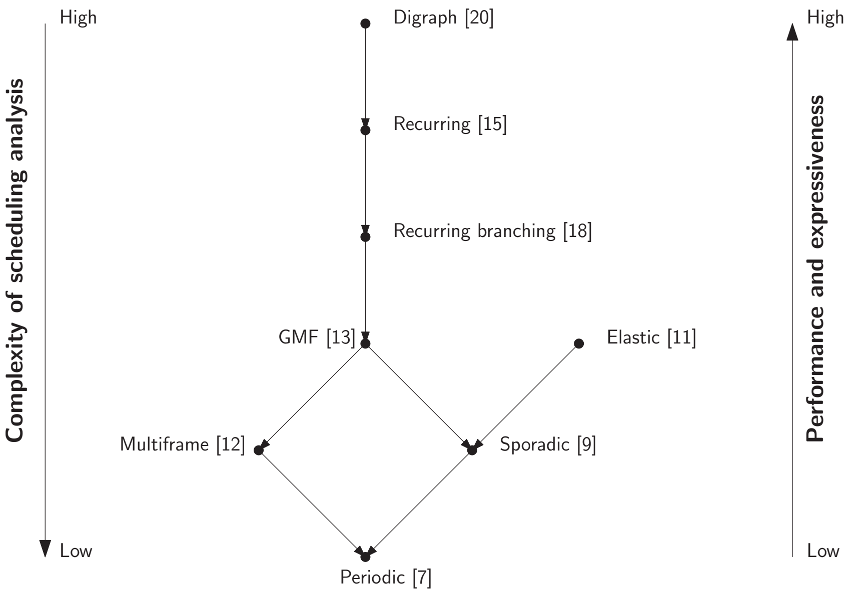

Fig. 3. Overview of task models where arrows indicate the specialization relationship. The higher a task model resides in the hierarchy, the higher its expressiveness (less abstract). The lower a task model resides in the hierarchy, the more abstraction we have and the less expensive the scheduling analysis is.

Sporadic task model [9]. Rather than a fixed separation of T_i time units between two subsequent job arrivals of task τ_i, a sporadic task generates a sequence of jobs which are separated by at least T_i time units. The period parameter of a task, T_i, therefore determines the *minimal inter-arrival time* of subsequent jobs. Each sporadic job may arrive at an arbitrary moment in time, i.e., it may delay its arrival for an arbitrarily long period.

A sporadic task can be seen as a sporadically periodic task which exhibits its worst-case processor demand when subsequent jobs arrive separated minimally in time [10]. That is, similar to a periodic task under arbitrary phasing.

Elastic task model [11]. Apart from a nominal period T_i, each task has a minimum period $T_{i,\min}$ and and maximum period $T_{i,\max}$. Given these extra bounds on the fluctuations of a task's period, one can compute the fluctuations in the response times of tasks by either assuming the worst-case individual utilization of a task (i.e., tasks are assumed to have a rate $T_{i,\min}$) or the best-case individual utilization of a task (i.e., tasks are assumed to have a rate $T_{i,\max}$).

Buttazzo et al. [11] have applied this model to control overload situations of sporadic tasks by means of dynamically adapting the nominal periods of individual tasks within the boundaries of $[T_{i,\min}, T_{i,\max}]$.

Multi-frame task model [12]. Each job of a multi-frame task may specify a different WCET, $E_{i,k}$. The job sequences generated by the multi-frame task (infinitely) cycle through the static list of job types.

Although the original model presented in [12] consider only implicit deadlines of tasks, this assumption is lifted by Baruah et al. [13]. Moreover, Baruah et al. [13] allow a different minimum separation, $T_{i,k}$, and a different relative deadline $D_{i,k}$ between two subsequent job arrivals of the same task. This model is referred to as the generalized multi-frame (GMF) task model. GMF tasks may implement state machines, as in some avionics and automotive applications, with a well-defined cycle and a WCET for each state [14].

Recurring task model [15]. The recurring task model [15] generalizes the sporadic task model [9], the GMF task model [13] and scheduling with internal deadlines [16]. A recurring task τ_i is represented by a directed a-cyclic graph (DAG) with a unique source and a unique sink[3]. Each vertex u in the graph represents a sub-task $\tau_{i,u}$, with its own WCET $E_{i,u}$ and an arbitrary deadline $D_{i,u}$. Whenever a job of a sub-task is triggered, it needs to complete its workload $E_{i,u}$ within $D_{i,u}$ time units. Each edge (u, v) in the graph is labeled with a value $T_{i,v}$ indicating the minimum separation between a trigger of sub-job $\tau_{i,u}$ and a trigger of $\tau_{i,v}$. Inherited from the sporadic task model, two subsequent arrivals of the source vertex are at least separated by T_i time units.

Baruah [15] presented a demand-bound test which checks whether or not all (internal) deadline constraints of a given task set are satisfied under EDF scheduling of tasks. The scheduling analysis under both EDF as well as fixed priority assignments of recurring tasks have been presented in [17]. These schedulability tests have an exponential worst-case complexity in the number of vertices of a task. When the DAG is a tree, the task model is called the *(recurring) branching task model* [18]. This model has a simplified analysis. A task can be analysed within a cubic complexity in the number of vertices and the analysis of an entire task set has a pseudo-polynomial time complexity.

Anand [19] has extended the recurring branching task model by explicitly modeling different lengths of branches and by explicitly modeling control variables at decision points.

Digraph task model [20]. The digraph task model [20] generalizes the recurring task model by allowing cyclic directed graphs. The analysis for EDF-scheduled digraph-task systems has been presented in [20] and [21].

Complexity of analyzing a schedule

Given a set of tasks, we would like to analyze whether or not all the tasks' deadlines are satisfied. The analysis of the schedule should preferably be

3. Both multi-frame task models in [12] and [13] are restricted to a chain of vertices.

computationally efficient. The algorithmic complexity of analyzing a schedule does not only depend on the expressiveness of the task model, it also depends on the scheduling policy. Table 1 presents an overview.

To analyze the schedulability of a task set, the jobs in a sufficiently large time window need to considered. The first problem of analyzing a schedule is to bound the length of the time window adequately. The second problem is then to define an appropriate upper bound on the workload of the task set within this window. In order to deem a task set schedulable, we must check whether or not all jobs in a so-called synchronous busy interval finish prior to their deadline. The synchronous busy interval is defined by the longest time window that the processor is busy with processing a job that is simultaneously released with a higher-priority job of other tasks and it includes the time required to process the higher-priority jobs subsequently released by other tasks.

One way of determining whether or not all the jobs meet their deadline is by computing the response time of the jobs. Joseph and Pandya [23] have developed response-time analysis for fixed-priority scheduling of sporadic tasks and Spuri [24] has presented response-time analysis for EDF scheduling of sporadic tasks. Contrary to fixed-priority scheduling of sporadic tasks, under EDF scheduling of sporadic tasks the synchronous busy interval is not unique for a set of tasks [24]. It might be necessary to consider several alignments of tasks and check all jobs in the resulting busy interval in order to find the worst-case response time of an EDF-scheduled task. Computing the response times of tasks can therefore be complicated and it is not always the most suitable method for analyzing a schedule. Fortunately, there exist boolean schedulability tests which allow to analyze the schedule of a task set within a finite time interval while assuming an initial synchronous release of all tasks in the system, even with EDF scheduling of tasks [10]. For more complicated task models with fixed task priorities, however, the analysis of a synchronous busy interval may result in just a sufficient scheduling analysis [22].

A schedulability test is considered to be efficient in the real-time community if resource-augmentation bounds exist. In other words, given a unary representation of a task set as an input, i.e., a finite sequence of jobs generated

TABLE 1
Overview of the algorithmic complexity of exact scheduling analyses under various task models (as it is presented by Stigge et al. [22]).

Task model	EDF scheduling	Fixed-priority scheduling
Periodic [7] and sporadic [9] task model	pseudo-polynomial	pseudo-polynomial
Generalized multi-frame task model [13] Recurring (branching) task model [15], [18] Digraph task model [20]	pseudo-polynomial	strongly coNP-hard

by the tasks within a certain time interval, the schedulability problem of the task set can be solved in polynomial time complexity in its input size. The algorithms for analyzing a schedule are therefore called weakly NP-hard and the algorithmic complexity of such a test is said to be *pseudo-polynomial*.

This class of pseudo-polynomial scheduling problems gives the opportunity to develop fully polynomial-time approximation schemes (FPTASes) in terms of *resource-augmentation bounds*. Intuitively, a resource-augmentation bound gives an upper bound on the processor speed that a task set requires continuously in order to avoid deadline misses. The resource-augmentation bounds of an FPTAS must be interpreted as follows:

- if an FPTAS indicates that a task set is schedulable, then the processor is at most $1 + \epsilon$ times as fast as required by an exact schedulability test, where ϵ can be selected by the system designer in order to trade the algorithmic complexity of the FPTAS against the precision. In other words, if the FPTAS deems a task set schedulable, then an exact test will fail the task set when the WCETs of the tasks are multiplied with a factor larger than $1 + \epsilon$.
- if the FPTAS fails to schedule a task set, then no further conclusions can be drawn.

FPTASes have already been developed for the timing analysis of periodic tasks and sporadic tasks. For example, Albers and Slomka [25] presented an FPTAS for sporadic tasks scheduled by an EDF scheduler and Fisher and Baruah [26] presented an FPTAS for sporadic tasks scheduled under a fixed-priority assignment[4]. Note that computing response times of tasks is NP-hard in the strong sense [28], because response times cannot be approximated within polynomial bounds. It is possible, however, to derive response times of sporadic tasks with a constant resource-augmentation bound [29].

The approximation of the schedulability analysis is beyond the scope of this work. Nevertheless, it is useful to understand the reasoning behind these polynomial-time approximation schemes. In this work we will design algorithms (see paper A and Paper C) to optimize the scheduling of dependent sporadic tasks, i.e, tasks that share more resources than just the processor. Our algorithms typically need to validate a schedulability condition in each step of the optimization. The existence of efficient approximation techniques for the verification of a schedule indicates the importance of developing algorithms to traverse the design space of a system within polynomial time complexity.

Finally, we recall that the relative algorithmic complexity of the various task models must be interpreted within the context of the basic scheduling assumptions of Liu and Layland [7]. For example, there is evidence [30] that the

4. The FPTAS by Fisher and Baruah [26] for fixed-priority scheduling of tasks is correct for deadline-constrained task sets only; it is flawed for arbitrary deadlines of tasks, which has been corrected by Nguyen et al. [27].

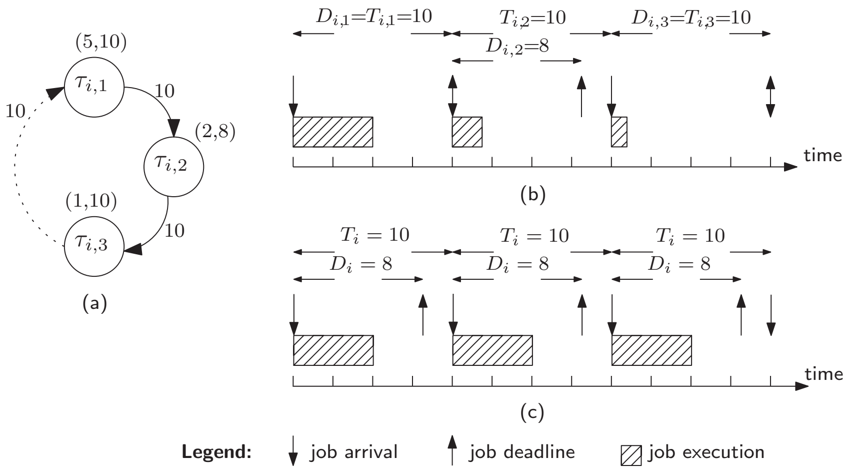

Fig. 4. A generalized multi-frame task [13] flattened into a sporadic task [9]. (a) represents the digraph of the task, (b) shows the corresponding time-line representation and (c) shows the time-line of a constructed sporadic task.

complexity of analyzing a set of sporadic tasks is much easier than analyzing a set of periodic tasks when self-suspension of tasks is allowed. Lifting the standard scheduling assumptions is beyond the scope of this work, however.

The model we will use and its performance

In this work we schedule a set of sporadic tasks upon a single preemptive processor. As we have seen just before, various sufficient conditions have been developed for this task model in order to determine whether a task set can meet its deadlines by a certain scheduling policy.

More general task models, i.e., digraph tasks, can be flattened in order to convert them into a sporadic task model. This flattening means that the sporadic task τ_i can be defined by the tightest $D_{i,k}$, the smallest $T_{i,k}$ and the largest $E_{i,k}$ of all the sub-tasks in the graph representations of a digraph task. Figure 4 shows an example of flattening the graph representation of a task into a sporadic task. Alternatively, each job in the digraph can be seen as a concurrent task, i.e., ignoring the mutual-exclusive execution intervals of those jobs. In both cases, the scheduling analysis on a flattened task system may become pessimistic.

Since we will use the sporadic task model, we recapitulate the exact scheduling analysis of a set of task that needs to execute on a single processor. These sufficient and necessary conditions are assumed to be well-understood by the reader and can be found back in the referred literature.

A set $\mathcal{T}$ of n deadline-constrained sporadic tasks can be scheduled under a fixed priority assignment upon one processor [31], if and only if,

$$\forall i \ : \ 1 \leq i \leq n \ : \ (\exists t \ : \ t \in \mathcal{S}_i \ : \ \mathrm{rbf}(i,t) \ \leq \ t)\,, \tag{1}$$

where

- $\mathrm{rbf}(i,t)$ defines the cumulative processor request bound of the tasks with a priority higher than, or equal to, τ_i in an arbitrary interval of length t, i.e,

$$\mathrm{rbf}(i,t) \quad = \quad \sum_{\pi_j \geq \pi_i} \left\lceil \frac{t}{T_j} \right\rceil E_j; \tag{2}$$

- $\mathcal{S}_i$ denotes a non-empty finite set of time points [31], i.e.,

$$\mathcal{S}_i \overset{\mathrm{def}}{=} \left\{ t = b{\cdot}\, T_a \;\middle|\; \pi_a > \pi_i;\ b \in \mathbb{N}^+;\ t \in (0, D_i] \right\} \cup \{D_i\}. \tag{3}$$

A set $\mathcal{T}$ of n deadline-constrained sporadic tasks can be scheduled with EDF upon one processor [10], if and only if

$$\forall t \ : \ t \in \mathcal{S} \ : \ \mathrm{dbf}(t) \quad \leq \quad t, \tag{4}$$

where

- $\mathrm{dbf}(t)$ defines the cumulative processor demand bound of the task set in an arbitrary interval of length t, i.e,

$$\mathrm{dbf}(t) \quad = \quad \sum_{1 \leq i \leq n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor E_i; \tag{5}$$

- $\mathcal{S}$ denotes a non-empty finite set of time points [32], i.e.,

$$\mathcal{S} \overset{\mathrm{def}}{=} \left\{ t = b{\cdot}\, T_i + D_i \;\middle|\; 1 \leq i \leq n;\ b \in \mathbb{N}^+;\ t \in (0, \mathrm{lcm}\{T_1, \ldots, T_n\}] \right\} \tag{6}$$

and the $\mathrm{lcm}\{T_1, \ldots, T_n\}$ denotes the least common multiple of the task periods.

The algorithmic complexity of verifying the scheduling conditions in (1) and in (4) is pseudo-polynomial.

2.2 Priority inversion of sporadic tasks

The scheduling of tasks discussed so far assumed fully preemptive task models, so that at each moment in time the highest priority job can be selected by the scheduler for execution on the processor. In practice, tasks may share more resources during (part of) their execution than just the processor. For example,

1) tasks may access shared memory to communicate with other tasks
   - via shared buffers and shared data structures; or
   - via memory-mapped devices.
2) tasks may use operating-system services, which requires
   - protection of in-kernel data structures using short non-preemptive critical sections;
   - mutual exclusion between the tasks that use shared device drivers and other services.
3) tasks may access the processor non-preemptively (i.e., so-called *pseudo-resources*) in order to
   - reduce the number of cache misses;
   - reduce pipeline flushes.

Access to shared resources is typically guarded by synchronization primitives provided by the real-time operating system, e.g., semaphores [33], in order to guarantee mutual exclusion. These resources are assumed to be accessed only in critical sections and are *serially reusable*.

A synchronization protocol arbitrates access to the class of resources that is referred to as *serially reusable, non-preemptive resources*. Baker [34] and Baruah [32] describe the meaning of these resources as follows. A job that begins executing on such a resource releases the resource only when it has completed its execution on the resource. Nevertheless, it is possible that a job accessing such a resource in a critical section is preempted within the critical section; however, it will not release the resource upon such preemption.

The classical semaphores do not provide suitable mechanisms for real-time arbitration of shared resources, because the use of semaphores may lead to so-called *unbounded priority-inversion times*. Figure 5 shows an example of this phenomenon. During priority inversion the highest-priority task τ_1 effectively runs at the lowest priority until that lowest priority task τ_n releases the resource R_ℓ that τ_1 wishes to access as well. The actual length of the time interval of priority inversion might not be unbounded; however, without a real-time protocol it is complicated to compute a bound on the priority-inversion duration. The *unbounded* durations of priority inversion reflect that each task must assume it has the lowest priority in the system and its execution is maximally delayed accordingly. This may easily lead to infeasible schedules.

The goal of real-time synchronization protocol is to bound priority-inversion times between tasks. Lampson et al. [35] already recognized the phenomenon of unbounded priority-inversion times and they suggest a solution to bound the durations of priority inversion. Their solution is currently known as the

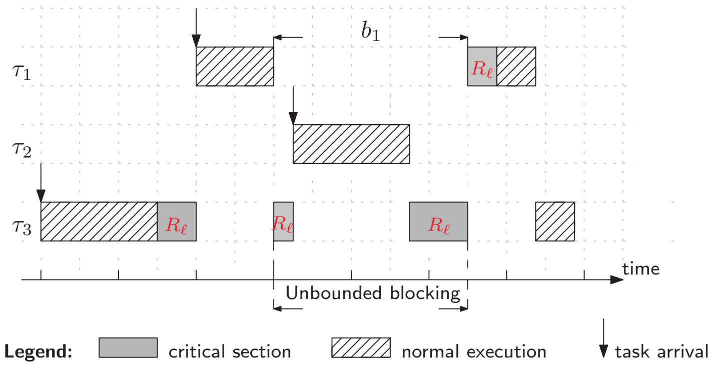

Fig. 5. Synchronization mechanisms, e.g., semaphores [33], which provide means to implement mutual-exclusive execution of tasks, can lead to so-called *unbounded blocking* of high-priority tasks. Real-time synchronization protocols have been invented to prevent this phenomenon.

immediate priority ceiling protocol (I-PCP) [36]. However, they did not describe the timing analysis of the tasks.

There are different ways to extend the scheduling analysis of task models with resource sharing. Firstly, we briefly recapitulate the traditional synchronization protocols. The tasks may access resources through these protocols at any arbitrary moment in time during their execution. Secondly, we look at tasks that consist of predefined series of critical sections (also called sub-jobs).

Real-time synchronization protocols

The key property of a real-time synchronization protocol is that the priority-inversion durations between tasks can be bounded based on just (*i*) the WCETs of the critical sections of all tasks and (*ii*) the relative priorities of the tasks that access the same resources. Thus, the WCETs of the entire tasks are irrelevant. For analysis purposes, we augment the sporadic-task model with the WCETs of the critical sections of tasks.

- $\mathcal{H}_i$ is a set WCETs of critical sections of a task τ_i. The cardinality of $\mathcal{H}_i$ is at most equal to the number of resources in the system, m.
- $h_{i\ell} \in \mathcal{H}_i$ is the WCET of a critical section of task τ_i to resource R_ℓ. A task may execute multiple critical sections accessing the same resource R_ℓ; the value of $h_{i\ell}$ is just the longest WCET among those critical sections.

To perform the timing analysis of tasks, many real-time synchronization protocols use the notion of a *resource ceiling*.

Definition 1: A resource ceiling rc_ℓ of resource R_ℓ is a statically computed value that indicates the highest priority task that wishes to access resource R_ℓ.

Under fixed-priority scheduling of tasks the priorities of the tasks can be used to define the resource ceiling. Under EDF scheduling of tasks, which is a dynamic-priority scheduler, the shortest *relative deadline* of the resource-sharing tasks defines the resource ceiling.

Some protocols also use the resource ceiling in order to implement the basic mutual-exclusion mechanism of a synchronization protocol. The most commonly used real-time protocols can be found in standard text books, for example, see [37]. We therefore give just a brief overview of their distinguishing characteristics, illustrated by Figure 6.

Priority-inheritance protocol [36]. The priority-inheritance protocol works as follows. If a higher-priority task blocks on a resource that has been locked by a lower-priority task, then the lower-priority task temporarily inherits the higher priority. As soon as the lower-priority task releases the resource, it falls back to its own priority. A task can inherit priorities transitively.

Spuri [38] has proposed the dynamic priority-inheritance protocol for EDF scheduling of tasks. Similarly to fixed-priority scheduling, a task inherits the absolute deadline of a blocked task until it releases the resource; thereafter, it resumes at its original deadline.

The priority-inheritance protocol uses the resource ceiling just for analysis purposes. A task can at most inherit a priority equal to the resource ceiling. Moreover, a higher-priority task can block at most once on each of its accessed resources within a synchronous busy interval. Formally, a task τ_i can experience a blocking duration of at most

$$b_i \quad = \sum_{1 \le \ell \le m} \max\left(0,\ \ \max\left\{h_{j\ell} \mid \mathrm{rc}_\ell \ge \pi_\mathrm{i} > \pi_\mathrm{j}\right\}\right). \tag{7}$$

The term in (7) can be further reduced by taking the number of lower priority tasks into account [36].

In an EDF-scheduled task system, only a task that has already started its execution and that has a deadline larger than a considered time interval of length t can block tasks with a deadline shorter than (or equal to) t, i.e.,

$$b(t) \quad = \sum_{1 \le \ell \le m} \max\left(0,\ \ \max\left\{h_{j\ell} \mid D_i \le t < D_j\right\}\right). \tag{8}$$

The phenomenon that a higher-priority task can block on a sequence of resources that are all locked by different lower-priority tasks is called *chained blocking* (see Figure 6). The chain of blocking durations results in a cumulative blocking defined by a sum of critical-section lengths, see (7) and (8), and deadlocks are possible.

Priority ceiling protocols [36]. The priority-ceiling protocol has been invented to overcome the penalties of chained blocking. It works as follows. During run-time a dynamically computed system ceiling, $\mathrm{sc}(\mathrm{t})$, is maintained,

$$\mathrm{sc}(\mathrm{t}) \quad = \quad \max\{\mathrm{rc}_\ell \mid \mathrm{R}_\ell \text{ is locked at time } \mathrm{t}\}\,. \tag{9}$$

If a task wishes to access a resource R_ℓ, the PCP grants access only if the priority of the task is higher than the current system ceiling. Furthermore, if a task blocks on a resource, the task that has locked the resource inherits the priority of the blocked task until it releases the resource (just like the PIP).

Since the resource ceiling represents the highest-priority task that may ever wish to access the same resource, the PCP avoids chained blocking and it avoids deadlocks [36]. A higher-priority task can therefore be blocked on at most one of its accessed resources within a synchronous busy interval. Formally, a task τ_i can experience a blocking duration of at most

$$b_i \quad = \quad \max\left(0,\ \max\{h_{j\ell} \mid 1 \le \ell \le m \wedge \mathrm{rc}_\ell \ge \pi_\mathrm{i} > \pi_\mathrm{j}\}\right). \tag{10}$$

Chen et al. [39] have extended the PCP for EDF scheduling of tasks. In an EDF-scheduled task system, only a task with a deadline larger than t that has already started its execution can block tasks with a deadline of at most t, i.e.,

$$b(t) \quad = \quad \max\left(0,\ \max\{h_{j\ell} \mid 1 \le \ell \le m \wedge D_i \le t < D_j\}\right). \tag{11}$$

Stack-based protocols [34], [36]. Similar to the default counting semaphores, the PIP and the PCP require waiting queues to track tasks that pend on a locked resource. When a task releases a resource, it comes with an event to notify blocked tasks. This may lead to an interleaved execution of high priority and low priority tasks, see Figure 6(a) and Figure 6(b). As a consequence, each task must store its execution state on a separate stack location. Stack-based synchronization protocols prevent interleaved execution of tasks. Contrary to the PIP and the PCP, the key property of stack-based protocols is that they have *non-blocking* lock and unlock operations.

Sha et al. [36] proposed a stack-based synchronization protocol which is widely applied in industrial systems. They proposed the *immediate PCP* (I-PCP) as a simpler alternative for the PCP. The I-PCP is, for example, part of the OSEK and POSIX operating-system standards[5]. Whereas the PCP waits with blocking of other tasks until one attempts to access a resource, the I-PCP immediately raises the priority of a task to the resource ceiling upon access of a resource. As soon as the resource is released, the task falls back to its previous priority.

The stack resource policy (SRP) [34] generalizes the I-PCP, i.e., it supports multi-unit resources and it supports dynamic priority scheduling using the EDF. Contrary to the I-PCP, the SRP changes the scheduler. That is, at each

5. The immediate priority ceiling protocol (I-PCP) is sometimes also referred to as the highest locker protocol (HLP). In the POSIX standard, the I-PCP is called Priority Protect Protocol (PPP).

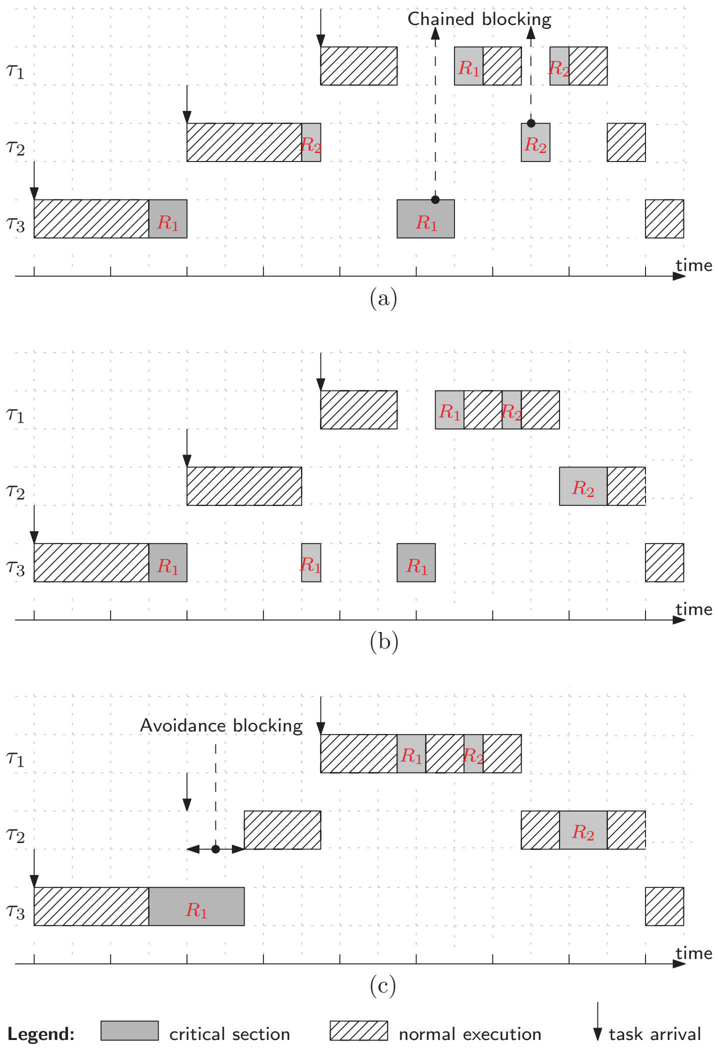

Fig. 6. Overview of different real-time synchronization protocols that bound the priority-inversion times between tasks. The PIP allows blocking of a higher-priority task by a chain of lower-priority tasks, but only once per accessed resource, see (a). The PCP, the I-PCP and the SRP avoid this phenomenon, see (b) and (c). The PCP blocks a higher-priority task at its attempt to access a resource when the system ceiling is higher than the task's priority, see (b). Contrary to the PCP, both the I-PCP and the SRP disallow any higher-priority task to start their execution when a resource with a ceiling higher than their priority is locked, see (c).

scheduling decision the highest priority task is dispatched that has the highest priority among all ready tasks and that has a higher priority than the current system ceiling. Under fixed-priority scheduling of tasks, the I-PCP and the SRP exhibit the same run-time behavior, see Figure 6(c). The same blocking term can be used for resource sharing arbitrated by the PCP, the I-PCP and the SRP (also with EDF). This blocking term has been defined in (10) and in (11).

Comparison of real-time synchronization protocols

We briefly compare the different real-time protocols. Further, we motivate why we prefer to use the SRP as a synchronization protocol in this work.

A nice property of the I-PCP and the SRP is that they prevent interleaved execution of low-priority tasks and high-priority tasks. Figure 6(c) illustrates this execution behavior, i.e., see the pyramid form of the job executions. These protocols therefore make it possible to share a single execution stack between all the tasks. This may considerably reduce the memory requirements of a task set [40]. Moreover, the I-PCP and the SRP do not need an event-queue infrastructure, because the waiting queues of tasks are contained in the ready queue (sorted by execution priority).

One reason for choosing PIP support in a real-time operating system, however, is that PIP can implement the basic functionality of mutual-exclusive execution between tasks without requiring any pre-computed resource ceilings (unlike the PCP, the I-PCP and the SRP). In the PIP the priority of the task that has locked the resource is dynamically raised to the priority of the task that is pending on a resource, which does not require knowledge of any resource ceiling. This property of the PIP is called *transparency*, i.e., the PIP does not require new kernel primitives compared to the default semaphores [33]. Despite this nice property, the analysis of tasks arbitrated by the PIP uses the notion of a resource ceiling [36] (just like the PCP, the I-PCP and the SRP do).

Some operating systems even do away with the transparency property of the PIP in order to reduce implementation costs. For example, μC/OS-II only supports a single task on each priority level, independent of the execution state of the task, because a priority is also used as a task identifier. A priority level is assigned to each resource on creation of the resource. The priority level is used to raise the priority of a task when it blocks a higher-priority task, which is named *priority calling* in the μC/OS-II terminology. The priority calling protocol deployed by μC/OS-II is well documented [41] and it implements the PIP non-transparently[6]. Because of the assignment of a unique priority to each resource, the transparent character of PIP is lost.

6. Many other operating-system vendors refuse to state clearly which synchronization protocol they have implemented [42]. For example, Zöbel et al. [42] have shown that many attempts to identify the truly implemented protocol fail.

The disadvantage of the I-PCP and the SRP is that these protocols block any task with a lower priority than the resource ceiling, even if none of those tasks pend on the same resource. This phenomenon is called *avoidance blocking*, see Figure 6(c). Avoidance blocking does not affect the computation of the blocking term compared to the PCP. The reason is as follows. A middle-priority task may get blocked by the PCP as soon as a high-priority task blocks (causing a priority-inheritance of the lowest priority task). With the I-PCP blocking cannot happen during the execution of a task; blocking can only happen prior to the execution of a task. Similar to the PCP, however, a task arbitrated by the I-PCP can be blocked at most once by a lower priority task in a synchronous busy interval. We conclude that the same blocking term can be used for resource sharing arbitrated by the I-PCP and the SRP as defined in (10) and in (11).

Advantages of the I-PCP and the SRP are their ease of implementation and their deadlock-avoidance property. Moreover, the worst-case blocking of tasks arbitrated by the PCP, the I-PCP or the SRP, i.e., see (10) and (11), is superior compared to the worst-case blocking of tasks arbitrated by the PIP, i.e., see (7) and (8). The PIP is therefore not very suitable for arbitrating hard real-time tasks. This is confirmed by Baruah [32], who has shown that the SRP+EDF is an optimal scheduling policy for hard real-time sporadic tasks among all work-conserving scheduling policy. Since this work considers the scheduling of hard real-time tasks, without further loss of generality, we focus on resource arbitration using the SRP.

Performance of limited-preemptive schedulers

The analysis of resource-sharing tasks typically assumes a *preemptive task model*. This means that a blocking term is included in the analysis of the schedule as a worst-case delay for the start of the execution of tasks. The blocking task may finish its work earlier, however, when it delays the execution of higher priority tasks beyond its own completion. Such accelerated execution of a task is not taken into account in the preemptive task model. So-called *limited-preemptive schedulers* do take into account these delays of higher-priority tasks.

Bertogna and Baruah [43] have presented an overview of limited-preemptive EDF scheduling of sporadic tasks. Buttazzo et al. [44] have surveyed limited-preemptive scheduling of sporadic tasks with fixed priorities. We briefly recapitulate the main ideas behind different task models.

Preemption-thresholds scheduling (PTS) [45]. With PTS, each task τ_i has a *preemption threshold* θ_i at least equal to its priority. A job of task τ_i competes for the processor at its priority when it is ready to execute but it has not yet started to execute. At run-time, when τ_i first starts to execute, its active priority is set to its preemption-threshold θ_i. When τ_i is executing, it can only be preempted by tasks with a priority higher than θ_i. PTS can specialize to both fully preemptive scheduling and non-preemptive scheduling of tasks.

Gai et al. [40] have shown that PTS can be unified with the SRP, where a task τ_i accesses a so-called pseudo-resource R_ρ as soon as it starts executing and it releases R_ρ upon completion. The corresponding worst-case critical-section length is $h_i = E_i$ and the resource ceiling is defined by $rc_\rho = \theta_i$. Then, the blocking times experienced by tasks can be computed according to the SRP as well, i.e., see (10) and (11). Given a task set with fixed priority assignments, Wang and Saksena [45] derived algorithms for selecting the preemption thresholds.

Scheduling with deferred preemptions (DPS) [16], [46]. In DPS, each job of task τ_i consists of a sequence of m_i sub-jobs[7], where $m_i \geq 1$. The a-th sub-job of τ_i is denoted by $\tau_{i,a}$ and characterized by a WCET, $E_{i,a}$, where $E_i = \sum_{a=1}^{m_i} E_{i,a}$. Sub-jobs are non-preemptive. Hence, a task can only be preempted between sub-jobs, i.e., at the so-called *preemption points*. DPS specializes to non-preemptive scheduling when $\forall_{1 \leq i \leq n} m_i = 1$.

The number of non-preemptive sub-jobs, m_i, is typically assumed to be static [16]. As an example, fixed-priority DPS, also referred to as FPDS [47] or co-operative scheduling [16], places preemption points statically [48]. However, Bertogna and Baruah [43] have presented a novel DPS scheme for EDF, where a job is dynamically split into non-preemptive sub-jobs without compromising EDF's optimality. Although the total number of preemptions in a fully-preemptive EDF schedule is bounded from above by the number of jobs being scheduled, there is no bound on the number of preemptions of an individual job. The DPS-EDF policy of Bertogna and Baruah [43] gives bounds also on the number of preemptions for the jobs of an individual task.

Scheduling with preemption thresholds for sub-jobs (PTS$^+$) [49], [50]. With PTS$^+$, each task τ_i consists of a sequence of m_i sub-jobs (similar to DPS). In addition, each sub-job $\tau_{i,a}$ has a preemption threshold $\theta_{i,a}$. When a sub-job $\tau_{i,a}$ is executing, it can only be preempted by tasks with a priority higher than its preemption threshold $\theta_{i,a}$. The preemption threshold $\theta_{i,a}$ therefore allows the threshold for preemption to be raised for the duration of sub-job $\tau_{i,a}$.

When $m_i = 1$, a task τ_i has no preemption points. That is, PTS$^+$ specializes to PTS when $\forall_{1 \leq i \leq n} m_i = 1$. Furthermore, it specializes to DPS when all sub-jobs are non-preemptive.

In line with the result of Gai et al. [40], Yao et al. [50] have shown that PTS$^+$ can be implemented using the SRP, where each sub-job accesses a pseudo-resource R_ρ as soon as it starts executing and it releases R_ρ upon completion. The corresponding worst-case critical-section length is $h_i = E_{i,a}$ and the resource

7. *Sub-jobs* in a limited-preemptive task model are different from *sub-tasks* in a preemptive digraph task model (see Figure 3). A digraph task represents possible sequences of jobs, i.e., sub-tasks, generated by that task, including deadlines and minimal inter-arrivals. The sub-jobs of a limited-preemptive task represent contiguous work to be done by a single job of that task.

ceiling is defined by $rc_\rho = \theta_{i,a}$. Then, the blocking times experienced by tasks can be computed according to the SRP as well, i.e., see (10) and (11).

PTS$^+$ is also a form of cooperative processor allocation [16] between the tasks and the scheduler. Sub-jobs are separated by *preemption points* (just like with DPS) and at these preemption points, a task relinquishes its control over any pseudo-resource on which it wishes to execute. Since the SRP+EDF is an optimal uni-processor scheduling policy, limited preemptive scheduling models cannot lead to feasible schedules that were infeasible with the SRP+EDF [46]. By scheduling tasks according to SRP+EDF, it is irrelevant how often each task accesses a shared resource and it is irrelevant when a task starts with accessing a shared resource during its execution. Thus, these two execution characteristics can be left out of the task model.

Contrary to EDF, PTS$^+$ under fixed-priority assignments of task – which we refer to as FPTS$^+$ (see Paper A) – can actually take advantage of the alignment of sub-jobs (just like FPDS, see [47] and [48]). The intuition behind this improvement is as follows. If a task finishes its job with a critical section, e.g., writing its output to a shared buffer, then this job may block any higher priority task until it has already finished its own execution. However, if the job of the resource-accessing task would not finish immediately upon completion of the critical section, then the higher priority tasks would delay the completion of that job after the critical section anyway. Thus, delaying the higher-priority tasks until completion of a lower-priority task may improve the schedulability of a task set. The following example illustrates this.

Example 1: Consider a system with three tasks $\tau_i = (T_i, E_i, D_i)$: $\tau_1 = (7, 1, 2)$, $\tau_2 = (15, 8, 15)$ and $\tau_3 = (26, 6, 26)$, where τ_1 is assigned highest priority and τ_3 is assigned the lowest priority. Figure 7(a) shows (part of) the worst-case schedule of these fully preemptive tasks under a synchronous release.

Figure 7(b) and Figure 7(c) show that the schedulability of tasks can be improved for restrictive forms of resource sharing. In these figures, only the critical section that completes the execution of a job is shaded; other critical sections may just delay preemptions of higher priority tasks, but those do not have the potential to accelerate the execution of a job. Because task τ_2 is allowed to execute for a duration of 1 time unit non-preemptively, the WCET for writing its output non-preemptively to a buffer is $E_{2,2} = 1$ time units. Using our analysis in Paper A, task τ_2 can tolerate a blocking of at most 5 time units. For task τ_3 we can either choose a DPS scheme with $h_{3,1} = 1$ time units or we can choose a PTS$^+$ scheme with $E_{3,2} = 5$ time units and a preemption threshold $\theta_{3,2} = \pi_2$. For these two cases, our analysis in Paper A finds a worst-case response time of $\mathtt{WR}_3 = 26$ time units and $\mathtt{WR}_3 = 17$ time units, respectively. In this example, PTS$^+$ generates a significant amount of slack in the system.

Hence, a lower preemption threshold may allow for a larger sub-job length,

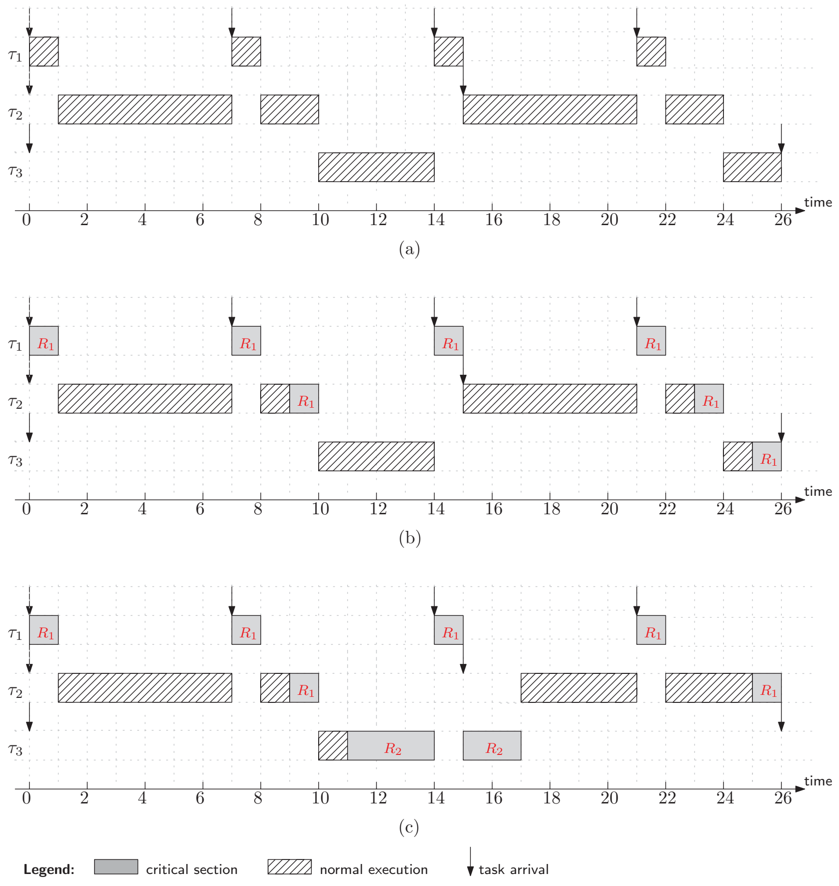

Fig. 7. Fixed-priority scheduling of tasks can take advantage of an appropriate alignment of limited-preemptive critical sections. Figure (a) illustrates a fully preemptive schedule (without resource sharing) of the task set given in Example 1. Figure (b) shows that tasks can execute a sequence of critical sections of (at most) 1 time unit non-preemptively (DPS), without violating any deadline constraints. Finally, Figure (c) shows that a lower priority task can further accelerate its execution by delaying just a selection of higher priority tasks (PTS$^+$).

while a higher threshold may reduce the sub-job length due to little tolerated blocking by (one of) the blocked tasks. The last sub-job of a task can therefore be seen as a critical section with trade-offs in preemption level and WCET.

Why does this scheduling improvement not apply to EDF? Contrary to fixed-priority scheduling, under EDF a job can preempt upon its release, only if it has an earlier absolute deadline than the currently executing job. For any arbitrary release, the relative deadline of the task determines whether or not a certain amount of blocking and interference can be allowed. Within the deadline, the interference is determined only by tighter absolute deadlines of other tasks. This is where fixed-priority scheduling looses scheduling performance: a lower-priority task is delayed by a higher-priority task, no matter their absolute deadlines. Limited-preemptive scheduling techniques can alleviate this problem and they can therefore improve the achievable utilization of fixed-priority schedulers significantly (see Paper A). Nevertheless, the performance cannot approach EDF (as it was conjectured by Bertogna et al. [48]). Even with limited-preemptive scheduling, the weakness of fixed-priority assignments to tasks remains that the preemption thresholds only increase the urgency of executing a job after it has already started executing. With EDF the urgency of executing a pending job also increases prior to the start of its execution.

Complexity of analyzing a schedule

We again use the sporadic task model and we recapitulate the scheduling analysis of a set of resource-sharing tasks that need to execute on a single processor. These conditions are assumed to be well-understood by the reader and can be found back in the referred literature.

A set $\mathcal{T}$ of n deadline-constrained sporadic tasks can be scheduled with SRP+EDF upon one processor [32], if and only if

$$\forall t \ : \ t \in \mathcal{S} \ : \ b(t) + \mathrm{dbf}(t) \quad \leq \quad t, \tag{12}$$

where

- $b(t)$ is the maximum priority-inversion time within an interval of length t due to local resource sharing, as defined in (11);
- the $\mathrm{dbf}(t)$ is defined in (5); and
- $\mathcal{S}$ denotes a non-empty finite set of time points [32], see (6).

A set $\mathcal{T}$ of n deadline-constrained sporadic tasks can be scheduled with the SRP and a fixed priority assignment upon one processor [34], if

$$\forall i \ : \ 1 \leq i \leq n \ : \ (\exists t \ : \ t \in \mathcal{S}_i \ : \ b_i + \mathrm{rbf}(i,t) \ \leq \ t)\,, \tag{13}$$

where

- b_i is the maximum priority-inversion time experienced by task τ_i due to local resource sharing, as defined in (10);
- the $\mathrm{rbf}(i,t)$ is defined in (2); and
- $\mathcal{S}_i$ denotes a non-empty finite set of time points [31], see (3).

The scheduling condition in (12) is exact [32], i.e., both sufficient and necessary, for scheduling the task set with resource constraints successfully by the SRP+EDF upon one processor. The scheduling condition in (13) is just sufficient for scheduling the task set successfully with fixed-priority and the SRP upon one processor [34], because it only considers the negative impact of blocking. For other synchronization protocols than the SRP, the blocking terms in (12) and in (13) must be changed accordingly (as described earlier in this section). The maximum amount of blocking that a task can tolerate without missing its deadline is called the *blocking tolerance* [51]. The algorithmic complexity of verifying the scheduling conditions in (12) and in (13) is pseudo-polynomial.

Paper A considers exact scheduling analysis for limited-preemptive schedulers under fixed-priority assignments of tasks. The key idea behind this analysis is as follows. Assuming a task with a sequence of sub-jobs, we compute the worst-case finishing time of a task until the final sub-job. Then, we take the worst-case finishing time as the start time of the last sub-job while taking into account the limitations of the preemptions incurred by its preemption threshold. Whereas the condition in (13) considers the entire job of a task holistically, our novel analysis in Paper A checks a vector of time points for each step in the analysis. The algorithmic complexity of analyzing a limited-preemptive schedule remains pseudo-polynomial (similar to the algorithmic complexity in (13) for a preemptive task model).

3 Hierarchical composition of real-time models

In this section, we present the basic structural design that underlies the systems considered in this work. Viewing a system as a hierarchy of components is useful, because it describes many different system architectures and it helps to focus on the similarities and differences between systems.

Building a system must follow closely the thoughts of a system designer, which calls for well-defined design schemes. We will therefore look at convenient units of composition, how to model dependencies between them and how to implement those compositional units and their dependencies.

3.1 The design we hold to

An hierarchical scheduling framework (HSF) – as proposed by Deng and Liu [52] – is implemented upon a single processor and contains components at each level in the scheduling hierarchy. Each component has a parent (except the root of the scheduling tree) and each component abstracts the workload generated by its child components deeper in the hierarchy, see Figure 8. At the leafs of the scheduling tree the hard real-time tasks reside; these tasks implement the real work to be executed.

The entities in our HSF, as illustrated in Figure 8, are bound to contracts. Each component receives one incoming supply of processor resources, i.e., depicted by one incoming bold arrow. The component therefore virtually executes upon a single processor and it can be unaware of its position in the scheduling hierarchy[8]. A local scheduler allocates the supplied resources to its children, i.e., the local tasks. Each component may select its own scheduling policy.

The tasks located in arbitrary components may require other resources than just the processor. A resource that is used in more than one component is denoted as a shared *global resource*. A resource that is only shared by tasks within a single component is a shared *local resource*. Any access to a resource is assumed to be arbitrated by a synchronization protocol.

Traditional synchronization protocols such as the PCP [36] and the SRP [34] can be used for local resource sharing in HSFs [52], as is formally proven in [54]. However, when a task accesses a global shared resource, one needs to consider the priority inversion between components as well as the local priority inversion between the tasks within a component. A global resource is therefore accompanied by a logical stub locally. It is this stub that is accessed by the local tasks and an a-synchronous upcall is made to the parent component upon an access of that stub. The same parent that also supplies the processor

8. In practice, the component cannot be entirely unaware of the scheduling level [53]. The tighter the timing constraints of a component are, the less scheduling overhead a component allows and, thus, the higher the component must be positioned in the scheduling hierarchy.

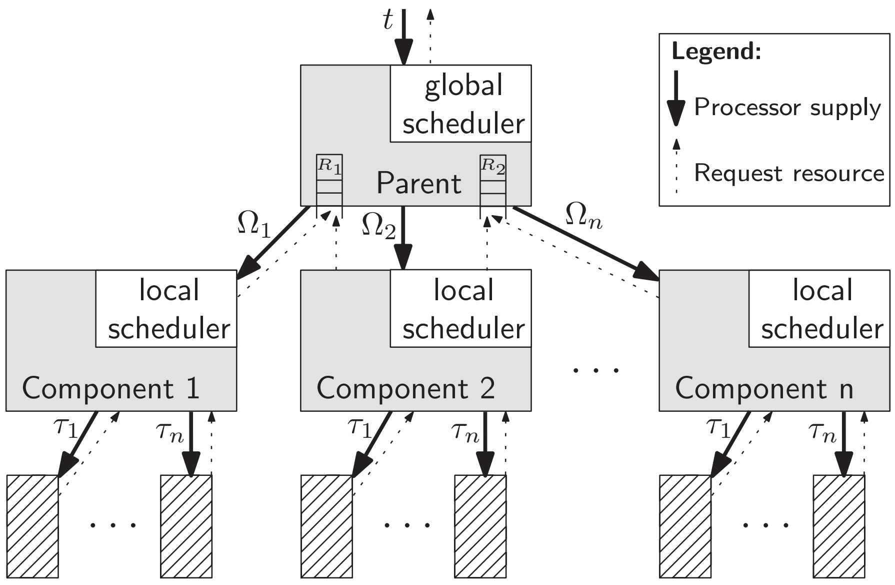

Fig. 8. A parent component implements a global scheduler to allocate a share of the processor and a share of other non-preemptive resources, e.g., R_1 and R_2, to each of its child components, $C_1 \ldots C_n$. Each child component, C_s, comprising a set of tasks, $\tau_1 \ldots \tau_n$, and a local scheduler, receives those shares that are specified by its interface Ω_s. Tasks, located in arbitrary components, may share resources.

resources is responsible for arbitrating access to the real global resource (see the dotted arrows in Figure 8). We will later consider various modifications on the traditional protocols in [36] and [34] for the implementation of such global synchronization protocols.

Figure 8 is rich enough to model multi-level HSFs. We only consider interaction with other components through well-defined interfaces. Such global interaction between tasks located in different components is always routed via the earliest-common parent component. Without further loss of generality, we therefore focus on two-level HSFs (just like Deng and Liu [52] did).

In the remainder of this section, we explain our representation and our corresponding interpretation of the resources supplied to an individual component in an HSF. We first look at resource-supply models for independent components, i.e., components that only share the processor and do not share any other resources. Next, we augment those resource-supply models with support for sharing non-preemptive resources between components.

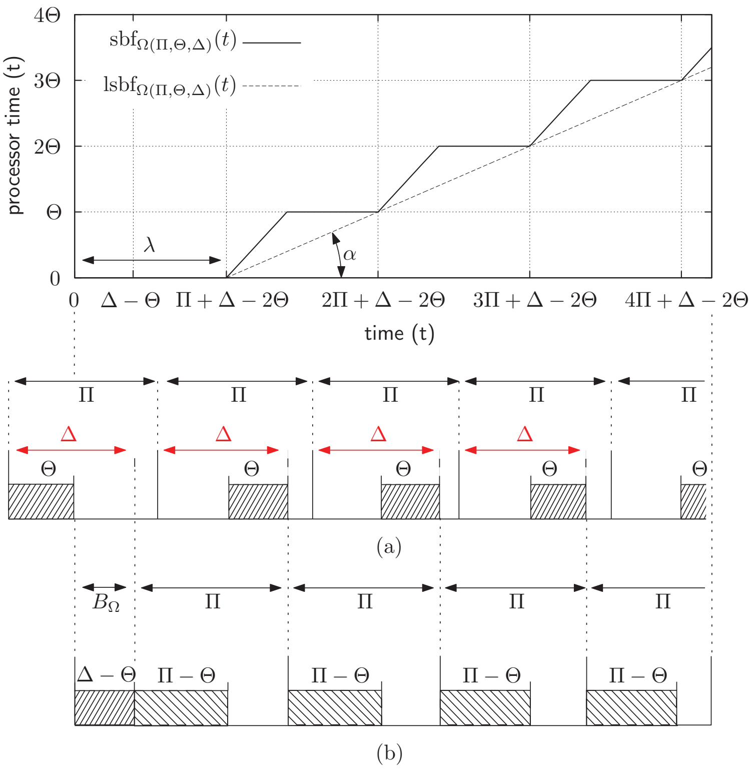

Fig. 9. The worst-case periodic resource supply to a component, $\mathrm{sbf}_\Omega(t)$, in an arbitrary time interval of length t, and its corresponding linear approximation, $\mathrm{lsbf}_\Omega(t)$, according to the EDP model $\Omega(\Pi,\Theta,\Delta)$.

Processor resource-supply models for independent components

Those parameters that define a worst-case processor supply, contemplating that timing constraints of a task set are satisfied, together form a so-called *real-time interface*. This interface – sometimes also referred to as a *supply contract* [55] – abstracts the timing constraints of tasks into a single real-time constraint.

Explicit-deadline periodic (EDP) resource model [56]. An EDP resource $\Omega(\Pi,\Theta,\Delta)$ supplies Θ time units of processor time every period of Π time units and Θ is provisioned no later than Δ time units from the start of the period Π (where $\Delta \le \Pi$). A component that receives its processor resources according to the EDP resource-supply model interferes with other components in the system as if it was as a single deadline-constrained periodic task.

The *supply bound function*, $\mathrm{sbf}_{\Omega(\Pi,\Theta,\Delta)}(t)$, returns the minimum processor supply for any sliding interval of length t, i.e.,

$$\mathrm{sbf}_\Omega(t) = \max \left\{ \begin{array}{l} 0, \\ \left(h_{(\Omega,t)} - 1\right)\Theta, \\ t - \left(h_{(\Omega,t)} + 1\right)(\Pi - \Theta) + (\Pi - \Delta) \end{array} \right\}, \tag{14}$$

where $h_{(\Omega,t)} = \left\lceil \frac{t-(\Delta-\Theta)}{\Pi} \right\rceil$.

The $\mathrm{sbf}_\Omega(t)$ is illustrated in Figure 9. The first clause of the $\max$-term prevents that supply bound becomes negative. The second and the third clause represent alternative views on the available processor time to the component:

1) In an interval of length t, the component receives *k times* Θ time units; see Figure 9(a). The component starts requesting for processor time when Θ has just been delivered, so that it has to wait for the longest possible duration until processor time is supplied.
2) In an interval of length t, the component receives *t minus the unavailable processor time*; see Figure 9(b). The unavailable processor time is characterized as follows: firstly, the delivery of processor resources is blocked for a duration of $B_\Omega = \Delta - \Theta$ time units. Then, immediately a periodic task, $\tau_\Omega = (\Pi, \Pi - \Theta, \Pi - \Theta)$ is released which periodically occupies the processor for $\Pi - \Theta$ time units.

The delivery of Θ runs at most one period Π late with respect to $t = 0$; similarly, the unavailable processor time runs at most one period Π ahead.

The notion of the unavailable processor time, captured by the last clause of the sbf_Ω, is interesting, because it illustrates the freedom of composition of the component. A lower priority component may block the component for at most $B_\Omega = \Delta - \Theta$ time units and, next, all the higher-priority (or equal-priority) components may at most consume $\Pi - \Theta$ time units. This is not the only interpretation, because there are other alignments of the worst-case blocking B_Ω and the worst-case interference τ_Ω that fit the sbf_Ω as well.

Time-division multiplexing. A time-division-multiplexed (TDM) resource $\Upsilon(\Pi, \Theta)$ is defined by:

$$\Upsilon(\Pi, \Theta) \overset{\mathrm{def}}{=} \Omega(\Pi, \Theta, \Theta). \tag{15}$$

An EDP resource models a fluctuating allocation time of the budget Θ and the degree of these fluctuations in two subsequent periods Π is bounded by deadline Δ. Contrary, a TDM resource supplies its budget Θ at the same time within each period Π, without fluctuations. The absence of such fluctuations may lead to tighter schedulability analysis. Easwaran et al. [56] refer to the TDM model as a bandwidth-optimal EDP-model.

Although the $B_\Omega = 0$ of a component, it is important to realize that TDM still allows resource sharing between components. The scheduling delays of components with equal priority are already taken into account as interference, see the definitions in (11) and in (10) of the blocking term. In other words, any extra processor bandwidth that is required for just the purpose of resource sharing between TDM-scheduled components must be allocated in and subtracted from the maximum allowable interference $\Pi - \Theta$.

Periodic resource model [57]. The periodic resource model $\Gamma(\Pi, \Theta)$ is a specialization of the EDP resource Ω, i.e., a periodic resource $\Gamma(\Pi, \Theta)$ has an implicit deadline for the allocation of budget Θ. More formally,

$$\Gamma(\Pi, \Theta) \overset{\text{def}}{=} \Omega(\Pi, \Theta, \Pi). \tag{16}$$

The resources supplied by $\Gamma(\Pi, \Theta)$ within two subsequent periods Π are subject to a maximum degree of freedom bounded by just the component period Π.

Bounded-delay model [58]. The bounded-delay model (α, λ) differs from the previous resource-supply models, because it does not necessarily require resources to be delivered periodically. Instead, it promises a continuous utilization, α, of the processor after an initial service delay of at most λ time units. The *supply bound function*, $\text{sbf}_{(\alpha,\lambda)}(t)$, returns the minimum processor supply for any sliding interval of length t, i.e.,

$$\text{sbf}_{(\alpha,\lambda)}(t) = \alpha\,(t - \lambda)\,. \tag{17}$$

This model is very general and it is difficult to select the parameters α and λ efficiently [59], [5]. A large service delay, λ, may need to be compensated with a large processor utilization, α, and vice versa. However, a component that meets its deadlines with just a small processor fraction may still leave little space for other components if it only allows a small service delay, λ. One way of limiting the search space for appropriate candidates of α and λ is putting extra requirements on the sizes of allocated processor quanta [58].

For example, the bounded-delay model can be used to estimate the resource supply of EDP resources linearly, i.e., as follows:

$$\lambda = \max\{t \mid \text{sbf}_\Omega(t) = 0\} = \Pi + \Delta - 2\Theta \quad \text{and} \quad \alpha = \frac{\Theta}{\Pi}. \tag{18}$$

Filling in these values of λ and α in the $\text{sbf}_{(\alpha,\lambda)}(t)$ defines the linear lower-bound of the supply bound function, i.e., the so-called $\text{lsbf}_\Omega(t)$ [57],

$$\text{lsbf}_\Omega(t) = \frac{\Theta}{\Pi}\,(t - (\Pi + \Delta - 2\Theta))\,. \tag{19}$$

On the one hand, the parameters Π and Δ are more suitable for selection during the component's design phase [60] and an appropriate selection of periodic parameters may give tighter scheduling bounds than the conservative approximation in (19) does. Figure 9 shows the $\text{sbf}_{\Omega(\Pi,\Theta,\Delta)}(t)$ of an EDP resource $\Omega(\Pi, \Theta, \Delta)$ and its corresponding bounded-delay approximation, $\text{lsbf}_{\Omega(\Pi,\Theta,\Delta)}(t)$.

By choosing the value of Δ appropriately, the definition of the $\mathrm{lsbf}_{\Omega}(t)$ can also be instantiated to approximate the supply of a periodic resource $\Gamma(\Pi, \Theta)$ or to approximate the supply of a TDM resource $\Upsilon(\Pi, \Theta)$.

On the other hand, the main advantage of the bounded-delay resource-supply model – also referred to as a latency-rate ($\mathcal{LR}$) server [61] – is that it is easy to aggregate series of interface parameters λ and α. This enables the computation of tight upper bounds on end-to-end delays and buffer requirements in a heterogeneous network of servers that may support different scheduling policies and different task models [61]. With the EDP resource-supply model, as well as with its derivatives, the aggregation of the interface parameters is hard, because it requires an alignment of the synchronous busy interval of local tasks and the period of the resource supply [62].

Other resource-supply models. There are alternative formal frameworks, e.g., real-time calculus [63], for characterizing the resource supply to a component in an interface. Unlike the resource-supply models in [56], [58], [57] and [63], various models, e.g., [64], [65], [66], [67] and [68], deviate from the principle of independent analysis of a component as they make assumptions, amongst others, on the relative priorities of other components in the system. These alternative models tighten the processor-supply bound of the serviced tasks. Unfortunately, the complexity of the analysis also increases for these models.

The approaches in [64], [65], [66] and [67] analyze the local and global schedulability of a system holistically. These works all consider two-level fixed-priority scheduling. Wandeler and Thiele [67] compute tight supply bounds by taking into account the interference of other components in the real-time calculus framework of [63]. Similarly, [64], [65] and [66] have extended the more traditional response-time analysis for this purpose.

In order to tighten the analysis of two-level EDF-scheduled systems, more advanced techniques are possible to shape the arrival of tasks exactly according to a predefined, arbitrary $\mathrm{dbf}(t)$ function and to enforce temporal isolation between components accordingly, for example, see [68].

Wang et al. [55] explicitly translate precedence constraints of local tasks to constraints on the resource supply of a component. These additional constraints on the interface inherently limit the composition, since independent components may have conflicting constraints. Instead, we will propose a virtual-time scheduler (Paper B) which makes it possible to leave precedence constraints implicit and to exclude them from the component's interface. In this entire work, however, we shall refrain from assumptions in the local analysis of a component on the global scheduling policies, including the exact timing of processor-time delivery. Our aim is to compose systems of components that can be developed and analyzed independently of other components, so that these components can be re-used in different systems and frameworks.

Augmenting resource-supply models for resource sharing

We take the EDP resource-supply model to specify the processor time allocated to a component. A budget Θ_s, specified by the interface Ω_s of a certain component C_s, defines the quantum of the processor that is required for the task set to meet the deadline constraints of the local tasks. In addition, tasks located in arbitrary components may require execution on the same non-preemptive resource. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of its budget, excessive blocking periods can occur [69] which may hamper the correct timeliness of other components. To prevent such budget depletion during global resource access, several synchronization protocols have been proposed in literature.

Although a global synchronization protocol must prevent budget depletion during global resource access, in order to do so, it may need to deliver processor resources differently or additionally. To be able to account the scheduling penalties that come with these changes in the processor supply, we need to augment the interface of a component with the amount of resources that this component requires on each non-preemptive resource. The set of these so-called resource holding times [70] – denoted by $\mathcal{X}_s$ – defines the amount of execution time on global resources that a component receives for an access to a resource. In other words, if component C_s is granted access to resource R_ℓ, it receives $X_{s\ell} \in \mathcal{X}_s$ time units of execution time on resource R_ℓ prior to deadline Π_s.

The value of the resource holding time of a component to a resource, i.e., $X_{s\ell}$, represents the largest amount of resources supplied to component C_s from the access until the release of a task τ_i to resource R_ℓ. It includes the cumulative processor requests of tasks within the same component C_s that can preempt τ_i while it is holding resource R_ℓ, see Figure 10.

The way of computing the resource holding times of a component and its tasks may depend on the local synchronization protocol and the local scheduling policy, for example, see the method under local SRP+fixed-priority scheduling by Bertogna et al. [70] and the method for local SRP+EDF by Bertogna et al. [71]. Moreover, various system assumptions, such as the global synchronization protocol, may influence the way of computing resource holding times, for example, see the methods developed for the global synchronization protocols BROE [72] and H-SRP [73]. We proposed a method (see Paper C) to unify the way of computing the resource holding times for a class of components.

To summarize, the *augmented* EDP resource model $\Omega = (\Pi, \Theta, \Delta, \mathcal{X})$

- gives Θ time units of processor time every period of Π time units and Θ is provisioned no later than Δ time units from the start of the period Π; and
- gives $X_\ell \in \mathcal{X}$ time units of processor time, provisioned no later than Π time units from the start of the period Π in which access to R_ℓ is granted.

The second item may call for the delivery of more processor time within a

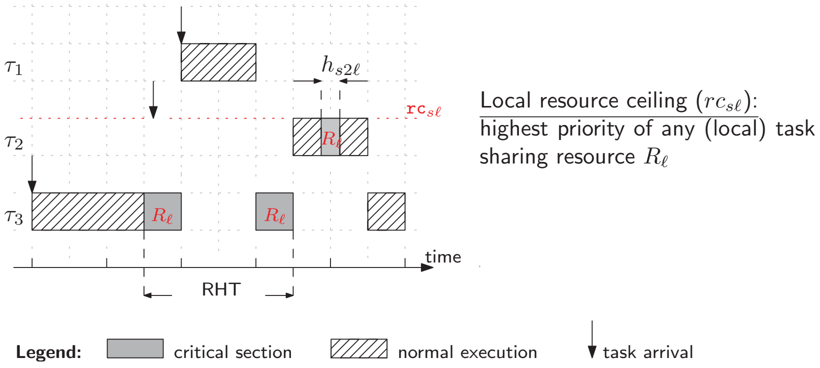

Fig. 10. The resource holding time (RHT) represents the cumulatively consumed processor time by any task of a component while one task holds a resource.

period Π than just Θ. Such a delivery of the extra processor time is necessary to complete resource access, but it may be delivered beyond deadline Δ. Only Θ must be delivered prior to Δ in order to satisfy deadline constraints of tasks. Any extra delivery of processor time guarantees that access to a resource R_ℓ is completed by the same job of a component as the job that initiated the access, just like the jobs of the task in Figure 10. As motivated further in Section 3.5, we use a deadline equal to period Π for this extra delivery.

3.2 Make tasks elementary units of composition

The task is a convenient unit to build real-time systems; the schedulability analysis of tasks is widely studied and is well understood. For this reason the task is also an appealing candidate to compose components hierarchically.

If the degree of complexity of a function, e.g., measured in terms of concurrency, is limited, then it is unnecessary to subdivide a function into different tasks. Thus, a simple function is written in a single task. A more complex function may require division into a set of tasks, i.e., together forming a component. A component is just an artefact of analysis, although it may serve also as a packaging unit for shipping a complex function to system builders. Each component is meant to be part of a larger system.

Conversely, a system can be decomposed into components with relative priorities. Regehr et al. [53] addressed the real-time analysis of restricted scheduling hierarchies. They proposed a method to flatten the scheduling hierarchy into a form where it can be analyzed using traditional real-time analysis. That is, they transform the scheduling hierarchy into a fixed-priority-scheduled task system and they use the preemption thresholds of Wang and

Saksena [45] to model mutual-exclusive execution between tasks. Gonzáles Harbour et al. [74] analyze a flattened system which may have groups of EDF-scheduled tasks within a larger system with tasks that have fixed priorities.

There are two main disadvantages of analyzing a flattened system, as is done by [53] and [74]. Firstly, it is unsuitable for the composition of independently developed components, because the tasks in the entire system are holistically analyzed. Hence, the entire system must be decomposed again and must be re-analyzed after each change, for example, after adding or removing a component. This is impractical, and sometimes even impossible, for open systems composed from components of multiple vendors.

Secondly, the analysis on a flattened system, i.e., composed of independently developed components, may be pessimistic. The reason is that the synchronous busy interval of a group of tasks (forming a component in the hierarchy) can be too long; even when the component's utilization is arbitrary small, a component may leave little space for timely execution of other components. Without mechanisms to control the workload generated by the tasks, the traditional real-time analysis is inconvenient for the composition of components. Fortunately, there exist composition techniques to overcome these issues.

The resource-supply contract of a component, i.e., the interface, specifies the periodic requirements of a component in order to meet its tasks' deadlines. This interface is determined based on a worst-case upper bound of the workload generated by jobs of tasks within a component. However, we need additional mechanisms to enforce periodic execution to a component during run time. The real-time systems community has therefore proposed *processor resource reservations* [75] to re-shape the workload generated by the tasks of a component into (sporadically) periodic chunks of execution. The implementation of processor reservations requires mechanisms for scheduling the reserved quanta, enforcing their availability and monitoring consumed reserves by the component.

The access of a component in the HSF to its processor reservations is mediated through a *server*. A server defines a policy to implement each of the three mechanisms of a reservation, which leads to (periodic) behavior of the component satisfying its interface. Servers are originally meant to handle a-periodic tasks with unknown arrival curves and to force periodic behavior to those a-periodic tasks. The other hard real-time recurring tasks, executing along with the server on the same processor, can therefore be analyzed even in the presence of unpredictable (a-periodic) tasks handled by that server. Similarly, a component in an HSF, serviced by a server, can be treated as if it was a single task on itself and existing timing analysis for tasks can be re-used for validating whether or not it is feasible to admit a component on a shared processor.

In addition to the three mechanisms inherited from a reservation, a server implements a policy to distribute the reserved processor resources to its serviced tasks. The most straightforward server policies are TDM and periodic polling [76]. Several server policies aim at improving the average response of tasks by means of *bandwidth-preservation*. This class of servers preserves their budget when no workload is pending to be serviced, so that the remaining budget can be used at a later moment in time when new tasks arrive. Contrary to bandwidth-preserving servers, other servers have a policy to deplete their unused budget, for example,

- a polling server [76] discards its remaining budget when no workload is pending (which affects its service guarantee negatively); and
- an idling server [64], similar to a TDM policy for $\Upsilon(\Pi, \Theta)$, idles away the remaining budget when no workload is pending.

Examples of bandwidth-preserving servers are the sporadic server [77] and the deferrable server [78]. The deferrable server, however, does not exactly behave as a periodic task, because the deferrable server can suspend itself when no workload is pending and, subsequently, it may execute two budgets back to back. The impact on the other components of handling tasks inside a deferrable server is measured by means of the *least upper bound on schedulability* [78][9], which measures the remaining processor bandwidth available to other tasks or components in the system when a particular server (e.g., a polling server or a deferrable server) executes at the highest priority. The sporadic server improves on the total utilization bound of the deferrable server, because it does not cause back-to-back executions of reserved budgets. Furthermore, a sporadic server is almost as good for the average response times of tasks as the deferrable server. The disadvantage of the sporadic server, however, is its difficult implementation, because one needs to keep track of when budget is consumed and how much is consumed. Some simplifications of the sporadic server have been proposed. However, these should be considered with care, since some of these simplifications suffer from anomalies. For example, the sporadic server described by the real-time POSIX standard may allocate too much processor time, thereby breaking temporal isolation between components [80].

In our work, we allocate a server to a component to enable predictable composition of components consisting of hard real-time tasks. The least amount of resources to be distributed by the server to these tasks can be computed by means of a resource-supply model. The resource-supply model does not make any assumption on the implementation of a specific server policy. An idling periodic server [64] or a TDM server are therefore often preferred for servicing

9. The remaining processor bandwidth after allocating a server to soft a-periodic workload at the highest priority, dimensioned with a utilization of U_s, is also referred to as the *total utilization bound* [78]. This bound is different from the so-called *utilization bounds* derived by, for example, Shin and Lee [57], [59] and Saewong et al. [79], who have derived a sufficient size U_s for a server that must service periodic workload with hard deadlines.

hard real-time workload, because of their predictable global interference pattern and their ease of implementation. However, in general the choice for binding a server to a component, i.e., the selection of the actual policy of distributing the processor reserves, is a deployment issue (see Paper D). During the timing analysis of a component the server type is therefore assumed to be unknown.

Most server policies found in standard text books, e.g., [37], [81], use their period parameter as an implicit deadline for the allocation of resources. In this way, the server guarantees a static processor utilization (also referred to as *processor bandwidth*) to its serviced tasks, thus, it implements a period bounded-delay model. As a consequence, a straightforward one-to-one mapping of other resource-supply parameters to server parameters may be impossible when the server parameters are more restrictive than the parameters of the resource-supply model. This may require a pessimistic over-allocation of processor bandwidth, especially for deadline-constrained tasks. Kumar et al. [82] have recently alleviated this pessimism by proposing a demand-bound server which guarantees a fixed quantum prior to an explicitly chosen deadline, i.e., they efficiently implemented a server policy for the EDP resource-supply model [56], $\Omega(\Pi, \Theta, \Delta)$. Our interest is not the relative performance of server policies; however, we are interested in servers as units of composition and the relative performance of them in combination with global synchronization protocols.

3.3 The rules of composition

Since we use tasks as units of composition in each level of the system's scheduling hierarchy, we can re-use the same schedulability analysis, no matter we schedule real tasks or components. In order to guarantee the deadline constraints of all tasks within the system, each artefact (a task or a component) within the entire scheduling tree must make its deadline. Compositional analysis enables us to analyze the schedulability of each component in the scheduling tree separately, assuming that all its children (i.e., the task set of the component) behave (sporadically) periodic, either by themselves or forced by means of, for example, a server.

Composition of independently developed components requires independence of the timing analysis of the global scheduling policies being implemented by a superficial parent component. The input for the local analysis of a component is a task set characterized by the period, WCET, deadline and critical section lengths of the tasks. Furthermore, we take as an input the relative task priorities, the local scheduling policies (for the processor and the other resources) and a resource supply model. The result of the local analysis is an interface. The interface of a component lacks any information about the inputs of the local analysis (such as the task model and the local scheduling policies), because those have become implementation details of the component. Nevertheless, the way in which interface parameters are computed can be influenced by them. This is similar to a task for which its internal functional behavior is also irrelevant globally. As such, we refrain from annotating an interface with the inputs of the local analysis.

Given a task set and the policies for scheduling tasks, we can verify whether or not all tasks meet their deadlines. We limit to stack-based policies [34] for arbitrating the access of tasks to shared resources. We now summarize the scheduling conditions for the bounded-delay resource-supply model.

According to Easwaran et al. [56], a task set $\mathcal{T}$ can be scheduled with the SRP and a fixed priority assignment upon an EDP resource $\Omega(\Pi, \Theta, \Delta)$, if

$$\forall i \; : \; 1 \leq i \leq n \; : \; (\exists t \; : \; t \in \mathcal{S}_i \; : \; b_i + \mathrm{rbf}(i,t) \; \leq \; \mathrm{lsbf}_\Omega(t)) \, , \tag{20}$$

where

- b_i is the maximum priority-inversion time experienced by task τ_i due to local resource sharing, as defined in (10);
- the $\mathrm{rbf}(i,t)$ is defined in (2); and
- $\mathcal{S}_i$ denotes a non-empty finite set of time points [31], see (3).

According to Easwaran et al. [56], a task set $\mathcal{T}$ can be scheduled with SRP+EDF upon an EDP resource $\Omega(\Pi, \Theta, \Delta)$, if

$$\forall t \; : \; t \in \mathcal{S} \; : \; b(t) + \mathrm{dbf}(t) \quad \leq \quad \mathrm{lsbf}_\Omega(t), \tag{21}$$

where

- $b(t)$ is the maximum priority-inversion time within an interval of length t due to local resource sharing, as defined in (11);
- the $\mathrm{dbf}(t)$ is defined in (5); and
- $\mathcal{S}$ denotes a non-empty finite set of time points [32], see (6).

The composition of a group of components follows the same rules as the composition of tasks. A parent component implements the global scheduling policies for the arbitration of its child component. Thus, in order to determine whether or not all child components meet their deadlines, we take as an input for the global scheduling analysis the following five elements:

1) a set of interfaces that specify the resource requirements of each of the child components in terms of (augmented) resource-supply models;
2) a server policy for each component to implement its resource-supply model as a regular task;
3) the relative component priorities and the corresponding policy to schedule the servers on the processor;

4) a synchronization protocol to arbitrate servers that wish to access other shared resources;
5) an (augmented) resource-supply model that determines the shape of the output of the analysis, i.e., the resources received by the parent component.

Based on these inputs, we can determine, using the conditions in (20) and (21), whether or not a parent component delivers sufficient time units (both on the processor and on other resources) in order to satisfy the deadline constraints of its child components (or tasks). This procedure is straightforward for the composition of independent components without global synchronization.

There is a catch in the composition of dependent components, however. We recall that the tasks perform the real work; thus, shared resources are actually accessed by the tasks within the components and not by the components themselves. In order to re-use the concepts of blocking that underly the conditions in (20) and (21), a component must truly behave as a task. As a consequence, a local task must initiate and complete any access to a shared resource in the same component period. In order to fulfill this requirement additional processor reserves might need to be allocated; this leads to a global increase of the rbf or the dbf of a component. There are several approaches for this, each of them with its own advantages and disadvantages, which deserve more attention (see Section 3.5).

Furthermore, dependent on the global scheduling policy of the child components, the bounded-delay approximation of the EDP resource-supply to a component, as modeled in the scheduling conditions (20) and (21), may be replaced by a more specific resource-supply model. For example,

- if a set of components is scheduled globally by TDM, then the deadline $\Delta_s = \Theta_s$ of each of those components C_s can be applied to tighten the analysis of the local tasks; i.e., the $\mathrm{lsbf}_{\Omega(\Pi,\Theta,\Delta)}(t)$ in the conditions in (20) and in (21) can be replaced by $\mathrm{sbf}_{\Upsilon(\Pi,\Theta)}(t)$.
- if an idling periodic server [64] is used to provision resources to a component, the periodic resource model by Shin and Lee [57] can be applied to tighten the analysis of the local tasks; i.e., the $\mathrm{lsbf}_{\Omega(\Pi,\Theta,\Delta)}(t)$ in the conditions in (20) and in (21) can be replaced by $\mathrm{sbf}_{\Gamma(\Pi,\Theta)}(t)$. An overview of servers belonging to this class, e.g., the idling periodic server, can be found in [83].

However, some global schedulers may explicitly require a bounded-delay analysis. For example, if a set of components is scheduled globally by the BROE [72] policy, then the bounded-delay resource-supply model must be used to analyze of the local tasks. The bounded-delay resource-supply model maximizes the independence of a component of its global scheduling policy and it therefore returns a better re-usability of the timing analysis of a component.

Exact algorithms for computing period-constrained budgets

Assume we are given a set of tasks and a local scheduling policy. We select a period parameter Π as the design parameter in the component interface, optionally complemented with an explicit deadline Δ. The question is then: what is the smallest budget Θ satisfying the scheduling condition in (20) or the condition in (21)?

In this section we answer this question. We re-consider the EDP resource-supply model, its periodic derivatives and its bounded-delay approximation.

Definition 2: We call an EDP interface $\Omega(\Pi, \Theta, \Delta)$ exact (or optimal), if for a given a task set $\mathcal{T}$ with local scheduling policy SP, a given period Π and a given deadline Δ the local schedulability condition by scheduling policy SP is satisfied with a budget Θ and with $\Theta - \epsilon$, for any infinitesimally small $\epsilon > 0$, the local schedulability condition violates that condition.

EDP resource model. Easwaran et al. [56] claim an exact algorithm for determining the budget of a set of independent tasks which has a better run-time efficiency than an exhaustive search proposed by Shin and Lee [57]. However, the algorithm in [56] may yield pessimistic budget allocations.

Example 2: Consider a component C_1 with a period $\Pi_1 = \Delta_1 = 10$ and with a single task $\tau_{1,1} = (27, 5, 27)$. Independent of the scheduling policy, EDF or FPS, Easwaran et al. [56] yield a budget of 3.42 time units. Equation (26) gives the same result as an exhaustive search: $\frac{8}{3}$ time units, which is an exact budget, since $\mathrm{rbf}(1, 27) = 5$ time units and the processor supply is $\mathrm{sbf}_{\Omega(10, \frac{8}{3}, 10)}(27) = 5$ time units. The budget allocations by Easwaran et al. [56] for EDP resources are therefore suboptimal.

The dissertation of Easwaran [60] presents an algorithm similar to his conference paper [56]. Contrary to [56], it comes with a proof. The algorithm in [60] suffers the same pessimism (see Example 2). We conclude that the algorithm by Easwaran et al. [56], [60], which considers both EDF scheduling and fixed-priority scheduling of tasks, is pessimistic, although it is claimed to be exact.

Fisher and Dewan [84] have presented a unique, fully polynomial approximation scheme (FPTAS) to calculate a budget for a given set of independent tasks scheduled by EDF, a given resource period Π and a given deadline Δ. Dewan and Fisher [85] have presented a different FPTAS for fixed-priority scheduling of tasks upon an EDP resource. We have shown in [86] that the algorithm in [85] may yield optimistic budgets and we proposed a correction for their algorithm. However, for an exact budget computation the algorithm in [86] can be simplified. We now present an algorithm which is computed more efficiently than the exhaustive search proposed by [57] and which returns an exact budget for a set of independent tasks.

Lemma 1: Given a component with a cumulative workload represented by variable $W(t)$ at time t, a period Π and a deadline Δ. The smallest budget Θ, satisfying the inequality $W(t) \leq \mathrm{sbf}_\Omega(t)$, is given by

$$\Theta \geq \frac{W(t)}{k} \quad \bigvee \quad \Theta \geq \frac{-t + W(t) + k\Pi + \Delta}{k+1} \tag{22}$$

such that $k \in \mathbb{N}^+$ and

$$\frac{W(t)}{k} < \frac{-t + W(t) + k\Pi + \Delta}{k+1} \Leftrightarrow k \geq \lfloor \ell \rfloor + 1 \tag{23}$$

and

$$\begin{aligned} \frac{W(t)}{k} \geq \frac{-t + W(t) + k\Pi + \Delta}{k+1} \quad &\Leftrightarrow \\ k \leq \lceil \ell \rceil \qquad & \text{if } \lceil \ell \rceil < \lfloor \ell \rfloor + 1 \\ k \leq \lceil \ell \rceil - 1 \qquad & \text{else,} \end{aligned} \tag{24}$$

where

$$\ell \quad = \quad \frac{t - \Delta + \sqrt{(t-\Delta)^2 + 4\Pi \cdot W(t)}}{2\Pi}. \tag{25}$$

Proof: Variable k reconstructs the value of $h_{(\Omega,t)}$ in (14) by computing the intersection between the two non-zero line segments in coordinate $(t,\ W(t))$, see (22). The intersection is characterized by the roots of the convex parabola $\Pi k^2 + k(\Pi - t) - W(t)$. The value of k is determined by observing that $h_{(\Omega,t)} \in \mathbb{N}^+$. The real-number representation of the positive root, ℓ, of the parabola is given in (25); its counter part always has a negative value, because the term in the square root dominates the preceding term. Since the left-hand sides of the bi-implications in (23) and (24) are strictly non-overlapping (mutually exclusive) inequalities, we have to guarantee a strictly smaller value $k \in \mathbb{N}^+$ in (24) than in (23). The case distinction detects whether or not $\lceil \ell \rceil = \lfloor \ell \rfloor + 1$. This concludes the proof. □

From Lemma 1, we can directly derive Algorithm 1 to compute an optimal budget for a given period Π and a given deadline Δ, satisfying the requested resources $W(t)$ at time t. Note that Algorithm 1 is independent of the local scheduling policy.

The design parameters of a component are the period Π and the deadline Δ. We now derive the EDP interfaces for task sets scheduled by a local fixed-priority scheduler or by a local EDF scheduler.

Given a period Π and a deadline Δ, we derive an EDP interface $\Omega(\Pi, \Theta^+, \Delta)$ under fixed-priority scheduling of tasks by rewriting the condition in (20) and by applying Lemma 1:

$$\Theta^+ = \max_{1 \leq i \leq n} \left\{ \min_{t \in S_i} \{\text{ComputePartialBudget}(\Pi,\ \Delta,\ b_i + \mathrm{rbf}(i,t),\ t)\} \right\}. \tag{26}$$

Algorithm 1 ComputePartialBudget(Π, Δ, $W(t)$, t)

```
1: $\ell = \frac{(t-\Delta)+\sqrt{(t-\Delta)^2+4\Pi\cdot W(t)}}{2\Pi}$
2: $\Theta_1 \leftarrow \frac{W(t)-t+(\lfloor\ell\rfloor+1)\Pi+\Delta}{\lfloor\ell\rfloor+2}$
3: $\Theta_2 \leftarrow \frac{W(t)}{\lceil\ell\rceil-1}$
4: if $\lceil\ell\rceil \leq \lfloor\ell\rfloor$ then
5:     $\Theta_2 \leftarrow \frac{W(t)}{\lceil\ell\rceil}$
6: end if
7: $\Theta_t^{\min} \leftarrow \min(\Theta_1,\ \Theta_2)$
8: return $\Theta_t^{\min}$
```

Equation (26) is a specialization of the exact instantiation of the approximation scheme for fixed-priority-scheduled EDP resources by Dewan and Fisher [85], but they have forgotten the if-statement that we have in line 4 of Algorithm 1. We recognized that this issue may lead to optimistic results [86].

Given a period Π and a deadline Δ, we derive an EDP interface $\Omega(\Pi, \Theta^+, \Delta)$ under EDF scheduling of tasks by rewriting the condition in (21) and by applying Lemma 1:

$$\Theta^+ = \max_{t\in\mathcal{S}} \{\text{ComputePartialBudget}(\Pi,\ \Delta,\ b(t)+\text{dbf}(t),\ t)\}\,. \tag{27}$$

For a set of independent tasks Equation (27) yields the same results as the exact characterization of the FPTAS presented by Fisher and Dewan [84]. Again, the dbf, rbf, the blocking terms (b_i and $b(t)$) and the sets $\mathcal{S}_i$ and $\mathcal{S}$ that appear in (26) and in (27) are defined in Section 2.1.

TDM resource model. We now present an algorithm to compute a budget for a component that is globally scheduled by TDM. Again, our algorithm is more efficient than an exhaustive search and it is optimal for independent tasks. Since under TDM holds $\Delta = \Theta$, the definition of $h_{(\Upsilon,t)} = \lceil \frac{t}{\Pi} \rceil$, see (14). Compared to Lemma 1, the absence of Θ in the ceiling term simplifies the re-construction of the value of $h_{(\Upsilon,t)}$.

Lemma 2: Given a cumulative workload of $W(t)$ at time t and a period Π for that component, the smallest budget Θ, satisfying the inequality $W(t) \leq \text{sbf}_\Upsilon(t)$, is given by

$$\Theta \geq \frac{W(t)}{k} \quad \bigvee \quad \Theta \geq \Pi + \frac{-t+W(t)}{k+1} \tag{28}$$

such that

$$k = \left\lfloor \frac{t}{\Pi} \right\rfloor. \tag{29}$$

Proof: Similar to Lemma 1. □

From Lemma 2, we can directly derive Algorithm 2 to compute an optimal budget for a given period Π and a given deadline Δ, satisfying the requested resources $W(t)$ at time t. Note that Algorithm 2 is independent of the local scheduling policy (similar to Algorithm 1).

Algorithm 2 ComputePartialTDMBudget(Π, $W(t)$, t)

```
1: $\Theta_1 \leftarrow \Pi + \frac{W(t)-t}{\lfloor \frac{t}{\Pi} \rfloor + 1}$
2: $\Theta_2 \leftarrow \frac{W(t)}{\lfloor \frac{t}{\Pi} \rfloor}$
3: $\Theta_t^{\min} \leftarrow \min(\Theta_1,\ \Theta_2)$
4: return $\Theta_t^{\min}$
```

The design parameter of a component is the period Π. We now derive the TDM interfaces for task sets scheduled by a local fixed-priority scheduler or by a local EDF scheduler.

Given a period Π, we derive a TDM interface $\Upsilon(\Pi, \Theta^+)$ under fixed-priority scheduling of tasks by rewriting the condition in (20) and by applying Lemma 2:

$$\Theta^+ = \max_{1 \le i \le n} \left\{ \min_{t \in S_i} \{\text{ComputePartialTDMBudget}(\Pi,\ b_i + \text{rbf}(i,t),\ t)\} \right\}. \quad (30)$$

Given a period Π, we derive a TDM interface $\Upsilon(\Pi, \Theta^+)$ under EDF scheduling of tasks by rewriting the condition in (21) and by applying Lemma 2:

$$\Theta^+ = \max_{t \in \mathcal{S}} \{\text{ComputePartialTDMBudget}(\Pi,\ b(t) + \text{dbf}(t),\ t)\}. \quad (31)$$

Again, the dbf, rbf, the blocking terms (b_i and $b(t)$) and the sets $\mathcal{S}_i$ and $\mathcal{S}$ that appear in (30) and in (31) are defined in Section 2.1.

Periodic resource model. Given a task set, a local scheduling policy and a component period Π, we can re-use the algorithms for computing exact EDP budgets in (26) and in (27) for the computation of an exact budget satisfying the periodic resource model by Shin and Lee [57]. This simply requires to fill in $\Delta = \Pi$, see (16).

Bounded-delay resource model. Assume we are given a set of tasks and a local scheduling policy. We select a period Π and an explicit deadline Δ for this component. We now search for the tightest possible bounded-delay approximation characterized by Θ that satisfies the scheduling condition in (20) or the condition in (21).

This question has been solved by Lipari and Bini [5, Section V] for fixed-priority scheduling of tasks upon the periodic resource model. Their result can be straightforwardly generalized for other local scheduling policies and for the EDP resource model. A generalization of their approach to the EDP resource-supply model works as follows.

Lemma 3: Given a cumulative workload of $W(t)$ at time t, a period Π and a deadline Δ for that component, the smallest budget that satisfies the inequality $W(t) \leq \mathrm{lsbf}_{\Omega}(t)$, i.e., returned by ComputePartialBDMBudget($\Pi,\ \Delta, W(t),\ t$), is given by

ComputePartialBDMBudget($\Pi,\ \Delta, W(t),\ t$) =

$$\frac{(\Pi + \Delta - t) + \sqrt{(\Pi + \Delta - t)^2 + 8\Pi W(t)}}{4}. \tag{32}$$

Proof: Simply compute the roots of the parabola defined by the inequality $W(t) \leq \mathrm{lsbf}_{\Omega}(t)$; this is similar to Lemma 1. □

Given a period Π and a deadline Δ, the tightest possible linearly approximated EDP interface $\Omega(\Pi, \Theta^-, \Delta)$ under fixed-priority scheduling of tasks can be obtained by rewriting the condition in (20) and by applying Lemma 3:

$$\Theta^- = \max_{1 \leq i \leq n} \left\{ \min_{t \in \mathcal{S}_i} \{\text{ComputePartialBDMBudget}(\Pi,\ \Delta,\ b_i + \mathrm{rbf}(i,t),\ t)\} \right\}. \tag{33}$$

Given a period Π and a deadline Δ, the tightest possible linearly approximated EDP interface $\Omega(\Pi, \Theta^-, \Delta)$ under EDF scheduling of tasks can be obtained by rewriting the condition in (21) and by applying Lemma 3:

$$\Theta^- = \max_{t \in \mathcal{S}} \{\text{ComputePartialBDMBudget}(\Pi,\ \Delta,\ b(t) + \mathrm{dbf}(t),\ t)\}. \tag{34}$$

Again, the dbf, rbf, the blocking terms (b_i and $b(t)$) and the sets $\mathcal{S}_i$ and $\mathcal{S}$ that appear in (33) and in (34) are defined in Section 2.1.

Finally, note that a tight bounded-delay approximation of the TDM model can be obtained by repeating the steps in this section. Similar to Lemma 2, one may use the definition of the TDM model, as expressed in (15) in terms of the EDP model. We leave the details as an exercise.

The models we will use in this work, their performance and complexity

In this work, we will mainly consider a set of components whom's resource requirements are specified by means of the periodic resource model by Shin and Lee [57] and its bounded-delay approximation, augmented with resource sharing. Each component consists of a set of sporadic, deadline-constrained tasks and a local scheduler (in some included papers in Part-II the model is slightly more liberal). The corresponding algorithms for computing periodic budgets, as presented in this section, are optimal for fully preemptive independent tasks; the algorithms may over-allocate processor time to tasks that share resources, because it is unknown when the limited-preemptive region is executed (as explained in Section 2.2). These algorithms are used for several experiments carried out in Part-II of this work.

We use the periodic resource-supply model to abstract the timing requirements of a component, because the model allows for the maximum freedom of delivering the processor time. A component, i.e., the local tasks and their local scheduler, are treated as if it was a single task on itself, independently of any other components or tasks in the system. This locality of the analysis comes with abstraction overhead. The corresponding abstraction overheads have been extensively investigated by, for example, Shin and Lee [57], [59]. They have compared sets of independent tasks scheduled upon the periodic-resource model and upon the bounded-delay model. Their results can be extended with support for local resource sharing in a straightforward way [57].

The complexity of analyzing a schedule of a component in an HSF is pseudo polynomial, i.e., similar to the complexity of a task set that has the entire processor at its disposal. In fact, the schedule of a component in an HSF can be represented by a flattened task set that occupies the entire processor (see Figure 9). In line with these similarities between hierarchical and non-hierarchical schedules, there exist FPTASes for determining scheduling bounds for independent sporadic tasks scheduled upon an EDP resource by various local policies. For example, the FPTAS by Albers and Slomka [25] for EDF scheduling of tasks has been extended for use within the EDP resource-supply model by Fisher and Dewan [84]. Similarly, the FPTAS by Fisher and Baruah [26] for fixed-priority scheduling of tasks has been extended for use within the EDP resource-supply model by Dewan and Fisher [85].

3.4 Handle external events with the destined task priority

An important source of unpredictability for real-time systems comes from a-synchronous events [37]. There can be various event sources such as timer interrupts, I/O interrupts and software interrupts (or events). Within an hierarchical real-time system, external events or interrupts can trigger the tasks of arbitrary components. This makes it even harder than in a non-hierarchical system to isolate other tasks in the system from the unpredictable interference caused by the handling of those arrived events.

The handling of programmed timer interrupts and time-keeping for components constrained by reservations are typically a special concern. For example, the hypervisor of Barham et al. [87] separates different timer queues for each partition (which hosts a component), so that each client OS can program its own real and virtual timers. The SPIRIT μkernel [88], XtratuM [89] and VMware [90] by default follow a similar approach. For backwards compatibility, VMware allows immediate forwarding of timer interrupts when they occur in real time. The latter does not scale, because each interrupt triggers a context switch to the receiving partition. Nevertheless, this effectively supports legacy OSes requiring a strictly periodic occurrence of a timer interrupt for local time keeping, e.g., Linux 2.4. As a result, the time within a component progresses in the same way as the real time.

In most cases, the time within a component progresses differently than the real time when a component is executed on a shared platform, because of the intervals of time where the processor is unavailable for the component. To track the so-called virtual time, a dedicated timer exists in the Portable Operating System Interface (POSIX) standard. Each process running on a POSIX-compliant platform has a virtual timer available that expires relative to its consumed processor time. When the virtual timer expires, a software interrupt – called a *signal* – is sent to the process. It is then the task of the HSF to deliver these signals to the destined threads without hampering the timing of other threads.

There are different type of signals; some signals can be sent by one thread to an arbitrary other thread in the system. For example, threads may make I/O requests on devices that respond with interrupts [91]. These interrupts typically arrive when the destined process is inactive. Zhang et al. [92] have clearly separated between top-half and bottom-half interrupt handlers of an hardware source. The top-half interrupt handlers are typically assumed to be negligibly short[10].

Zhang et al. [92] focussed on predictable handling of the bottom-half interrupts, because these can take significantly more processor time than their top-half counter parts. In fact, bottom-half interrupt handlers are the same as software interrupts (signals) and they can be seen as high-priority threads. Contrary to paying attention to these priorities, however, Zhang et al. [92] proposed mechanisms within Linux to ensure that bottom-half interrupt handlers are scheduled in accordance with the urgency and importance of the threads (tasks) that led to their occurrence. In this way, the overheads for handling interrupts is charged to the associated thread or process. Parmer et al. [91] have adopted the approach by Zhang et al. [92] for an operating system that supports hierarchical composition of components, i.e., called CompositeOS. For this purpose, they have presented new mechanisms to limit unnecessary calls to external schedulers of other components in the HSF; that is, scheduler calls are deferred until the receiving thread of an event must urgently execute.

A key for predictable composition of components in an HSF is managing interrupts. In many of the works described so far, e.g., [88], [87], [89], [92] and [91], the enabling technology is a conversion of real interrupts (both from

10. In case this assumption may harm the predictability of the system, filters implemented in dedicated hardware have been presented, e.g., by Agron et al. [93], in order to regulate the delivery of hardware interrupts to the processor. Also software-based solutions have been presented, e.g., see [94] and [95], for systems where each interrupt source can be enabled and disabled separately.

real hardware sources and from software sources) into light-weight events that can be managed. The light-weight events are then delivered to the destined task according to the task's priority, i.e., the interrupt appears to a task from a virtual source. We observed that many academically implemented HSFs inside real-time operating systems refrain from the appropriate virtualization of interrupt sources other than timer interrupts. For example, this holds for the HSFs in VxWorks [96], FreeRTOS [97] and, the one on which we base our work, μC/OS-II [98]. This is different in, e.g., RED Linux [99] and CompositeOS [91]. The latter two introduced buffering of the arrived events until the destined task has the highest priority to execute. Thus, the priority of a component on itself is unimportant; it only relates the priorities of its local tasks to the priority of the tasks that are located in different components.

After a duration of unavailable processor time, a component in the HSF can have multiple pending events. These are locally handled by several tasks in a (static or dynamic) priority order. Proper management of events in component-based systems has been a well studied problem. However, none of the existing HSF-based platforms guarantees that the order in which event-triggered tasks are activated in a component is preserved compared to the order of execution of that component on a dedicated processor. We propose a *virtual-scheduling* mechanism to complement the existing mechanisms for event handling in an HSF with such support (see paper B). In this way, legacy real-time components that have been analyzed upon a slower, dedicated processor can be integrated by fitting a resource-supply model without the need to repeat the timing analysis of the tasks. With our virtual-scheduling approach, events are by definition delivered during the execution of the destined component.

3.5 Set a component deadline to constrain resource sharing

Consider a component C_s which requires access to a set of resources. We have earlier defined an augmented resource-supply model $\Omega_s = (\Pi_s, \Theta_s, \Delta_s, \mathcal{X}_s)$, which includes the resource holding times of a component upon its accessed non-preemptive resources. These resource holding times of a component determine the blocking caused to other components during admission control (i.e, the global analysis of the composed components), similar to the analysis of resource-sharing tasks (see Section 2.2). The additional meaning of these resource holding times – considered in this section – is a difficult one and it can affect the timing analysis in different ways.

Firstly, we recall that only Θ_s must be delivered prior to Δ_s in order to satisfy deadline constraints of the local tasks. Secondly, the interface Ω_s gives $X_\ell \in \mathcal{X}_s$ time units of processor time, provisioned no later than Π_s time units from the start of the period Π_s in which access to R_ℓ is granted.

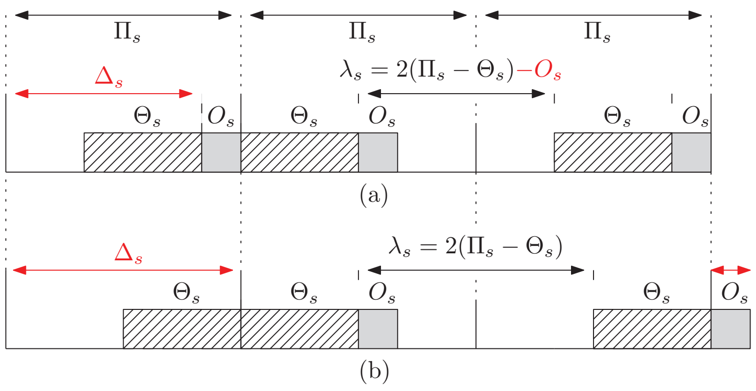
Π_s
Π_s
Π_s
Δ_s
$\lambda_s = 2(\Pi_s - \Theta_s) - O_s$
Θ_s
O_s
(a)
Δ_s
$\lambda_s = 2(\Pi_s - \Theta_s)$
(b)

Fig. 11. Setting a component deadline for global resource sharing of local tasks may reduce the service delay of processor time, (a), and it may therefore improve the local schedulability. Without such a component deadline, (b), the global schedulability of components may improve.

The second requirement may impose more rigorous changes than strictly necessary to the way in which the processor resources are delivered to a component. For example, the computed value of Θ_s can be smaller than a resource holding time $X_\ell \in \mathcal{X}_s$. When a task of component C_s is granted access to resource R_ℓ, this component requires at least a worst-case delivery of X_ℓ time units of processor supply within its current period Π_s, i.e., an over-allocation of at least $O_s = X_\ell - \Theta_s$. The exact size and nature of the over-allocations, O_s, depend on the synchronization protocol being used (see Paper C).

Just for the ease of component analysis, we have chosen to make the period of delivering the over-allocations of budgets synchronous with the component period. In fact, the extra processor time, O_s, can be delivered synchronously or a-synchronously with the component's period Π_s. Figure 11 shows these two options. Figure 11(a) shows that the resource access must complete within the same component period (but it may overrun the deadline Δ_s); Figure 11(b) shows that the resource access is allowed to overrun also the component's period, Π_s, a-synchronously. Despite the extra freedom in Figure 11(b), any extra delivery of processor time must ultimately guarantee that access to a global resource is completed by the same job of a component as the job that initiated the access, just like the tasks.

It is a complicated choice whether synchronization of global resources should be constrained with the component period synchronously or a-synchronously, because both have their advantages and disadvantages in terms of the estimated scheduling penalties for global resource sharing. However, it is not just these scheduling penalties that are affected. It is also the re-usability of the component that may be sacrificed. We briefly look at the impacts on both the local scheduling analysis of a component and the global scheduling analysis being used for composing a component with other components.

Local scheduling analysis. Assume a synchronous deadline Π_s for finishing granted access to a globally shared resource. We may be able to use this information to tighten the required budget Θ_s of a component, see Figure 11. Since an over-allocation O_s has to fit within Π_s, it makes no sense to choose a deadline Δ_s larger than $\Pi_s - O_s$. Applying this decrease of deadline Δ_s leads to a smaller service delay, λ_s. The decrease of λ_s may consequently decrease the size of budget Θ_s.

However, the reduction of Θ_s also has down sides. The reasoning is as follows. Budget Θ_s must be delivered within deadline Δ_s. Firstly, there are not so many server policies that allow a constrained deadline, which makes its hard to implement the computed interface. Secondly, the size of budget Θ_s is affected by the assumption that any access to any resources is arbitrated through a global synchronization protocol. The need for a global synchronization protocol depends also on other components. Even if global sharing of a resource is unnecessary, budget Θ_s must be delivered prior to the tightened deadline Δ_s. In this case, the analysis performs poor, because the so-called *density*, $\frac{\Theta_s}{\Delta_s}$, of the global schedule is higher than necessary. A larger deadline Δ_s might have resulted in a larger budget Θ_s, but it also leads to a lower density. This improves the global schedulability.

Whether global resource sharing is arbitrated a-synchronously or synchronously with the period of the processor supply is irrelevant during the local analysis. Moreover, the reusability of a component can be increased by means of entirely ignoring the global synchronization protocol during the local analysis.

Global scheduling analysis. As we have just concluded, the choice of allocating over-allocations of budget O_s synchronously or a-synchronously with period Π_s is supposed to be a decision at the global scheduling level. An a-synchronous allocation, as depicted in Figure 11(b), may lead to a tighter global analysis, thus, more components can be scheduled jointly on the same platform. For one global synchronization protocol, Keskin et al. [100] have compared two synchronous allocation policies, i.e., the ones in [73] and in [101], and a novel a-synchronous allocation policy. Experimental results indicate that this choice has little impact on the ratio of systems that can be scheduled successfully. The improvements of the scheduling analysis in [101] and [100] mainly come from the applied limited-preemption techniques and the tighter local analysis. Although the scheduling analysis may slightly improve, the disadvantage of allowing a-synchronous over-allocations is the additional implementation complexity for keeping track of a-synchronous budget replenishment [100].

We therefore opt for analyzing the re-allocations of budgets due to global synchronization by means of using one synchronous deadline for each resource-sharing component.

We conclude that it is useful to set the period Π as a component deadline to constrain resource sharing during the global scheduling analysis. In this way, we can keep the local timing analysis of a component independent of its global scheduling policy and synchronization protocol. This property of the timing analysis of a component is called *opacity* (Paper C). The remaining questions, being imposed by the need for global resource sharing, are about the reservations of a component C_s on non-preemptive resources, i.e.,

- how are the allocations of O_s managed by the different protocols?
- how are the allocations of O_s and the resource holding times in $\mathcal{X}_s$ enforced?

The remainder of this section briefly addresses these two topics.

Global synchronization protocols

We already saw that there is a need to complement the traditional synchronization protocols, e.g., the PIP, the PCP and the SRP, with mechanisms that ensure that tasks finish their critical section in the same component period as the critical section has been initiated in. We give a brief overview of the different protocols found in literature. The PIP extensions are summarized by bandwidth-inheritance (BWI) protocols; the SRP-based flavors are the main topic of our work and these are therefore summarized separately.

Bandwidth inheritance (BWI). Steinberg [102] have implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach has been described by Lipari et al. [103] for earliest-deadline-first (EDF)-based systems and it is termed bandwidth-inheritance (BWI).

BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol (PIP) by Sha et al. [36]: when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. The donatee then executes in the budget of the donor and at the priority of the donor until it releases the resource. BWI-like protocols are not very suitable for arbitrating hard real-time tasks in HSFs, because the worst-case interference of all tasks in other components that access global resources needs to be added to a component's budget at integration time in order to guarantee its internal tasks' schedulability [104] (which is even worse than PIP's blocking, see Section 2.2). This leads to pessimistic budget allocations for hard real-time components.

Hierarchical stack resource policy (HSRP). The HSRP [105] trivially extends the SRP to HSFs: a task that wishes to access a global shared resource cannot be blocked upon its attempt. Since HSRP's lock operation is non-blocking, the HSRP allows the tasks of a component to overrun their budget until the resource is released. HSRP has two flavors: overrun with payback (OWP) and overrun without payback (ONP). The term without payback means that the additional amount of budget consumed during an overrun does not have to be returned in the next budget period.

Subsystem integration and resource allocation policy (SIRAP). The SIRAP [106], [107] allows a task to access a global resource only if it has sufficient budget to complete the entire critical section. If a task attempts to access a resource and the remaining budget of the component is insufficient, then the task blocks itself until the component's budget is replenished, i.e., the remainder of the budget is idled away. As soon as sufficient budget is available, the SRP is used to arbitrate resource access between the components globally. Strictly speaking, the SIRAP is different from the SRP, because a task arbitrated by the SRP cannot block at the time of accessing a resource. The challenge of analyzing a component arbitrated by the SIRAP is to bound the amount of inserted idle time in each component period [107].

Bounded-delay resource-sharing open environment (BROE). BROE [72] comes with a special server policy for a component and this server implements the bounded-delay resource-supply model. If a tasks serviced by a BROE server requests access to a global shared resource, the entire server may suspend itself. However, such a request only causes a server self-suspension if there is insufficient budget to complete the critical section and if the supplied budget by the server is running ahead with respect to its guaranteed processor utilization. If there is insufficient budget to complete the critical section and if the supplied budget by the server is running late, then the budget of the server is immediately replenished; however, the replenished budget is served at a lower priority. BROE is restricted to global SRP+EDF scheduling of components and it cannot be generalized for other global scheduling policies.

Temporal isolation of independent components

We have earlier defined an augmented resource-supply model which captures the component's reservations on the processor as well as on non-preemptive resources. A server implements the processor reservations. If a component wishes to execute more than it has specified in its timing interface, then the server prohibits it to do so. The same degree of temporal isolation between components is difficult to achieve when a task shares resources with other tasks located across their processor reservation [108]. Whenever a component exceeds its resource holding time, as specified in its interface, other components in the system also risk missing their deadlines.

Consider a malicious situation where a task enters an infinite critical section; obviously, this violates any reasonable timing interface. A nice property of the

BWI is that only those components that share the same resource are affected by the overrunning critical section. The reason is as follows. The malicious component itself cannot consume more processor time than it has reserved; this is enforced by a server. In addition, the malicious component may receive the reserved processor time from donor components in order to continue the critical section. However, the amount of donated processor time is bounded by the amount that the donor would have ultimately available for itself. We conclude that BWI isolates other components in the system that do not share the same resource from the malicious effects of an overrunning critical section.

If resource access is arbitrated by a ceiling-based protocol, like the SRP or the PCP, there is a risk that even independent components miss a deadline whenever an arbitrary component overruns its specified resource holding time. For example, consider the SRP. We recall that scheduling performance of the SRP is made possible by so-called avoidance blocking (see Figure 6 in Section 2.2). Thus, a component with a preemption level lower than the resource ceiling may experience blocking, even when it is entirely independent. We have proposed an SRP extension to resolve this issue (Paper E), which is called *hierarchical synchronization with temporal protection* (HSTP).

The key idea of HSTP is to monitor the resource holding times of a component. When a component attempts to exceed the specified length of the resource holding time, a preemption point is placed immediately prior to the continuation of the executing component. An overrunning critical section forces the behavior of PTS^{+} to a component (see Section 2.2) and any independent component will therefore continue meeting its deadlines. The accessed resource stays occupied by the component, so that no other component can access the same resource.

Apart from HSTP's temporal protection mechanism for independent components, we also proposed bandwidth-donation mechanisms to alleviate the scheduling penalties for the resource-sharing components. Our mechanisms are tailored to the SRP-based protocols described above, i.e., HSRP, SIRAP and BROE, and they preserve a special property: even if a component executes on a donation, it will not hamper independent components. Santos et al. [109] proposed an extension to the BWI, called the clearing fund protocol (CFP), to compensate for excessive donations, i.e., the inheritor component becomes a debtor of the donor component. It may be possible to make the CFP applicable to SRP-based protocols, but the CFP is tailored to the constant bandwidth server (CBS) [110] and it is hard to make the CFP applicable to other server policies. Contrary to the CFP, our bandwidth-donation mechanisms apply to any server with periodic processor allocations. In any case, the intervention in the schedule with bandwidth donations after an excessive resource holding time – as is done by the CFP [109] and optionally by HSTP (Paper E) – lacks further hard real-time guarantees to any of the components involved.

4 RESULTS, DISCUSSION AND CONCLUSIONS

In the remainder of this section, we first give an overview of the results and scientific contributions presented in this work. Secondly, we conclude and discuss the obtained results.

4.1 Overview of the included papers

In this section, we give an overview of the contributions in the papers included in the second part of this work. The notations, the assumptions and the models may differ slightly per paper.

Paper A – Generalized fixed-priority scheduling with limited preemptions

R.J. Bril, M.M.H.P. van den Heuvel, U. Keskin and J.J. Lukkien, *Generalized fixed-priority scheduling with limited preemptions*, in Proc. 24th Euromicro Conference on Real-Time Systems (ECRTS), pp. 209–220, Pisa, Italy, July 2012.

Abstract Fixed-priority scheduling with deferred preemption (FPDS) and fixed-priority scheduling with preemption thresholds (FPTS) have been proposed in the literature as viable alternatives to fixed-priority preemptive scheduling (FPPS), that reduce memory requirements, reduce the cost of arbitrary preemptions, and may improve the feasibility of a task set even when preemption overheads are neglected.

This paper aims at advancing the relative strength of limited-preemptive schedulers by combining FPDS and FPTS. In particular, we present a refinement of FPDS with preemption thresholds for both jobs and sub-jobs, termed FPGS. We provide an exact schedulability analysis for FPGS, and show how to maximize the feasibility of a set of sporadic tasks under FPGS for given priorities, computation times, periods, and deadlines of tasks. We evaluate the effectiveness of FPGS by comparing the feasibility of task sets under FPGS with other fixed-priority scheduling algorithms by means of a simulation. Our experiments show that FPGS allows an increase of the number of task sets that are schedulable under fixed-priority scheduling.

Contribution The basic idea of this paper, i.e., the generalized scheme for fixed-priority schedulers and the corresponding response-time analysis, has been proposed by Reinder Bril. This paper continued an earlier work [49] by Ugur Keskin, Reinder Bril and Johan Lukkien for more specific fixed-priority limited-preemptive schedulers. Martijn van den Heuvel was the main responsible for the schedulability analysis that uses request-bound functions, the derivation of the corresponding optimization algorithms and the simulations.

Paper B – Virtual scheduling of periodic tasks for compositional real-time guarantees

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Virtual scheduling of periodic tasks for compositional real-time guarantees*, In Proc. 8th IEEE International Symposium on Industrial Embedded Systems (SIES), (to appear), Porto, Portugal, June 2013.

Abstract Consider a legacy application executing a set of time-triggered periodic tasks on a uni-processor platform. In this paper, we extend the hierarchical scheduling framework to allow the integration of the same application as a component on a faster processor which needs to be shared with other components. After admission of this application into the framework, the integrated component still has to satisfy its internal deadline constraints and it must execute jobs in the same order as on the dedicated reference processor, regardless of the actual supply of processor resources. We propose a method for this – called *virtual scheduling* – which is independent of the component-level scheduling policy. Moreover, virtual scheduling is transparent to a component, so that it is even applicable without making modifications to the code or specification of the application.

Contribution The main idea of this paper is from Martijn van den Heuvel. The idea has been further developed and refined in close co-operation with both Reinder Bril and Johan Lukkien.

Paper C – Opaque analysis for resource-sharing components in hierarchical real-time systems

M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte, *Opaque analysis for resource-sharing components in hierarchical real-time systems*, submitted to the real-time systems journal.

Abstract Hierarchical scheduling frameworks (HSFs) have been developed to enable composition and reuse of independently developed and analyzed real-time components in complex systems. In practice, these components share more resources than just the processor, requiring arbitration through a global (system-level) synchronization protocol.

In this paper we propose opaque analysis methods to integrate resource-sharing components into uni-processor HSFs. A local (component-level) schedulability analysis is opaque if it is independent (or agnostic) of the global synchronization protocol. An individual component can therefore be analyzed as if all resources are entirely dedicated to it. This locality of the analysis obtained from opacity enables us to derive a computationally tractable method for

exploring and selecting the design parameters of resource-sharing components with the objective to minimize the system load. Moreover, given a real-time interface of a component that is derived by means of an opaque analysis, the component can be used with an arbitrary global synchronization protocol. Hence, opacity extends the independence of a component of the global scheduling model, thereby effectively increasing the reusability of the component.

To arbitrate resource access between components, we consider four existing protocols: SIRAP, BROE and HSRP – comprising overrun with payback (OWP) and overrun without payback (ONP). We classify local analyses for each synchronization protocol based on the notion of opacity and we develop new analysis for those protocols that are non-opaque.

Finally, we compare different analyses for SIRAP, ONP, OWP and BROE by means of an extensive simulation study. From the results we derive guidelines for selecting a global synchronization protocol.

Contribution The main idea of opaque analysis, including the new overrun analysis, came from Martijn van den Heuvel. The notion of opacity has been further leveraged in close co-operation with mainly Moris Behnam and Reinder Bril. The idea to exploit the property of opaque analysis in order to optimize the overall load on the processor came from Moris; the algorithms and the simulations have been developed by Martijn van den Heuvel. All authors have contributed to the writing of the paper.

Paper D – Transparent synchronization protocols for compositional real-time systems

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Transparent Synchronization Protocols for Compositional Real-Time Systems*, in IEEE Transactions on Industrial Informatics (TII), pp. 322–336, vol. 8, issue 2, May 2012.

Abstract Hierarchical scheduling frameworks (HSFs) provide means for composing complex real-time systems from well-defined, independently analyzed components. To support resource sharing in two-level HSFs, three synchronization protocols based on the stack resource policy (SRP) have recently been presented for single-processor execution platforms, i.e., HSRP, SIRAP and BROE. This paper presents a transparent implementation of these three protocols side-by-side in an HSF-enabled real-time operating system. Transparent synchronization interfaces make it possible to select a protocol during integration time based on its relative strengths.

A timing interface describes the required budget to execute a component on a shared platform and an accessor’s maximum critical-section execution time to global shared resources. These resources are arbitrated based on the available

budget of the accessing task. We enable this explicit synchronization of virtual time with global time by means of a novel virtual-timer mechanism. Moreover, we investigate system overheads caused by each synchronization protocol, so that these can be included in the system analysis. Based on the analytical and implementation overheads of each protocol, we present guidelines for the selection of a synchronization protocol during system integration.

Finally, we show that unknown task-arrival times considerably complicate an efficient implementation of SIRAP's self-suspension mechanism. We briefly discuss the implementation complexity caused by these arrivals for bandwidth-preserving servers, e.g., deferrable servers and BROE.

Contribution The main idea of providing transparent programming interfaces, regardless of the synchronization protocol, came from Reinder Bril. The different synchronization protocols considered in this paper have been implementated and evaluated by Martijn van den Heuvel. All authors have contributed to the writing of the paper.

Paper E – Dependable resource sharing for compositional real-time systems

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Dependable resource sharing for compositional real-time systems*, in Proc. 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 153–163, Toyama, Japan, August 2011.

Abstract Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating temporal isolation between components that need to be integrated on a single shared processor. In the presence of shared resources, however, temporal isolation may break when one of the accessing components executes longer than specified during global resource access. The ability to confine such temporal faults makes the HSF more dependable. As a solution we propose a stack-resource-policy (SRP)-based synchronization protocol for HSFs, named Hierarchical Synchronization protocol with Temporal Protection (HSTP).

When a component exceeds its specified critical-section length, HSTP enforces a component to self-donate its own budget to accelerate the resource release. In addition, a component that blocks on a locked resource may donate budget. The schedulability of those components that are independent of the locked resource is unaffected. HSTP efficiently limits the propagation of temporal faults to resource-sharing components by disabling local preemptions in a component during resource access. We finally show that HSTP is SRP-compliant and applies to existing synchronization protocols for HSFs.

Contribution The main idea of developing isolation mechanism in order to protect independent components against other misbehaving resource-sharing components came from Martijn van den Heuvel. The different policies that exhibit our design criterion have been lined up in close co-operation with Reinder Bril.

Paper F – Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources*, in Proc. 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 39–48, Porto, Portugal, July 2011.

Abstract Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating temporal isolation between components that need to be integrated on a single shared processor. To support resource sharing within two-level, fixed priority scheduled HSFs, two synchronization protocols based on the stack resource policy (SRP) have recently been presented, i.e. HSRP and SIRAP. In the presence of shared resources, however, temporal isolation may break when one of the accessing components executes longer than specified during global resource access. As a solution we propose a SRP-based synchronization protocol for HSFs, named Basic Hierarchical Synchronization protocol with Temporal Protection (B-HSTP). The schedulability of those components that are independent of the unavailable resource is unaffected.

This paper describes an implementation to provide HSFs, accompanied by SRP-based synchronization protocols, with means for temporal isolation. We base our implementations on the commercially available real-time operating system μC/OS-II, extended with proprietary support for two-level fixed priority preemptive scheduling. We specifically show the implementation of B-HSTP and we investigate the system overhead induced by its synchronization primitives in combination with HSRP and SIRAP. By supporting both protocols in our HSF, their primitives can be selected based on the protocol's relative strengths.

Contribution The main idea of developing isolation mechanism in order to protect independent components against other misbehaving resource-sharing components came from Martijn van den Heuvel. This paper compares the protocol implementations in Paper D to new implementations of those protocols that are complemented with the (basic) isolation mechanisms presented in paper E. The experiments have been carried out by Martijn van den Heuvel.

4.2 Discussion and conclusions

In this work we revisited synchronization and composition of real-time components upon a single processor and other non-preemptive resources. We focused on two types of synchronization between the tasks located in arbitrary components: a-synchronous resource sharing and communication (i.e., event or interrupt driven) and synchronous resource sharing and communication (via so-called non-preemptive resources). We give a brief overview of the lessons we have learned related to these forms of synchronization.

The independence of a real-time component of its global scheduler is important during its development. We therefore made the timing requirements of the tasks transparent regardless of the component's dependencies on these two types of task synchronization (Paper B and Paper D). This means that the component developer can be unaware that the processor and other non-preemptive resource may need to be shared with other real-time components.

The composition of components forms an hierarchical scheduling framework (HSF), which is established via well-defined timing interfaces. In order to derive a timing interface for a component, we must perform timing analysis. The analysis compares the tasks' timing characteristics to a resource-supply model which bounds the variability of any of the supplied resources. In Paper C, we have augmented the resource-supply models for HSFs with a method that computes the resource sharing parameters of a component determinately, regardless of the global scheduling and the global resource-arbitration policies.

Unfortunately, it is hard to abstract entirely from the resource-supply model in the timing-requirements analysis of a component. The role of the resource-supply model is to abstract from (*i*) the task model of local tasks and (*ii*) the actual allocation policy of processor bandwidth and other resources. The latter requires at least the awareness of the existence of those allocation policies, i.e., the allocated processor time might be supplied discontinuously. It is possible, however, to enforce the same schedule regardless of the delays in the processor supply, as long as the maximum delay is bounded.

We have proposed a method (see Paper B) – called *virtual scheduling* – to establish this. A major advantage of virtual scheduling for the composition of a system is that an individual component virtually executes on a processor with a continuous supply and, thus, a real-time component preserves its predictable schedule after composition with other components.

After admitting a component on a shared platform with other components, the HSF reserves the processor share and the shares of the other non-preemptive resources required by that component. Also the access to those resources is arbitrated through the allocated virtual platform. From the component's point of view, it is better to allocate larger quanta. This reduces the cost of preemptions and makes it easier to schedule tasks. From the platform's point of view, it is better to allocate smaller quanta. This better approximates the supply function and smaller quanta therefore lead to less over-provisioning of processor bandwidth. This trade off affects the period of the reservations, captured in (or derived from) the interface of a component.

There are many different policies to provision the reserved processor bandwidth. Since the resource-supply model abstracts from these server policies anyway, we conclude that the most suitable and predictable method of composing hard real-time components is through periodic, idling processor reservations. This policy shapes a component as a simple periodic task. As investigated in Paper D, many of the more complicated processor allocation policies introduce either more allocations of processor bandwidth during the timing analysis or higher implementation costs (and sometimes even both).

We also investigated the analytical scheduling penalties and the implementation costs of SRP-based real-time synchronization protocols in HSFs (Paper D). These protocols provide well-defined programming interfaces for arbitrating access to globally shared non-preemptive resources. We considered the following protocols: HSRP – comprising overrun without payback (ONP) and overrun with payback (OWP) –, SIRAP and BROE. Each of these protocols presents different scheduling mechanisms that prevent budget depletion during the execution of a component on a non-preemptive resource. For this purpose, HSRP allows a component to overrun its allocated budget temporarily. Contrary to the HSRP, both SIRAP and BROE use a self-blocking mechanism in order to avoid budget overruns for the completion of a critical section.

Looking at the scheduling analysis of SIRAP and BROE, they both outperform the HSRP. The real-time performances of SIRAP and BROE are competitive to each other, but the scheduling performance of SIRAP is less sensitive to task characteristics than BROE is. The reasons are twofold (see Paper C):

1) BROE inherently uses the bounded-delay resource-supply model, which approximates the periodic processor supply to a component linearly. This linear approximation may give a pessimistic estimation of the required resources when the deadlines of tasks are heavily constrained; in this situation, even HSRP can sometimes outperform BROE.
2) SIRAP's analysis exploits many detailed task characteristics. The disadvantage of this, however, is that an analyzed component – meant to be arbitrated by SIRAP – cannot be reused in an HSF with another global synchronization protocol.

We have solved the latter inconvenience, so that the interface parameters of a component can be computed independently of the global synchronization protocol. Under this abstraction, however, the advantages of the scheduling analysis of SIRAP disappear compared to HSRP and BROE.

Looking at the implementation costs of global synchronization protocols in an real-time operating system (Paper D), we considered two major sources that impact their performance: (*i*) the underlying server model and (*ii*) the cost of the primitives for synchronizing the local and global schedulers. BROE scores poorly at the first metric, because it comes with its own server which is considerably more complicated than a periodic, idling server. Both SIRAP and HSRP are independent of an underlying server policy. SIRAP scores poorly at the second metric, because it requires to prevent other local tasks from executing during the inserted idle times, i.e., those tasks with a preemption level lower than the local resource ceiling must be suspended. This is different for BROE and HSRP, because they re-arrange the supplied server budgets, so that local arbitration of tasks can be left to the local scheduler without further modifications.

Re-arranging the supplied server budgets, as is done by BROE and HSRP, might be considered harmful for the global predictability of the system. Here SIRAP is the preferred protocol. Fortunately, there exist simplifications that also allow efficient local scheduling in combination with global resource arbitration through SIRAP. Despite the negative impact on the schedule of a component, for several reasons it can also be a good idea to disable local preemptions during global resource access. For SIRAP's idle-time insertion, this means the expensive task suspensions and resumptions can be replaced by just a spin lock. Furthermore, the resource holding times of a component become more predictable if critical sections cannot be preempted locally (Paper E).

As we have seen, there are many different system parameters that influence the performance of a global synchronization protocol. It is therefore complicated to choose a global synchronization protocol appropriately, even for system integrators with comprehensive knowledge about real-time resource-sharing aspects. For this reason, we have made the APIs of these protocols transparent (see Paper D), so that a component developer can be unaware of (*i*) the scope of resource sharing (local versus global) and (*ii*) the actual protocol being used for arbitrating access to a shared resource. The additional advantage of transparent APIs is that a component developer cannot use protocol-specific exploits. This mitigates the integration of a component in an HSF, because the schedule of the component itself is more predictable.

For the enforcement of reservations of components on non-preemptive resources, we presented an SRP extension (Paper E), called hierarchical synchronization with temporal protection (HSTP). The main design objective of our protocol is to limit the propagation of temporal faults to independent component in case a task exceeds its specified worst-case critical-section length. To achieve this, HSTP dynamically places preemption points in an over-running critical section to enforce the maximum contiguous holding time of each resource.

One way or an other, HSTP must limit the local sources of unpredictable execution that may impact the resource holding times. HSTP therefore prohibits preemptions by other tasks within a component during critical sections. This means that we cannot have multiple tasks from the same component handling interrupts. It is possible, however, to bypass the latter limitation by means of multi-level reservations, for example, as has been done by Parmer et al. [91] and de Niz et al. [108]. Their approaches did not apply HSTP's idea of placing preemption points in an over-running critical section, i.e., contrary to [91] and [108], HSTP allows a critical section to consume its component's allocated budget entirely and in a controlled manner.

Finally, we recall that enforcement of global temporal isolation of independent components (in the presence of other dependent components) comes into play, only if tasks violate the timing interface of the constituting component. This may have a major consequence straddling the local schedulability of the misbehaving component. That is, in case a non-preemptive resource is shared with an other component, the blocking times can be unbounded for all the tasks in any of the resource-sharing components. In such situations, additional policies [111], e.g., roll-back or roll-forward of a misbehaving task, are needed to enforce progress inside those resource-sharing components. Secondly, the implementation costs of the timer mechanisms that are required to monitor the consumed resources by a component on each of its accessed resources can be large (Paper F). These two reasons indicate that resource sharing between components should be avoided when possible; it should be considered with care otherwise. In particular, it stresses the importance that resource sharing is abstracted appropriately at the level of local development and local analysis, so that it is easier during composition to judge the risks of global resource sharing on the predictability of the system.

5 Future work

We have focused in this work on the composition and scheduling of tasks under the objective of minimizing the required processor resources. Firstly, we consider future research directions for further optimizing the resource requirements of components that need to share one processor. Secondly, we look at emerging challenges arising from the needs to share resources between independently analyzed and certified components of different criticality. Finally, we briefly look forward to the impact of broadening the scope of our proposed methods from a uni-processor platform to a homogeneous multi-processor platform.

5.1 Optimizing the number of systems scheduled successfully

In this work we have presented two techniques for maximizing the number of systems that can be scheduled without deadline misses:

1) at the component level using limited-preemption techniques (Paper A);
2) at the system level using two-level preemptive scheduling and resource arbitration using the SRP (Paper C).

The first technique explores the choices of selectively disabling local preemptions of a component which has the entire processor at its disposal. Alternatively, it can be seen as finding the configuration of a task set that allows for the largest scheduling delays, i.e., the longest critical sections. The second technique considers global resource sharing under preemptive two-level scheduling. This can be seen as finding the configuration of a system that allows for the slowest processor, i.e., the minimal *system load*. The second technique does not only optimize the number of systems that can be scheduled successfully, the returned system configuration also imposes the smallest possible system load.

In an HSF, the interface of the component captures the reservations on all the resources required by the component, including the processor. In this work, optimizing the schedulability of components in an HSF has not been considered for tasks that are locally scheduled with limited-preemption techniques. In other words, the analysis of a task arbitrated by the SRP takes no advantage of an increased preemption level. Furthermore, we assumed that all the resources used by the tasks of a component have one-unit capacity.

Finally, the period of a component determines a deadline for the completion of the critical sections of its local tasks. The same component period also determines the sizes of the reserved quanta, i.e., contributing to the overall processor bandwidth required to schedule the system (which determines the system load). The period of a component therefore also impacts the success of meeting deadlines.

In the remainder of this section, we look more carefully to these three limitations of our work.

Optimality of limited-preemptive schedules

As we have seen in Section 2.2, the analysis of a fixed-priority preemptive schedule is pessimistic for modeling limited-preemptive execution of tasks, because preemptive models do not take into account the acceleration of a task after it has started its execution; only the disadvantage of blocking is taken into account. We have considered a variety of limited-preemptive fixed-priority schedulers that tighten the component's analysis. The model with deferred preemptions (FPDS) seems the easiest to analyze and it allows to analyze tasks with non-preemptive sub-jobs. For FPDS, in particular, Davis and Bertogna [112] have recently shown where to place the preemption points in the tasks and how to assign the optimal priorities to tasks accordingly. Their algorithms meet all deadlines of a task set, if this is possible by FPDS. For other limited preemptive scheduling models that are based on preemption thresholds, we have assumed in our work that priorities of tasks are given.

The optimization algorithm by Davis and Bertogna [112] is inapplicable to preemption-threshold schemes in general. Under FPDS sub-jobs are by definition non-preemptive. Contrary to FPDS, where the priorities of tasks and the placement of preemption points are of importance for the schedulability of a task set, for more general limited-preemptive scheduling schemes we also need to determine the optimal preemption thresholds of the sub-jobs of tasks. Even for the most basic preemption-threshold scheduling scheme – as proposed by Wang and Saksena [45] – only exponential algorithms are known for determining priorities of tasks and determining thresholds of tasks optimally [112]. It would be interesting to see how much processor resources can be gained by limited preemptive scheduling with an optimal assignment of priorities compared to the deadline-monotonic assignment of priorities that we have used in our experiments.

Moreover, we have only considered limited-preemptive scheduling for a single component that has the entire processor at its disposal. It would be interesting to apply limited-preemptive scheduling to tasks in an hierarchical system. Especially, the optimizations of two-level fixed-priority HSFs with budget overruns are interesting. Since overruns happen at the end of a server budget and a server can be viewed as a task on itself, limited-preemptive techniques can be used to tighten the global analysis (as we have done in [100]). The problem of placing preemption points within the tasks of different components of an HSF in an optimal manner has not been studied yet. Moreover, it would be interesting to see how the sizes of limited-preemptive sub-jobs of tasks within the entire system need to be reconfigured when components enter or leave the system.

Multi-unit resources

One may make duplicates of resources, for example, using the algorithm proposed by Baruah [32], in order to reduce the blocking between tasks or between components. A smaller blocking term increases the schedulability of the system. However, the capacity of the resource (number of available units) may still be constrained compared to the number of tasks requiring the same resource, especially in open environments where components can dynamically enter or leave the system. In such cases, resource arbitration may have to deal with multi-unit resources.

One way of looking at multi-unit resources is to consider each unit as a one-unit resource on itself. This view can be directly applied to our current algorithms for optimizing the system load in HSFs. It is not guaranteed that our algorithms return an optimal selection of interfaces for components (in terms of the system load), because statically allocating the available units of a resource to the accessing tasks is sub-optimal in terms of schedulability.

Fortunately, the SRP [34] supports dynamic arbitration of multi-unit resources. The use of multi-unit resources requires to adapt the way of computing the preemption levels of tasks and it changes the number of possible resource ceilings accordingly. The increase in the number of possible resource ceilings relates polynomially to the number of available units of a resource. Despite the polynomial bounds on the expansion of the system's dimension, computing the smallest capacity per resource such that the system load of an HSF is minimized is a difficult and intractable problem [113].

We have therefore presented a selection procedures for interfaces of components sharing just one-unit resources, so that the selected interfaces minimize the system load. To demonstrate the algorithmic complexity of optimizing the usage of multi-unit resources, let us start with resources of one-unit capacity that need to be shared globally (like our current method requires). Our selection procedure is based on a finite set of interface candidates sorted by a non-decreasing processor requirement. As a result, the only way to reduce the system load is by decreasing the global blocking between components in each iteration of the optimization procedure. In our current method, the only way to reduce the global blocking is by means of reducing the resource holding times. Reducing the resource holding times can only be achieved by increasing the local blocking. This results never in a smaller budget, so that the interface candidates of a component are indeed traversed in a non-decreasing sorted order of the component's processor requirement. Our procedure therefore converges in a polynomial number of steps towards an optimal selection of interfaces for all the components in the HSF.

An alternative way of decreasing the global blocking between components could be by means of increasing the capacity of the resource that causes the largest blocking. This alternative is problematic for the convergence of the system load, because the availability of one extra unit of a resource results also in less local blocking between tasks of the resource-sharing components. The required budget of those components may therefore decrease and, thus, the iterations for selecting interface candidates follow no longer a non-decreasing processor requirement. Hence, it requires further study to optimize the system load of components accessing multi-unit resources.

Selecting component periods

Integrating resource-sharing components into the HSF requires that components cannot be preempted at arbitrary moments in time by arbitrary other components. To be able to guarantee the timeliness of other components, the period of a component determines the deadline for the completion of a critical section. In the presence of shared non-preemptive resources, the selection of P_s is therefore constrained by the resource holding time.

It is already hard to determine a proper period for independent components in an HSF [60]. In the presence of shared resources, the same period serves as an extra deadline for each of the tasks for the completion of an access to a shared resource. The selection of just one period for the arbitration of all the resources required by a set of tasks seems to be a natural choice, even when a component has the entire platform at its disposal. For example, Thiele and Ernst [114] derive one period for the tasks in a pipeline that share resources from the throughput constraints of a component and this period determines the blocking tolerance of those tasks. However, the model by Thiele and Ernst [114] considers only local resource sharing and each task can only access a single shared resource; they do not consider global resource sharing. We can support global resource sharing by means of extending the model in [114] with our notion of an opaque component design. The challenge then is to derive one common period across all the groups of tasks that share resources.

Similarly, when tasks execute on a cluster of homogeneous processors, literature suggests that a single period for the allocation of all the reserved quanta on each of the processors of the cluster is a good choice [115]. However, sharing of more resources than just processor(s) between different components may require larger quanta of reservations. The allocated processor quanta must be large enough to complete the entire resource holding times of a component and this requirement applies to each of the processors that can be used by this component. In any system holds that the larger the reserved quanta are, the more over-allocations of the processor bandwidth.

5.2 Mixed-criticality systems

Many industrial standards have been developed over the years to promote the exchange and the integration of software components from multiple vendors. For example, a consortium of automotive suppliers and manufacturers has developed the AUTOSAR standard. Many of these standards, however, including AUTOSAR, lack explicit information about timing requirements in their meta-model.

Instead, the functional-safety standards and regulations often describe classes of risk and the corresponding necessary safety requirements for achieving an acceptable residual risk. For example, the standard DO 178B specifies 5 classes of risk for aeronautic systems. In November 2011, the ISO 26262 functional safety standard for automotive systems has been published. ISO 26262 specifies so-called Automotive Safety Integrity Levels (ASILs). The standard provides requirements, applicable throughout the life cycle of all automotive electronic systems, for validation and confirmation measures to ensure that a sufficient and acceptable level of safety is achieved.

There is an emerging need to certify safety-critical systems according to global standards, e.g., ISO 26262, before these systems enter the market. The real-time systems community has therefore developed novel task models and scheduling policies over the past few years. In the remainder of this section, we look at three important research topics related to the certification of a system. Firstly, we consider scheduling of tasks that have been assigned levels of mixed criticality. Secondly, we look at the composition of components that have mixed criticality levels (with or without shared resources). Finally, we define a new concept, i.e., *resilience*, which aims at graceful recovery from temporal faults.

Modified task model and its (lacking) schedulability results

The seminal model by Vestal [116] requires to assign one criticality level to each (sporadic) task. The higher the criticality level of a task, the higher its estimated WCET. The schedulability condition considers a different WCET for each task at each criticality level. All the WCET values that have an impact on whether a task meets its deadline or not must be of the same criticality level as that required by the task [117].

The classical ways of prioritizing tasks optimally (rate monotonic or deadline monotonic) do not take into account the criticality of a task. Criticality As Priority Assignment (CAPA) guarantees timeliness of the highest criticality task, no matter how lower-criticality tasks behave, but it can be resource inefficient. Vestal's algorithm [116] for assigning fixed priorities to tasks that have different criticality levels has shown to be optimal and equivalent to Audsley's algorithm [118], i.e., both Vestal's and Audsley's algorithm will find a priority assignment if a feasible priority assignment exists [119]. In Vestal's model, EDF is no longer optimal; Baruah and Vestal [117] presented an example of a task set which can be scheduled by fixed-priority preemptive scheduling and cannot be scheduled by EDF. They concluded that the performance of scheduling policies using fixed task priorities and fixed job priorities, like EDF, are incomparable for the scheduling of mixed-criticality tasks.

Criticality scheduling may require an additional run-time mechanism in order to change the scheduling policy immediately upon detection of suspicious task executions, i.e., when a task exceeds its WCET for its level of criticality. De Niz et al. [120] presented a scheme that allows the regular priority assignments and the priorities are reversed according to the relative criticality of a task whenever a high-criticality task (with a low priority) has not completed its execution upon an instant where it has zero slack. The criticality levels of tasks are different from the common notion of system modes. A survey on mode-change protocols can be found in [121]. Contrary to, for example, the zero-slack mechanism by de Niz et al. [120], changing of modes typically entails a non-negligible transition latency.

Also blocking of mixed-criticality tasks, e.g., due to resource sharing or limited preemptive scheduling, may delay the required prompt intervention. To the best of our knowledge, blocking is not considered in the current literature on scheduling of mixed-criticality tasks. It would be interesting to take into account limited preemptions in the model of mixed-criticality systems. Next, it would be challenging to investigate the relative performance of limited-preemptive execution by EDF and by fixed-priority scheduling of mixed-criticality tasks.

Composition of real-time components

In this work, we have chosen the tasks as the preferred unit of composition for real-time systems, i.e., a set of tasks of a component is represented as if it is a task on itself. The most complex situation is when each individual task can have different criticality levels with respect to tasks in other components. The problem of scheduling tasks with mixed criticality within a system composed of a hierarchy of schedulers has not been considered yet.

In the first place, we could take a simplified model. It would be interesting to look at situations where the criticality of a task is expressed relative to the tasks in the same component only. Then, each component expresses one level of criticality relative to other components at the global level of an HSF.

To support resource sharing between tasks of different components, it would be interesting to extend the zero-slack mechanism by de Niz et al. [120] with the notion of blocking. Our basic SRP modification, called HSTP, prevents unbounded priority inversion for independent components in the presence of other resource-sharing components. It implements a solution for this purpose for relatively tightly coupled components with the same criticality. HSTP is not adequate for resources shared among components with different criticality, since it still allows a low-criticality component to disrupt a high-criticality one. In a mixed-criticality system different recovery mechanisms are required, e.g., to terminate a low-criticality component that does not return the resource in time for a high-criticality component and to restore the state of that resource.

Robust and resilient scheduling

The current scheduling models for tasks with mixed criticality levels allow a deadline miss of a low-criticality task when a high-criticality task is in danger of missing its deadline. Nevertheless, the model is based on giving hard guarantees to the highest criticality tasks. This meets the expectations of building a robust system. Robustness defines how well a system is prepared against propagation of timing faults.

For the scheduling of tasks in non-hierarchical systems, Davis and Burns [122] proposed a well-defined measure of robustness. They defined a non-decreasing function of time that characterizes the worst-case overhead from unpredictable sources that a task may experience without missing its deadline[11]. Davis and Burns [122] accordingly defined an algorithm for assigning priorities to tasks in such a way that the overall measure of the robustness-functions is maximized. We believe that robustness alone is not sufficient in highly critical systems.

Another measure would be how well a system can deal with timing faults. We use the term *resilience* to define the degree in which a system can recover readily from the adversities caused by timing faults. In highly critical systems the assumption is that bad things will happen, no matter how well the system is designed and tested. The traditional notion of robustness cannot be used directly, because robustness is typically measured by a non-decreasing function of time. For example, it does not support probabilistic models. Following de A. Lima and Burns [123], the term resilience is already used in the real-time systems community to give a measure of how many faults (including a cost model for recovery from a fault) a task set can absorb without missing a deadline. The resilience in these works comes from the mechanisms to recover from functional faults in the system. From a timing perspective, the execution costs of the recovery mechanisms – assuming a maximum frequency of their occurrence [123] – can be captured in a robustness function.

To the best of our knowledge, it has not been investigated how to mitigate the propagation of deadline misses (or other timing faults) between the tasks of a real-time component. The chance of yet an other deadline miss is therefore unknown for the specific scenario after a deadline miss of a task has occurred.

11. Resource-supply models express also the maximum amount of unavailable processor time a task set may experience without missing any deadlines. However, each of these models uses a function with a pre-defined shape.

Whenever a deadline miss happens, novel graceful mechanisms are required for recovery from that temporal fault, so that the system can return into an execution state with a predictable workload upper bound. Our new notion of resilience now refers to the time it takes to achieve this. It might be necessary to intervene in the schedule to accelerate such recovery, for example, by suspending low-criticality tasks temporally or by re-allocating budgets from low-criticality components. This can further improve the resilience of a real-time system.

A simple heuristic to improve the resilience of a system is to over-allocate resources, so that the available slack can be used to recover from timing faults. Reservation-based scheduling on itself just prevents propagation of temporal faults to other independent components, thus, increasing just the robustness. However, the resource-supply models that we used for the composition of components are inherently pessimistic, because they refrain from any assumptions on the actual interference caused by other components in the system. It would be interesting to see how the available slack, obtained from the pessimism in those models, can be exploited to improve the resilience of a system.

Moreover, it would be interesting to extend our measure of resilience to resource sharing in HSFs. The resource-holding times define over-estimated reservations on non-preemptive resources. Our notion of opaque analysis abstracts global resource arbitration and local resource arbitration, so that different synchronization protocols can be compared at the same level of abstraction. In our current work (Paper E and Paper F), we did not evaluate our protocol, HSTP, or other synchronization protocols in terms of quality metrics like resilience and robustness. It would be interesting to investigate the implementations of several synchronization protocols, e.g., BWI, HSRP, SIRAP or BROE, in terms of resilience.

5.3 Multi-processor scheduling

In this work we considered a hierarchical composition of real-time components upon a single processor. Each component abstracts a group of real-time tasks as if it was a single task on itself. The hierarchy of schedulers forms a tree, because each component receives resources from one parent through a single supply function. The result is that a component virtually executes on a uni-processor albeit possibly at a lower speed.

There are several models that allow modeling of multiple parallel functions of resource supplies to components [124]. In these models, components can be composed in a directed a-cyclic graph. The result is that a component virtually executes on a multi-processor platform, where each supply function models a virtual processor with a certain speed. There are different ways to describe the interface of a component under a supply by parallel processors. We are searching for the same abstraction for the composition of components on a multi-processor platform as we have already obtained on a uni-processor platform, i.e., in terms of processor requirements and in terms of non-preemptive resources.

In the remainder of this section, we first briefly summarize the recent advances of composition of components on multi-processor systems with and without resource sharing. Finally, we look at topics of future work on multi-processor systems that relate to our research on uni-processor systems.

Composition using parallel supply functions

The most general model of the parallel supply function (PSF) of a component by Bini et al. [125] improves their initial PSF model presented in [124]. The PSF [125] is an extension of the single processor supply bound function, $sbf(t)$, described by a set of m functions Y_k. The function $Y_k(t)$ is the minimum amount of resources provided in any interval of length t by at most k virtual processors. A component is schedulable on a given platform with m parallel processor reservations, if for every interval of length t there exists a $k \leq m$ such that the component requirement in any interval of length t does not exceed $Y_k(t)$.

In the multi-processor periodic resource (MPR) model by Shin et al. [115], all m supply functions are implemented by periodic reservations, where all functions have the same period P and all functions share a cumulative budget Q. The exact budget of a virtual processor on itself is irrelevant, as long as the sum of all budgets is Q. A disadvantage [62] of the MPR model by Shin et al. [115] is that a component may not be guaranteed on any arbitrary platform that satisfies the interface description of the periodic PSF, if the periods of the parallel supply functions are unaligned. As a consequence, implementing a virtual platform using the MPR model requires time synchronization between the virtual processors.

In order to lift this limitation, Lipari and Bini [62] use a different interface model: the *bounded delay multi-partition* (BDM). The BDM model is an extension of the bounded-delay model by Feng and Mok [58]. A BDM interface is characterized by one initial service delay λ and and by m values of α_k which denote the minimum cumulative utilization with k processors. This means that all the functions $Y_k(t)$ are approximated with linear functions with an initial delay of length λ and a constant slope of α_k.

Composition of components in the presence of shared resources

Following the seminal work of Rajkumar et al. [126] in 1988, many works have considered resource sharing between tasks executing on a multiprocessor platform. Extended versions of the uni-processor synchronization protocols, e.g., the SRP and the PCP, are commonly used for this purpose on multi-processor platforms. Davis and Burns [127] have recently surveyed different policies for scheduling tasks on homogeneous multi-processors, with and without resource sharing. Brandenburg et al. [128] have investigated the relative performance of blocking and non-blocking approaches for tasks that access shared resources. The blocking approaches considered both busy-waiting and suspending; the non-blocking approaches considered lock-free and wait-free. Each approach has its advantages and its disadvantages.

The composition of independently developed components upon a multi-processor platform in the presence of shared resources has recently received some attention [129]. Nemati et al. [129] assume partitioned scheduling of tasks (tasks are statically allocated to one processor) and, in addition, they assume that one processor hosts only one component. Thus, one processor serves as a container and no further reservations are needed for implementing processor isolation between components. Nemati and Nolte [130] also showed how to compute the resource holding times under a more liberal component model where a local static-priority scheduler determines which tasks to execute on the m identical processors. The authors of [129], [130] made the timing analysis of a system compositional. However, task-based locking protocols are used for the synchronization of tasks. It would be interesting to look at various local scheduling policies and to look at different global synchronization protocols.

Open problems

Although the contributions of this work apply to uni-processor systems, it would be interesting to apply them to multi-processor systems. In particular, it would be interesting to look at the following issues.

1) **Opacity:** similar to our notion of opaque analysis for resource sharing between different components, it would be interesting to decouple the local analysis of a component from the global composition of components. This includes the way of accounting the blocking on globally shared resources. Using PSFs, for example, see [125], [124], [62] and [115], the required processor resources of a component can be abstracted regardless of the global scheduling policy. Extending this abstraction with the notion of shared resources requires that the interface of a component should not be influenced by the global synchronization protocol. This extends the independence of a component of the entire global scheduling model, thereby further increasing the re-usability of the component upon multi-processor systems.

2) **Synchronization protocols:** the BDM model by Lipari and Bini [62] seems to be an attractive model to abstract the processor requirements of a component. In the context of the bounded-delay model by Feng and Mok [58] for uni-processor HSFs, BROE performs superior in terms of the system load compared to all other global synchronization protocols. It would therefore be interesting to extend BROE, or another component-level self-blocking technique, to multi-processor systems.

Most existing protocols for resource sharing on multi-processors penalize the tasks that try to access unavailable resources. Alternatively, a BROE-like protocol may temporarily suspend an entire component. A potential advantage may be that the inherent costs for time synchronization between the allocated cluster of processors, i.e., caused by the access of a resource that is shared across processors, can be easily bounded per component.

3) **Temporal isolation:** similar to uni-processor systems, additional mechanism are required in the presence of non-preemptive resources to ensure temporal isolation between components. The implementation overheads of our protocol for uni-processor HSFs, i.e., HSTP, is efficient compared to the implementation of multi-level reservations, because critical sections are execute with local preemptions disabled. No other task than the resource-accessing one can therefore cause a violation of the specified resource-holding time.

Disabling preemptions for short critical sections is often affordable within the compositional models on a uni-processor system [72]. When a component is allocated a cluster of processors, however, disabling all local preemptions during a critical section prohibits any concurrency, even on other processors than the one allocated to the resource-accessing task. We leave it as future work to investigate alternative approaches – and their cost – for confinement of temporal faults during access to shared non-preemptive resources in hierarchical multi-processor systems.

4) **Virtual scheduling:** our technique for virtual scheduling has been developed for uni-processor systems in order to reproduce the schedule of a component under a continuous processor when the same component is mapped on a discontinuous virtual processor. It would be interesting to see how to extend the notion and the meaning of virtual scheduling using PSFs which define cumulative processor speeds. For example, (*i*) a legacy uni-processor component needs to execute on a cluster of virtual processors or (*ii*) a multi-processor component needs to execute on a discontinuous cluster of virtual processors.

Another application of virtual scheduling in the context of multi-processor systems may be for the purpose of reserving shares for components of a single communication bus. For example, a legacy component can be given a dedicated processor. Other components that execute on other processors may block the availability of the bus for our legacy component. The blocking must be bounded to guarantee the timeliness of our legacy component. The increase in speed of the communication buses with peripherals (like the network or the main memory) in computer systems grows much slower than the increase of the (cumulative) processor speeds does. Allocating a non-depletable share of the bus to our legacy component may therefore make the rest of the processors unusable. With virtual scheduling, the maximum jitter tolerance, λ, of our legacy component can be used to define the granularity of a (non-preemptive) bus-transfer in a simple way (which then also applies to other components in the system). It would be interesting to see how virtual scheduling shapes contention on a shared communication bus in a multi-processor system.

References

[1] M. Di Natale and A. Sangiovanni-Vincentelli, “Moving from federated to integrated architectures in automotive: The role of standards, methods and tools,” *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, April 2010.

[2] C. Lu, R. Rajkumar, and E. Tovar, “Guest editorial special section on cyber-physical systems and cooperating objects,” *IEEE Transactions on Industrial Informatics (TII)*, vol. 8, no. 2, p. 378, May 2012.

[3] B. Bonsen, R. Mansvelders, and E. Vermeer, “Integrated vehicle dynamics control using state dependent riccati equations,” in *International Symposium on Advanced Vehicle Control (AVEC)*, August 2010.

[4] C. Lozoya, M. Velasco, and P. Marti, “The one-shot task model for robust real-time embedded control systems,” *IEEE Transactions on Industrial Informatics (TII)*, vol. 4, no. 3, pp. 164–174, August 2008.

[5] G. Lipari and E. Bini, “A methodology for designing hierarchical scheduling systems,” *Journal of Embedded Computing (JEC)*, vol. 1, no. 2, pp. 257–269, 2005.

[6] N. Serreli, G. Lipari, and E. Bini, “The distributed deadline synchronization protocol for real-time systems scheduled by EDF,” in *Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2010, pp. 1–8.

[7] C. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[8] J. Leung and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic, real-time tasks,” *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, December 1982.

[9] A.-L. Mok, “Fundamental design problems of distributed systems for the hard-real-time environment,” PhD thesis, Massachusetts Institute of Technology, May 1983, http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-297.pdf.

[10] S. Baruah, A. Mok, and L. Rosier, “Preemptively scheduling hard-real-time sporadic tasks on one processor,” in *Real-Time Systems Symposium (RTSS)*, Dec 1990, pp. 182–190.

[11] G. Buttazzo, G. Lipari, and L. Abeni, “Elastic task model for adaptive rate control,” in *Real-Time Systems Symposium (RTSS)*, December 1998, pp. 286–295.

[12] A. Mok and D. Chen, “A multiframe model for real-time tasks,” in *Real-Time Systems Symposium (RTSS)*, December 1996, pp. 22–29.

[13] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, “Generalized multiframe tasks,” *Real-Time Systems*, vol. 17, pp. 5–22, 1999.

[14] A. Zuhily, “Scheduling analysis of fixed priority hard real-time systems with multiframe tasks,” Ph.D. dissertation, University of York, York, UK, January 2009, http://www.cs.york.ac.uk/rts/documents/thesis/zuhily09.pdf.

[15] S. Baruah, “A general model for recurring real-time tasks,” in *Real-Time Systems Symposium (RTSS)*, December 1998, pp. 114–122.

[16] A. Burns, “Preemptive priority based scheduling: An appropriate engineering approach,” in *Advances in Real-Time Systems*, S. Son, Ed. Prentice-Hall, 1994, pp. 225–248.

[17] S. K. Baruah, “Dynamic- and static-priority scheduling of recurring real-time tasks,” *Real-Time Systems*, vol. 24, pp. 93–128, 2003.

[18] S. Baruah, “Feasibility analysis of recurring branching tasks,” in *Euromicro Workshop on Real-Time Systems*, June 1998, pp. 138–145.

[19] M. Anand, “Conditional models for compositional design of real-time embedded systems,” Ph.D. dissertation, University of Pennsylvania, May 2008.

[20] M. Stigge, P. Ekberg, N. Guan, and W. Yi, “The digraph real-time task model,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2011, pp. 71–80.

[21] ——, “On the tractability of digraph-based task models,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2011, pp. 162–171.

[22] M. Stigge and W. Yi, “Hardness results for static priority real-time scheduling,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2012, pp. 189–198.

[23] M. Joseph and P. Pandya, “Finding response times in a real-time system,” *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[24] M. Spuri, “Analysis of deadline scheduled real-time systems,” Institut National de Recherche et Informatique et en Automatique (INRIA), France, Tech. Rep. 2772, January 1996.

[25] K. Albers and F. Slomka, “An event stream driven approximation for the analysis of real-time systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2004, pp. 187–195.

[26] N. Fisher and S. Baruah, “A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005, pp. 117–126.

[27] T. H. C. Nguyen, P. Richard, and N. Fisher, “The fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines revisited,” in *International Conference on Real-Time and Network Systems (RTNS)*, November 2010.

[28] F. Eisenbrand and T. Rothvoss, “Static-priority real-time scheduling: Response time computation is np-hard,” in *Real-Time Systems Symposium (RTSS)*, December 2008, pp. 397–406.

[29] P. Richard, G. Kemayo, F. Ridouard, E. Grolleau, and T. H. C. Nguyen, “Response time bounds for static-priority tasks and arbitrary relative deadlines with resource augmentation,” in *Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2012.

[30] K. Lakshmanan and R. Rajkumar, “Scheduling self-suspending real-time tasks with rate-monotonic priorities,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2010, pp. 3–12.

[31] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” in *Real-Time Systems Symposium (RTSS)*, December 1989, pp. 166–171.

[32] S. K. Baruah, “Resource sharing in EDF-scheduled systems: A closer look,” in *Real-Time Systems Symposium (RTSS)*, 2006, pp. 379–387.

[33] E. W. Dijkstra, “Cooperating sequential processes,” Tech. Rep. EWD-123, 1965.

[34] T. Baker, “Stack-based scheduling of realtime processes,” *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, March 1991.

[35] B. W. Lampson and D. D. Redell, “Experience with processes and monitors in Mesa,” *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, February 1980.

[36] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronisation,” *IEEE Transactions on Computers (TC)*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[37] G. Buttazzo, *Hard real-time computing systems - predictable scheduling algorithms and applications (2^{nd} edition)*. Springer, 2005.

[38] M. Spuri, “Earliest deadline scheduling in real-time systems,” Ph.D. dissertation, Scuola Superiore Sant’Anna, Pisa, Italy, 1995.

[39] M.-I. Chen and K.-J. Lin, “Dynamic priority ceilings: A concurrency control protocol for real-time systems,” *Real-Time Systems*, vol. 2, pp. 325–346, 1990.

[40] P. Gai, G. Lipari, and M. Di Natale, “Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip,” in *Real-Time Systems Symposium (RTSS)*, December 2001, pp. 73–83.

[41] J. J. Labrosse, *MicroC/OS-II: the real-time kernel (2nd edition)*. CMP Books, 2002.

[42] D. Zöbel, P. Polock, and A. van Arkel, “Testing for the conformance of real-time protocols implemented by operating systems,” in *International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Electronic Notes in Theoretical Computer Science, Vol. 133, May 2005*, September 2004, pp. 315–332.

[43] M. Bertogna and S. Baruah, “Limited preemption EDF scheduling of sporadic task systems,” *IEEE Transactions on Industrial Informatics (TII)*, vol. 6, no. 4, pp. 579–591, November 2010.

[44] G. C. Buttazzo, M. Bertogna, and G. Yao, “Limited preemptive scheduling for real-time systems. a survey,” *IEEE Transactions on Industrial Informatics (TII)*, vol. 9, no. 1, pp. 3–15, February 2013.

[45] Y. Wang and M. Saksena, “Scheduling fixed-priority tasks with preemption threshold,” in *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, December 1999, pp. 328–335.

[46] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005, pp. 137–144.

[47] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, no. 1-3, pp. 63–119, August 2009.

[48] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Real-Time Systems Symposium (RTSS)*, December 2011, pp. 251–260.

[49] U. Keskin, R.J. Bril, and J.J. Lukkien, "Exact response-time analysis for fixed-priority preemption-threshold scheduling," in *Conference on Emerging Technologies and Factory Automation (ETFA), Work-in-progress (WiP) session*, September 2010, pp. 1–4.

[50] G. Yao and G. Buttazzo, "Reducing stack with intra-task threshold priorities in real-time systems," in *Conference on Embedded Software (EMSOFT)*, October 2010, pp. 109–118.

[51] V. Lortz and K. Shin, "Semaphore queue priority assignment for real-time multiprocessor synchronization," *IEEE Transactions on Software Engineering (TSE)*, vol. 21, no. 10, pp. 834 – 844, October 1995.

[52] Z. Deng and J.-S. Liu, "Scheduling real-time applications in open environment," in *Real-Time Systems Symposium (RTSS)*, December 1997, pp. 308–319.

[53] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Real-Time Systems Symposium (RTSS)*, December 2003, pp. 25–36.

[54] L. Almeida and P. Peidreiras, "Scheduling with temporal partitions: response-time analysis and server design," in *Conference on Embedded Software (EMSOFT)*, September 2004, pp. 95–103.

[55] W. Wang, A. K. Mok, and G. Fohler, "Pre-scheduling," *Real-time Systems*, vol. 30, pp. 83–103, 2005.

[56] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *Real-Time Systems Symposium (RTSS)*, December 2007, pp. 129–138.

[57] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–39, 2008.

[58] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in *Real-Time Systems Symposium (RTSS)*, December 2002, pp. 26–35.

[59] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Real-Time Systems Symposium (RTSS)*, December 2004, pp. 57–67.

[60] A. Easwaran, "Advances in hierarchical real-time systems: Incrementality, optimality, and multiprocessor clustering," Ph.D. dissertation, University of Pennsylvania, Pennsylvania, USA, 2008.

[61] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 611–624, October 1998.

[62] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation," in *Real-Time Systems Symposium (RTSS)*, December 2010, pp. 249–258.

[63] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *International Symposium on Circuits and Systems (ISCAS)*, vol. 4, May 2000, pp. 101–104.

[64] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Real-Time Systems Symposium (RTSS)*, December 2005, pp. 389–398.

[65] R. J. Bril, W. F. J. Verhaegh, and C. C. Wüst, "A cognac-glass algorithm for conditionally guaranteed budgets," in *Real-Time Systems Symposium (RTSS)*, December 2006, pp. 388–397.

[66] P. Balbastre, I. Ripoll, and A. Crespo, "Exact response time analysis of hierarchical fixed-priority scheduling," in *Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2009, pp. 315–320.

[67] E. Wandeler and L. Thiele, "Real-time interfaces for interface-based design of real-time

systems with fixed priority scheduling," in *Conference on Embedded Software (EMSOFT)*, 2005, pp. 80–89.

[68] F. Dewan and N. Fisher, "Efficient admission control for enforcing arbitrary real-time demand-curve interfaces," in *Real-Time Systems Symposium (RTSS)*, December 2012, p. (to appear).

[69] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems*, vol. 9, no. 1, pp. 31–67, 1995.

[70] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *International Parallel and Distributed Processing Symposium (IPDPS)*, March 2007, pp. 1–8.

[71] ——, "Resource holding times: computation and optimization," *Real-Time Systems*, vol. 41, no. 2, pp. 87–117, 2009.

[72] ——, "Resource-sharing servers for open environments," *IEEE Transactions on Industrial Informatics (TII)*, vol. 5, no. 3, pp. 202–219, August 2009.

[73] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *IEEE Transactions on Industrial Informatics (TII)*, vol. 6, no. 1, pp. 93 –104, February 2010.

[74] M. González Harbour and J.C. Palencia, "Response time analysis for tasks scheduled under EDF within fixed priorities," in *Real-Time Systems Symposium (RTSS)*, December 2003, pp. 200–209.

[75] C. Mercer, S. Savage, and H. Tokuda, "Processor capability reserves: Operating system support for multimedia applications," in *International Conference on Multimedia Computing and Systems (ICMCS)*, May 1994, pp. 90–99.

[76] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Real-Time Systems Symposium (RTSS)*, December 1987, pp. 261–270.

[77] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, June 1989.

[78] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers (TC)*, vol. 44, no. 1, pp. 73–91, January 1995.

[79] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein, "Analysis of hierarchical fixed-priority scheduling," in *Euromicro Conference on Real-Time Systems (ECRTS)*, June 2002, pp. 152–160.

[80] M. Stanovich, T. Baker, A.-I. Wang, and M. González Harbour, "Defects of the POSIX sporadic server and how to correct them," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2010, pp. 35–45.

[81] J. Liu, *Real-Time Systems.* Prentice Hall, 2000.

[82] P. Kumar, J.-J. Chen, and L. Thiele, "Demand bound server: Generalized resource reservation for hard real-time systems," in *International Conference on Embedded Software (EMSOFT)*, October 2011, pp. 233–242.

[83] P. Kumar, J.-J. Chen, L. Thiele, A. Schranzhofer, and G. Buttazzo, "Real-time analysis of servers for general job arrivals," in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2011, pp. 251–258.

[84] N. Fisher and F. Dewan, "A bandwidth allocation scheme for compositional real-time systems with periodic resources," *Real-Time Systems*, vol. 48, no. 3, pp. 223–263, 2012.

[85] F. Dewan and N. Fisher, "Approximate bandwidth allocation for fixed-priority-scheduled periodic resources," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2010, pp. 247–256.

[86] M. M. H. P. van den Heuvel, P. J. L. Cuijpers, J. J. Lukkien, and N. W. Fisher, "Revised budget allocations for fixed-priority-scheduled periodic resources," Eindhoven University of Technology, Tech. Rep. CS-12-03, February 2012.

[87] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 164–177.

[88] D. Kim, Y.-H. Lee, and M. Younis, "Spirit-μkernel for strongly partitioned real-time systems," in *Conference on Real-Time Computing Systems and Applications (RTCSA)*, December 2000, pp. 73–80.

[89] A. Crespo, I. Ripoll, and M. Masmano, “Partitioned embedded architecture based on hypervisor: The XtratuM approach,” in *European Dependable Computing Conference (EDCC)*, April 2010, pp. 67–72.
[90] VMware, “Timekeeping in VMware virtual machines - VMware ESX 4.0/ESXI 4.0, VMware Workstation 7.0,” *Information guide*, May 2010.
[91] G. Parmer and R. West, “Predictable interrupt management and scheduling in the Composite component-based system,” in *Real-Time Systems Symposium (RTSS)*, December 2008, pp. 232–243.
[92] Y. Zhang and R. West, “Process-aware interrupt scheduling and accounting,” in *Real-Time Systems Symposium (RTSS)*, December 2006, pp. 191–201.
[93] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, “Run-time services for hybrid cpu/fpga systems on chip,” in *Real-Time Systems Symposium (RTSS)*, December 2006, pp. 3–12.
[94] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi, “Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005, pp. 98–105.
[95] J. Regehr and U. Duongsaa, “Preventing interrupt overload,” in *Conference on Languages, compilers, and tools for embedded systems (LCTES)*, 2005, pp. 50–58.
[96] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, “Towards hierarchical scheduling on top of VxWorks,” in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2008.
[97] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar, “Support for hierarchical scheduling in FreeRTOS,” in *Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2011.
[98] M. Holenderski, “Multi-resource management in embedded real-time systems,” Ph.D. dissertation, Eindhoven University of technology, October 2012.
[99] K.-J. Lin and Y.-C. Wang, “The design and implementation of real-time schedulers in RED-linux,” *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1114–1130, July 2003.
[100] U. Keskin, M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien, M. Behnam, and T. Nolte, “An engineering approach to synchronization based on overrun for compositional real-time systems,” in *Symposium on Industrial Embedded Systems (SIES)*, June 2011.
[101] M. Behnam, T. Nolte, and R. J. Bril, “Tighter schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks,” in *Conference on Engineering of Complex Computer Systems*, April 2011.
[102] U. Steinberg, J. Wolter, and H. Härtig, “Fast component interaction for real-time systems,” in *Euromicro Conference on Real-Time Systems*, July 2005, pp. 89–97.
[103] G. Lipari, G. Lamastra, and L. Abeni, “Task synchronization in reservation-based real-time systems,” *IEEE Transactions on Computers (TC)*, vol. 53, no. 12, pp. 1591–1601, December 2004.
[104] M. Behnam, “Synchronization protocols for a compositional real-time scheduling framework,” Ph.D. dissertation, Mälardalen University, November 2010.
[105] R. Davis and A. Burns, “Resource sharing in hierarchical fixed priority pre-emptive systems,” in *Real-Time Systems Symposium (RTSS)*, 2006, pp. 257–267.
[106] M. Behnam, I. Shin, T. Nolte, and M. Nolin, “SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems,” in *Conference on Embedded Software (EMSOFT)*, October 2007, pp. 279–288.
[107] M. Behnam, T. Nolte, and R. J. Bril, “Bounding the number of self-blocking occurrences of SIRAP,” in *Real-Time Systems Symposium (RTSS)*, December 2010.
[108] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, “Resource sharing in reservation-based systems,” in *Real-Time Systems Symposium (RTSS)*, December 2001, pp. 171–180.
[109] R. Santos, G. Lipari, and J. Santos, “Improving the schedulability of soft real-time open dynamic systems: The inheritor is actually a debtor,” *Journal of Systems and Software*, vol. 81, pp. 1093–1104, July 2008.
[110] L. Abeni and G. Buttazzo, “Integrating multimedia applications in hard real-time systems,” in *Real-Time Systems Symposium (RTSS)*, December 1998, pp. 4–13.

[111] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.
[112] R. I. Davis and M. Bertogna, “Optimal fixed priority scheduling with deferred pre-emption,” in *Real-Time Systems Symposium (RTSS)*, December 2012, p. (to appear).
[113] J.-J. Chen and S. Chakraborty, “Partitioned packing and scheduling for sporadic real-time tasks in identical multiprocessor systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2012, pp. 24–33.
[114] D. Thiele and R. Ernst, “Optimizing performance analysis for synchronous dataflow graphs with shared resources,” in *Design, Automation Test in Europe Conference (DATE)*, March 2012, pp. 635–640.
[115] I. Shin, A. Easwaran, and I. Lee, “Hierarchical scheduling framework for virtual clustering of multiprocessors,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2008, pp. 181–190.
[116] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *Real-Time Systems Symposium (RTSS)*, December 2007, pp. 239–243.
[117] S. Baruah and S. Vestal, “Schedulability analysis of sporadic tasks with multiple criticality specifications,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2008, pp. 147–155.
[118] N. Audsley, “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times,” University of York, UK, Tech. Rep. YCS-164, December 1991.
[119] F. Dorin, P. Richard, M. Richard, and J. Goossens, “Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities,” *Real-Time Systems*, vol. 46, pp. 305–331, 2010.
[120] D. de Niz, K. Lakshmanan, and R. Rajkumar, “On the scheduling of mixed-criticality real-time task sets,” *Real-Time Systems Symposium (RTSS)*, pp. 291–300, December 2009.
[121] J. Real and A. Crespo, “Mode change protocols for real-time systems: a survey and a new protocol,” *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, March 2004.
[122] R. Davis and A. Burns, “Robust priority assignment for fixed priority real-time systems,” in *Real-Time Systems Symposium (RTSS)*, December 2007, pp. 3–14.
[123] G. M. de A. Lima and A. Burns, “An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems,” *IEEE Transactions on Computers (TC)*, vol. 52, no. 10, pp. 1332–1346, October 2003.
[124] E. Bini, G. Buttazzo, and M. Bertogna, “The multi supply function abstraction for multiprocessors,” in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2009, pp. 294–302.
[125] E. Bini, M. Bertogna, and S. Baruah, “Virtual multiprocessor platforms: Specification and use,” in *Real-Time Systems Symposium (RTSS)*, December 2009, pp. 437–446.
[126] R. Rajkumar, L. Sha, and J. Lehoczky, “Real-time synchronization protocols for multiprocessors,” in *Real-Time Systems Symposium (RTSS)*, December 1988, pp. 259–269.
[127] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, pp. 1–44, October 2011.
[128] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson, “Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2008, pp. 342–353.
[129] F. Nemati, M. Behnam, and T. Nolte, “Independently-developed real-time systems on multicores with shared resources,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2011, pp. 251–261.
[130] F. Nemati and T. Nolte, “Resource hold times under multiprocessor static-priority global scheduling,” in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2011, pp. 197–206.

Part II

Included papers

PAPER A:

GENERALIZED FIXED-PRIORITY SCHEDULING WITH LIMITED PRE-EMPTIONS

R.J. Bril, M.M.H.P. van den Heuvel, U. Keskin and J.J. Lukkien

In proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS), pp. 209–220, July 2012

ABSTRACT

Fixed-priority scheduling with deferred preemption (FPDS) and fixed-priority scheduling with preemption thresholds (FPTS) have been proposed in the literature as viable alternatives to fixed-priority preemptive scheduling (FPPS), that reduce memory requirements, reduce the cost of arbitrary preemptions, and may improve the feasibility of a task set even when preemption overheads are neglected.

This paper aims at advancing the relative strength of limited-preemptive schedulers by combining FPDS and FPTS. In particular, we present a refinement of FPDS with preemption thresholds for both jobs and sub-jobs, termed FPGS. We provide an exact schedulability analysis for FPGS, and show how to maximize the feasibility of a set of sporadic tasks under FPGS for given priorities, computation times, periods, and deadlines of tasks. We evaluate the effectiveness of FPGS by comparing the feasibility of task sets under FPGS with other fixed-priority scheduling algorithms by means of a simulation. Our experiments show that FPGS allows an increase of the number of task sets that are schedulable under fixed-priority scheduling.

1 INTRODUCTION

1.1 Background and motivation

Based on the seminal paper of Liu and Layland [1], many results have been achieved in the area of analysis for fixed-priority preemptive scheduling (FPPS). Arbitrary preemption of real-time tasks has a number of drawbacks, though. Because all tasks may be active at the same time, the memory has to be dimensioned for the sum of the worst-case memory requirements of the tasks. In systems using cache memory, e.g. to bridge the speed gap between processors and main memory, arbitrary preemptions induce additional cache flushes and reloads. Although fixed-priority non-preemptive scheduling (FPNS) may resolve these problems, it generally leads to reduced schedulability compared to FPPS [2]. Therefore, two main limited-preemptive schemes have been proposed between the extremes of arbitrary preemption and no preemption: fixed-priority scheduling with deferred preemption (FPDS) or co-operative scheduling [3, 4] and fixed-priority scheduling with preemption thresholds (FPTS) [5, 6, 7, 8].

These two schemes are based on orthogonal refinements of the scheduling model for FPPS, and as a consequence, neither of the two schemes generalizes the other. For FPDS, each job (or activation) of a task is assumed to consist of a sequence of sub-jobs, where sub-jobs are non-preemptable. When a job is executing, it can only be preempted between consecutive sub-jobs, i.e. at so-called *preemption points*. For FPTS, each job of a task has a so-called *preemption threshold* next to a priority. When a job is executing, it can only be preempted by jobs with a priority higher than its preemption threshold. The reduction of the memory requirements for FPDS and FPTS are addressed in [9] and [6, 10], respectively. The reduction of the cost of arbitrary preemptions for FPDS has been addressed in [11, 12, 4, 3, 13]. The improvement of the feasibility of a task set has been addressed for FPDS in [4, 14] and for FPTS in [6, 14]. Because each scheme has its own relative strengths and weaknesses, we seek to combine both schemes into a single scheme, which we term generalized fixed-priority scheduling (FPGS). In this paper, we focus on the improvement of the feasibility that can be achieved by FPGS.

1.2 Contributions

This paper presents four major contributions. Firstly, we present FPGS, a refinement of FPDS with preemption thresholds for both jobs and sub-jobs. This novel scheme generalizes existing fixed-priority scheduling (FPS) algorithms, such as FPNS, FPPS, and FPTS. Secondly, we provide and prove an exact schedulability analysis for FPGS. Because FPGS generalizes existing FPS algorithms, its analysis specializes to the analysis for any of the other algorithms. Thirdly, we show how to maximize schedulability of a set of sporadic tasks under FPGS for given priorities, computation times, periods and deadlines of tasks by determining optimal preemption thresholds for jobs and sub-jobs, and lengths of final sub-jobs of tasks. Our approach for FPGS is inspired by and refines to the approach presented in [14]. Fourthly, we evaluate the schedulability of

FPGS compared to other FPS algorithms by means of a simulation. Our experiments show that FPGS allows an increase of the number of task sets that are schedulable under FPS.

1.3 Overview

The remainder of the paper is organized as follows. In Section 2, a scheduling model for FPGS is presented. Because FPGS refines FPDS, we briefly recapitulate the worst-case response time analysis for FPDS in Section 3. Section 4 presents the worst-case response time analysis for FPGS. Section 5 describes how to maximize the schedulability of a set of tasks under FPGS. The effectiveness of FPGS to improve the feasibility is evaluated in Section 6. This paper is concluded in Section 7.

2 REAL-TIME SCHEDULING MODEL

This section starts with a basic, continuous scheduling model for FPPS, i.e. we assume time to be taken from the real domain ($\mathbb{R}$), similar to, e.g. [15, 4, 14]. We subsequently refine this basic model for FPTS [5] and FPDS [4]. Next, our generalized model for FPGS is presented. The section is concluded with remarks, including an example.

2.1 Basic model for FPPS

We assume a single processor and a set $\mathcal{T}$ of n independent, sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$, with unique priorities $\pi_1, \pi_2, \ldots, \pi_n$. At any moment in time, the processor is used to execute the highest priority task that has work pending. For notational convenience, we assume that (*i*) tasks are given in order of decreasing priorities, i.e. τ_1 has highest priority and τ_n has lowest priority, and (*ii*) a higher priority is represented by a higher value, i.e. $\pi_1 > \pi_2 > \ldots > \pi_n$.

Each task τ_i is characterized by a *minimal inter-arrival time* $T_i \in \mathbb{R}^+$, a *worst-case computation time* $C_i \in \mathbb{R}^+$, and a (*relative*) *deadline* $D_i \in \mathbb{R}^+$. The deadline D_i may be smaller than, equal to, or larger than the period T_i. A release of a task is also termed a *job*. The first job arrives at an arbitrary time.

We also adopt standard basic assumptions [1], i.e. tasks do not suspend themselves, a job of a task does not start before its previous job is completed, and the overhead of context switching and task scheduling is ignored.

2.2 Refined model for FPTS

In FPTS, each task τ_i has a *preemption threshold* θ_i, where $\pi_1 \geq \theta_i \geq \pi_i$. When τ_i is executing, it can only be preempted by tasks with a priority higher than θ_i. Note that we have FPPS and FPNS as special cases when $\forall_{1 \leq i \leq n} \theta_i = \pi_i$ and $\forall_{1 \leq i \leq n} \theta_i = \pi_1$, respectively.

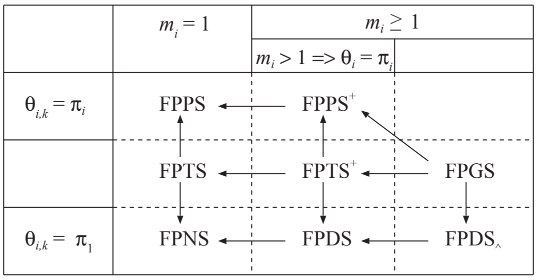

Figure 1. A generalization graph for fixed-priority scheduling algorithms, assuming (1). The equations in the boxes denote additional constraints on m_i, θ_i (columns), and $\theta_{i,k}$ (rows) for FPPS, FPTS, and FPDS. The other algorithms, i.e. FPPS$^+$, FPTS$^+$, and FPDS$_\wedge$, follow from these additional constraints.

2.3 Refined model for FPDS

In FPDS, each job of task τ_i consists of a sequence of m_i sub-jobs, where $m_i \geq 1$. The k^{th} sub-job of τ_i is denoted by $\tau_{i,k}$ and characterized by a worst-case computation time $C_{i,k} \in \mathbb{R}^+$, where $C_i = \sum_{k=1}^{m_i} C_{i,k}$. Sub-jobs are non-preemptable. Hence, a task can only be preempted between consecutive sub-jobs, i.e. at so-called *preemption points*. Note that we have FPNS as special case when $\forall_{1 \leq i \leq n} m_i = 1$. Further note that FPTS (or FPPS) is not a special case of FPDS nor vice versa.

2.4 Generalized model for FPGS

We now present a model for FPGS, that generalizes both FPTS and FPDS. Similar to FPTS, we assume that each task τ_i has a preemption threshold θ_i, which serves as a *minimum* preemption threshold for τ_i in our generalized model. Similar to FPDS, we assume that each task τ_i consists of a sequence of m_i sub-jobs, where consecutive sub-jobs are separated by preemption points. We now define a preemption threshold for each sub-job.

Each sub-job $\tau_{i,k}$ has a *preemption threshold* $\theta_{i,k}$, where $\pi_1 \geq \theta_{i,k} \geq \theta_i$. When a sub-job $\tau_{i,k}$ is executing, it can only be preempted by tasks with a priority higher than its preemption threshold $\theta_{i,k}$. At a preemption point, a task τ_i can only be preempted by tasks with a priority higher than θ_i. The preemption threshold $\theta_{i,k}$ therefore allows the threshold for preemption to be raised for the duration of sub-job $\tau_{i,k}$. When $m_i = 1$, there are no preemption points. To have FPTS as special case, we adopt $\theta_{i,1} = \theta_i$ for $m_i = 1$. In summary:

$$\forall_{1 \leq i \leq n} \left((m_i = 1 \Rightarrow \theta_{i,1} = \theta_i) \wedge \forall_{1 \leq k \leq m_i} \pi_1 \geq \theta_{i,k} \geq \theta_i \geq \pi_i \right). \tag{1}$$

We have FPTS as special case when $\forall_{1 \leq i \leq n} m_i = 1$. Similarly, we have FPDS as special case when

$$\underset{1 \leq i \leq n}{\forall} \left((m_i > 1 \Rightarrow \theta_i = \pi_i) \wedge \underset{1 \leq k \leq m_i}{\forall} \theta_{i,k} = \pi_1 \right). \tag{2}$$

A generalization graph for fixed-priority scheduling algorithms based on (1) is shown in Figure 1. Given FPGS and the constraints for FPPS, FPTS, and FPDS, the other algorithms are derived. FPPS$^+$ and FPTS$^+$ denote generalizations of FPPS and FPTS, respectively, where each job of task τ_i may consist of a sequence of m_i sub-jobs. For both generalizations, the preemption threshold θ_i is assumed to be equal to π_i when $m_i > 1$, similar to FPDS. FPPS$^+$ models situations where *internal deadlines* [3] play a role.

FPTS$^+$ generalizes FPDS with preemption thresholds at the sub-job level. A description of and analysis for FPTS$^+$ can already be found in [8][1]. A special case of FPTS$^+$ is described in [16][2]. A description of FPTS$^+$ can also be found in [17][3].

FPDS$_\wedge$ generalizes FPDS, where the preemption threshold θ_i of task τ_i in its preemption points may be larger than π_i.

2.5 Concluding remarks

Due to a preemption threshold, a task τ_i can *defer* the preemption and execution of a higher priority task τ_j. As a result, a next job of τ_i may experience a higher interference. We will use the term *blocking* of a task τ_j to denote the time that the execution of τ_j is deferred by lower priority tasks.

As illustrated in Figure 1, FPGS generalizes FPPS, FPTS, FPNS, FPDS, FPPS$^+$, FPTS$^+$, and FPDS$_\wedge$. By an appropriate instantiation, i.e. selection of m_i, θ_i, and $\theta_{i,k}$, the schedulability analysis for FPGS will therefore specialize to the analysis for any of the other algorithms. FPGS improves schedulability compared to existing algorithms, as will be illustrated below.

Example: Consider a set $\mathcal{T}_1$ with four tasks with characteristics given in Table 1. The set is schedulable under FPGS, but not under any other FPS algorithm. Using the optimization algorithms in [5] and [14], the set remains unschedulable under FPTS and FPDS. Using our optimization algorithm, the set remains unschedulable under FPTS$^+$ and FPDS$_\wedge$.

3 Recap of existing analysis for FPDS [4]

In this section, we briefly recapitulate the worst-case response-time analysis for FPDS as described in [4], i.e. for a continuous scheduling model. We start this section with

1. The analysis is only postulated in [8], however, not proven.
2. In [16], all sub-jobs of a task τ_i have the same preemption threshold. Moreover, all sub-jobs of τ_i have the same computation time q_i, with the possible exception of the final sub-job, i.e. $C_{i,m_i} = C_i - (m_i - 1)q_i$.
3. In [17], FPTS$^+$ is used to reduce memory requirements. The schedulability analysis for FPTS$^+$ is based on FPPS, however, and therefore only sufficient.

Table 1
Task set characteristics of $\mathcal{T}_1$ and worst-case response times under FPGS, denoted by WR_i^{G}.

task	T_i	D_i	C_{ik}	π_i	$\theta_i = \theta_{i,1}$	$\theta_{i,2}$	WR_i^{G}
τ_1	16	16	2	4	4	–	16
τ_2	210	210	36 + 14	3	3	4	170
τ_3	435	435	166 + 14	2	2	4	434
τ_4	480	435	86 + 14	1	3	4	434

basic terminology and definitions. Next, we describe the worst-case blocking of tasks under FPDS and recapitulate the notion of ε-critical instant, on which the analysis is based. The worst-case response-time analysis concludes the section.

3.1 Basic terminology and definitions

The response time of job k of task τ_i of the task set $\mathcal{T}$ is denoted by $R_{i,k}$. The *worst-case response-time* WR_i of a task τ_i is defined as the longest response time of its jobs, i.e.

$$WR_i \stackrel{\text{def}}{=} \sup_k R_{i,k}. \tag{3}$$

A task set $\mathcal{T}$ is schedulable when all its tasks meet their deadlines, i.e.

$$\forall_{1 \le i \le n} \; WR_i \le D_i. \tag{4}$$

A *critical instant* of a task is defined to be a (hypothetical) instant that leads to the worst-case response time for that task [1]. The worst-case response time of a task τ_i is found in a so-called level-i active period[4] that starts at a critical instant of τ_i. To define the latter notion, we first define the notion of pending load. The *pending load* $P_i(t)$ is the amount of processing at time t that still needs to be performed for the jobs of tasks with a priority higher than or equal to task τ_i that are released before time t. A *level-i active period* is an interval $[t_s, t_e)$ such that $P_i(t_s) = 0$, $P_i(t_e) = 0$, and $P_i(t) > 0$ for all $t \in (t_s, t_e)$.

The (*worst-case*) *utilization* $U^{\mathcal{T}}$ of $\mathcal{T}$ is the fraction of the processor time spent on the execution of $\mathcal{T}$ [1], i.e.

$$U^{\mathcal{T}} \stackrel{\text{def}}{=} \sum_{1 \le i \le n} \frac{C_i}{T_i}. \tag{5}$$

4. The notion of level-i active period supersedes the notion of level-i busy period [18]; see [4].

3.2 Blocking and ε-critical instant

The *worst-case blocking* $B_i^{\mathrm{D}} \in \mathbb{R}^+ \cup \{0\}$ of task τ_i by a lower priority task is equal to the longest computation time of any sub-job of a task with a priority lower than τ_i. This blocking is given by

$$B_i^{\mathrm{D}} = \max\left(0, \max_{l:\pi_i > \pi_l} \max_{1 \le k \le m_l} C_{l,k}\right). \tag{6}$$

The outermost max in (6) is used to define B_i^{D} for tasks that do not experience blocking, i.e. the lowest priority task τ_n. If (and only if) $B_i^{\mathrm{D}} > 0$ then this blocking time is a *supremum* (and not a maximum) and cannot be assumed, because (*i*) the sub-job of the lower priority task causing the blocking of task τ_i has to start strictly before the activation of τ_i and (*ii*) a continuous scheduling model is considered. Similarly, the critical instant cannot be assumed when $B_i^{\mathrm{D}} > 0$.

Given an infinitesimal time $\varepsilon > 0$, the maximum response time of task τ_i under FPDS is assumed when the level-i active period is started at an ε-*critical instant*, i.e. when τ_i has a simultaneous release with all higher priority tasks and a sub-job of the lower priority tasks with computation time B_i^{D} starts a time ε before that simultaneous release. This maximum response time of task τ_i is then given by the maximum of the response times of the jobs of τ_i in the level-i active period. Because the maximum response time is a strictly increasing function of the blocking time, the worst-case response time of τ_i under FPDS, denoted by WR_i^{D}, is found by letting ε go to zero. This WR_i^{D} is a supremum (and not maximum) when $B_i^{\mathrm{D}} > 0$, i.e. for all tasks except the lowest priority task.

3.3 Worst-case response times

The worst-case length WL_i^{D} of a level-i active period is given by the smallest $x \in \mathbb{R}^+$ that satisfies the following recursive equation

$$x = B_i^{\mathrm{D}} + \sum_{j:\pi_j \ge \pi_i} \left\lceil \frac{x}{T_j} \right\rceil C_j. \tag{7}$$

WL_i^{D} can be found by the following iterative procedure:

$$\begin{cases} WL_i^{(0)} & = & B_i^{\mathrm{D}} + \sum_{j:\pi_j \ge \pi_i} C_j \\ WL_i^{(l+1)} & = & B_i^{\mathrm{D}} + \sum_{j:\pi_j \ge \pi_i} \left\lceil \frac{WL_i^{(l)}}{T_j} \right\rceil C_j, \quad l = 0, 1, \ldots \end{cases}. \tag{8}$$

The procedure is stopped when the same value is found for two successive iterations of l, yielding WL_i^{D}. The procedure is guaranteed to terminate for all i when the utilization $U^{\mathcal{T}}$ of the task set $\mathcal{T}$ is less than one, i.e. $U^{\mathcal{T}} < 1$.

Although WL_i^{D} is a supremum (and not a maximum) when $B_i^{\mathrm{D}} > 0$, the worst-case number wl_i^{D} of jobs of task τ_i in a level-i active period is always a maximum, and given by

$$wl_i^{\mathrm{D}} = \left\lceil \frac{WL_i^{\mathrm{D}}}{T_i} \right\rceil . \tag{9}$$

Assuming a critical instant for task τ_i at time zero with an activation of job zero of τ_i, the worst-case response time WR_i^{D} of τ_i is now given by

$$WR_i^{\mathrm{D}} = \max_{0 \le k < wl_i^{\mathrm{D}}} WR_{i,k}^{\mathrm{D}}, \tag{10}$$

where $WR_{i,k}^{\mathrm{D}}$ denotes the worst-case response time of job k. The latter is given by

$$WR_{i,k}^{\mathrm{D}} = S_{i,k,m_i}^{\mathrm{D}} + C_{i,m_i} - kT_i, \tag{11}$$

where S_{i,k,m_i}^{D} denotes the *worst-case start time* of the last sub-job of job k relative to the start of the level-i active period, given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = \begin{cases} B_i^{\mathrm{D}} + (k+1)C_i - C_{i,m_i} + \sum\limits_{h:\pi_h > \pi_i} \left\lceil \frac{x}{T_h} \right\rceil C_h & \text{for } B_i^{\mathrm{D}} > 0 \\ (k+1)C_i - C_{i,m_i} + \sum\limits_{h:\pi_h > \pi_i} \left(\left\lfloor \frac{x}{T_h} \right\rfloor + 1 \right) C_h & \text{for } B_i^{\mathrm{D}} = 0 \end{cases} . \tag{12}$$

Similar to (7) for WL_i^{D}, the latter recursive equations can be solved by means of an iterative procedure, starting with a lower bound, e.g. $B_i^{\mathrm{D}} + (k+1)C_i - C_{i,m_i}$. Note that the analysis is not uniform, i.e. the analysis for $B_i^{\mathrm{D}} = 0$ (the lowest priority task), differs from the analysis for $B_i^{\mathrm{D}} > 0$ (all other tasks). This anomaly is an immediate consequence of the fact that a continuous scheduling model is assumed.

4 Analysis for FPGS

In this section, we present worst-case response-time analysis for FPGS for a continuous scheduling model.

We start this section with a description of interference and blocking under FPGS. Next, we determine the maximum response time of task τ_i by assuming a minimal amount of time δ that (a sub-job of) a lower priority task needs to start before the activation of task τ_i to cause blocking of τ_i. We subsequently derive the worst-case response time of τ_i by letting δ go to zero. The section is concluded with remarks.

4.1 Interference and blocking

A task τ_i can experience interference from all higher priority tasks, and a job of τ_i can experience additional interference from previous jobs of τ_i. For interference, two cases can be distinguished: (*i*) *delay* of the start of a job and (*ii*) *preemption* during the execution of a job. All tasks with a higher priority can delay the start of a job, and previous jobs of a task can delay the start of a next job. However, a task τ_i can only be preempted by a task τ_h when the priority of τ_h is higher than a preemption threshold of τ_i, i.e. $\pi_h > \theta_{i,k}$ or $\pi_h > \theta_i$.

Due to the limited preemptive scheduling of our model, a task τ_i can be blocked by a lower priority task τ_l when a preemption threshold of τ_l prevents τ_i from preempting τ_l, i.e. when $\theta_l \geq \pi_i$ or $\theta_{l,k} \geq \pi_i$. In the former case, τ_l can block τ_i for at most its worst-case computation time C_l. When $\theta_l < \pi_i$, τ_l can block τ_i for at most the largest $C_{l,k}$ with $\theta_{l,k} \geq \pi_i$ of its sub-jobs. The worst-case blocking time B_i^{G} of task τ_i is therefore given by

$$B_i^{\mathrm{G}} = \max\left(0, \max_{l:\pi_i>\pi_l} (\{C_l \mid \theta_l \geq \pi_i\}, \max_{1\leq k\leq m_l} \{C_{l,k} \mid \theta_{l,k} \geq \pi_i\})\right). \tag{13}$$

The outermost max in (13) is used to define B_i^{G} in situations where there exists no lower priority task with a preemption threshold preventing τ_i to preempt, e.g. for the lowest priority task τ_n. Similar to B_i^{D}, B_i^{G} can not be assumed when $B_i^{\mathrm{G}} > 0$. Hence, this blocking time is the supremum of the blocking times that can actually occur.

4.2 Maximum response times

We now determine the maximum response time $R_i^{\mathrm{G}}(B_i)$ of task τ_i by assuming a maximum blocking time[5] B_i for τ_i, i.e.

$$B_i = \max(0, B_i^{\mathrm{G}} - \delta), \tag{14}$$

where δ denotes an amount of time that a (sub-job of a) lower priority task needs to start before the activation of task τ_i to cause B_i blocking of τ_i. Given this maximum blocking time, we can apply classical real-time theory to determine maximum response times.

The maximum response time $R_i^{\mathrm{G}}(B_i)$ of τ_i under FPGS is assumed when the level-*i* active period is started at a δ-*critical instant*, i.e. when τ_i has a simultaneous release with all higher priority tasks and a job or sub-job of the lower priority tasks with computation time B_i^{G} starts a time δ before that simultaneous release. A level-*i* active period assumes its maximum length $WL_i^{\mathrm{G}}(B_i)$ when started at a δ-critical instant. This maximum length $WL_i^{\mathrm{G}}(B_i)$ is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = B_i + \sum_{j:\pi_j \geq \pi_i} \left\lceil \frac{x}{T_j} \right\rceil C_j. \tag{15}$$

Similar to WL_i^{D}, $WL_i^{\mathrm{G}}(B_i)$ can be found by an iterative procedure; see (8). The maximum number $wl_i^{\mathrm{G}}(B_i)$ of jobs of task τ_i in a level-*i* active period is given by

$$wl_i^{\mathrm{G}}(B_i) = \left\lceil \frac{WL_i^{\mathrm{G}}(B_i)}{T_i} \right\rceil. \tag{16}$$

5. Although B_i depends on δ, we omit "(δ)" for readability.

The maximum response time $R_i^{\mathrm{G}}(B_i)$ of τ_i is now given by

$$R_i^{\mathrm{G}}(B_i) = \max_{0 \le k < wl_i^{\mathrm{G}}(B_i)} R_{i,k}^{\mathrm{G}}(B_i), \tag{17}$$

where $R_{i,k}^{\mathrm{G}}(B_i)$ denotes the maximum response time of job k, assuming a δ-critical instant at time zero with a release of job 0. Unlike FPDS, where all tasks with a priority higher than π_i can preempt τ_i at preemption points, only tasks with a priority higher than the preemption threshold θ_i of τ_i can preempt τ_i at those points under FPGS. Hence, only those latter tasks can delay the start of the last sub-job τ_{i,m_i} of τ_i. Moreover, the last sub-job τ_{i,m_i} of τ_i can be preempted, by tasks with a priority higher than the preemption threshold θ_{i,m_i} of τ_{i,m_i}.

Our analysis for FPGS therefore consists of four stages. We first determine the maximum start-time $S_{i,k}^{\mathrm{G}}(B_i)$ of job k with $0 \le k < wl_i^{\mathrm{G}}(B_i)$ of task τ_i. Next, we determine the maximum start-time $S_{i,k,m_i}^{\mathrm{G}}(B_i)$ of the last sub-job of job k of τ_i. We subsequently determine the maximum finalization time $F_{i,k}^{\mathrm{G}}(B_i)$ of job k of τ_i. Finally, we determine the maximum response time $R_{i,k}^{\mathrm{G}}(B_i)$ of job k of τ_i.

4.2.1 Maximum start-time of a job

Lemma 1: The maximum start-time $S_{i,k}^{\mathrm{G}}(B_i)$ of job k with $0 \le k < wl_i^{\mathrm{G}}(B_i)$ of a task τ_i relative to the start of the level-i active period is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = B_i + kC_i + \sum_{h:\pi_h > \pi_i} \left(\left\lfloor \frac{x}{T_h} \right\rfloor + 1 \right) C_h. \tag{18}$$

Proof: When job k of task τ_i can start its execution at time $S_{i,k}^{\mathrm{G}}(B_i)$, (*i*) the lower priority task causing the blocking B_i has completed its job or sub-job, (*ii*) all earlier jobs of τ_i have completed at or before $S_{i,k}^{\mathrm{G}}(B_i)$, (*iii*) all higher priority tasks that are released before $S_{i,k}^{\mathrm{G}}(B_i)$ have completed at or before $S_{i,k}^{\mathrm{G}}(B_i)$, and (*iv*) no higher priority task is released at $S_{i,k}^{\mathrm{G}}(B_i)$. The start-time $S_{i,k}^{\mathrm{G}}(B_i)$ of job k of τ_i is therefore given by the smallest $x \in \mathbb{R}^+$ satisfying (18). Note that case (*iv*) gives rise to the “floor + 1” (rather than a “ceiling”) term in (18). □

4.2.2 Maximum start-time of last sub-job of a job

Lemma 2: The maximum start time $S_{i,k,m_i}^{\mathrm{G}}(B_i)$ of the last sub-job of job k with $0 \le k < wl_i^{\mathrm{G}}(B_i)$ of a task τ_i with $m_i > 1$ relative to the start of the level-i active period is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = S_{i,k}^{\mathrm{G}}(B_i) + C_i - C_{i,m_i} + \sum_{h:\pi_h > \theta_i} \left(\left\lfloor \frac{x}{T_h} \right\rfloor - \left\lfloor \frac{S_{i,k}^{\mathrm{G}}(B_i)}{T_h} \right\rfloor \right) C_h. \tag{19}$$

Proof: We consider the cases $m_i = 1$ and $m_i > 1$ separately. For $m_i = 1$, the start-time of the last sub-job of a job is equal to the start-time of that job. We therefore have to show that $S^{\mathrm{G}}_{i,k}(B_i)$ is the smallest positive solution of (19). To that end, we first observe that the maximum start-time of the last sub-job of a job is at least equal to the maximum start-time of that job, i.e. $S^{\mathrm{G}}_{i,k,m_i}(B_i) \geq S^{\mathrm{G}}_{i,k}(B_i)$ for $m_i \geq 1$. Next, $S^{\mathrm{G}}_{i,k}(B_i)$ is a solution of (19), which follows immediately by substituting $S^{\mathrm{G}}_{i,k}(B_i)$ for x in (19) and the fact that $C_i = C_{i,m_i}$ for $m_i = 1$. Hence, $S^{\mathrm{G}}_{i,k}(B_i)$ is the smallest positive solution of (19).

Now consider the case $m_i > 1$. When the final sub-job of job k of τ_i with $m_i > 1$ can start its execution at time $S^{\mathrm{G}}_{i,k,m_i}(B_i)$, (*i*) all earlier jobs have completed at or before $S^{\mathrm{G}}_{i,k}(B_i)$, which is strictly smaller than $S^{\mathrm{G}}_{i,k,m_i}(B_i)$, (*ii*) all earlier sub-jobs of job k with a cumulative computation time of $C_i - C_{i,m_i} > 0$ have completed at or before $S^{\mathrm{G}}_{i,k,m_i}(B_i)$, (*iii*) all tasks with a priority higher than θ_i that are released before $S^{\mathrm{G}}_{i,k,m_i}(B_i)$ have completed at or before $S^{\mathrm{G}}_{i,k,m_i}(B_i)$, and (*iv*) no task with a priority higher than θ_i is released at $S^{\mathrm{G}}_{i,k,m_i}(B_i)$. The start-time $S^{\mathrm{G}}_{i,k,m_i}(B_i)$ of the final sub-job of job k of τ_i is therefore given by the smallest $x \in \mathbb{R}^+$ satisfying (19). Note that case (*iv*) gives rise to a term $\sum_{h:\pi_h>\theta_i}\left(\left\lfloor \frac{x}{T_h} \right\rfloor + 1\right) C_h$. Further note that the summation in (19) includes a term $-\left\lfloor \frac{S^{\mathrm{G}}_{i,k}(B_i)}{T_h} \right\rfloor$ rather than $+1$ to prevent that activations of higher priority tasks in the initial part of the interval of length $S^{\mathrm{G}}_{i,k}(B_i)$ are accounted for twice. □

4.2.3 Maximum finalization time of a job

Lemma 3: The maximum finalization time $F^{\mathrm{G}}_{i,k}(B_i)$ of job k with $0 \leq k < wl^{\mathrm{G}}_i(B_i)$ of a task τ_i relative to the start of the level-i active period is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = S^{\mathrm{G}}_{i,k,m_i}(B_i) + C_{i,m_i} + \sum_{h:\pi_h>\theta_{i,m_i}} \left(\left\lceil \frac{x}{T_h} \right\rceil - \left(\left\lfloor \frac{S^{\mathrm{G}}_{i,k,m_i}(B_i)}{T_h} \right\rfloor + 1 \right) \right) C_h. \tag{20}$$

Proof: The last sub-job τ_{i,m_i} of a job can only be preempted by tasks with a higher priority than the preemption threshold θ_{i,m_i} of τ_{i,m_i}. For a maximum blocking time B_i, the maximum finalization-time $F^{\mathrm{G}}_{i,k}(B_i)$ of job k of τ_i is therefore given by the smallest $x \in \mathbb{R}^+$ satisfying (20). Note that similar to (19) the summation term prevents activations of higher priority tasks in the initial interval of length $S^{\mathrm{G}}_{i,k,m_i}(B_i)$ to be accounted for twice. □

4.2.4 Maximum response-time of a job

The maximum response time $R^{\mathrm{G}}_{i,k}(B_i)$ of job $\tau_{i,k}$ can be expressed in terms of its maximum finalization time $F^{\mathrm{G}}_{i,k}(B_i)$ relative to the start of the level-i active period, i.e.

$$R^{\mathrm{G}}_{i,k}(B_i) = F^{\mathrm{G}}_{i,k}(B_i) - kT_i. \tag{21}$$

Theorem 4: The maximum response time $R_i^{\mathrm{G}}(B_i)$ of a task τ_i under FPGS is given by

$$R_i^{\mathrm{G}}(B_i) = \max_{0 \le k < wl_i^{\mathrm{G}}(B_i)} (F_{i,k}^{\mathrm{G}}(B_i) - kT_i). \tag{22}$$

Proof: Follows immediately from (17), (21), and Lemma 3. □

4.3 Worst-case response times

We observe that the maximum length $WL_i^{\mathrm{G}}(B_i)$, the maximum start times $S_{i,k}^{\mathrm{G}}(B_i)$ and $S_{i,k,m_i}^{\mathrm{G}}(B_i)$, the maximum finalization time $F_{i,k}^{\mathrm{G}}(B_i)$ and the maximum response time $R_{i,k}^{\mathrm{G}}(B_i)$ are all strictly increasing functions of B_i. To determine worst-case response times under FPGS, we therefore let δ go to zero (i.e. $\delta \downarrow 0$), effectively letting B_i go to B_i^{G} (i.e. $B_i \uparrow B_i^{\mathrm{G}}$). As a result, the critical instant can not be assumed when $B_i^{\mathrm{G}} > 0$, similar to FPDS.

In the remainder of this section, we first consider the worst-case length of a level-i active period. We subsequently address the same four stages as in the previous section, i.e. worst-case start-time $S_{i,k}^{\mathrm{G}}$ of job k of τ_i, worst-case start-time S_{i,k,m_i}^{G} of the last sub-job of job k of τ_i, worst-case finalization-time $F_{i,k}^{\mathrm{G}}$ of job k of τ_i, and finally the worst-case response time $WR_{i,k}^{\mathrm{G}}$ of job k of τ_i and the worst-case response time WR_i^{G} of τ_i.

Lemma 5: The worst-case length WL_i^{G} of a level-i active period is given by the smallest $x \in \mathbb{R}^+$ that satisfies the following recursive equation

$$x = B_i^{\mathrm{G}} + \sum_{j:\pi_j \ge \pi_i} \left\lceil \frac{x}{T_j} \right\rceil C_j. \tag{23}$$

Proof: Similar to the proof of the length WL_i^{D} for FPDS; see Lemma 5 in [4]. □
WL_i^{G} can be determined by an iterative procedure, similar to WL_i^{D}; see (8). Similar to WL_i^{D}, WL_i^{G} is a supremum (and not a maximum) when $B_i^{\mathrm{G}} > 0$. The worst-case number of jobs in a level-i active period is a maximum, and given by

$$wl_i^{\mathrm{G}} = \left\lceil \frac{WL_i^{\mathrm{G}}}{T_i} \right\rceil. \tag{24}$$

4.3.1 Worst-case start-time of a job

Lemma 6: The worst-case start-time $S_{i,k}^{\mathrm{G}}$ of job k with $0 \le k < wl_i^{\mathrm{G}}$ of a task τ_i relative to the start of the level-i active period is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = \begin{cases} B_i^{\mathrm{G}} + kC_i + \sum_{h:\pi_h > \pi_i} \left\lceil \frac{x}{T_h} \right\rceil C_h & \text{for } B_i^{\mathrm{G}} > 0 \\ kC_i + \sum_{h:\pi_h > \pi_i} \left(\left\lfloor \frac{x}{T_h} \right\rfloor + 1 \right) C_h & \text{for } B_i^{\mathrm{G}} = 0 \end{cases}. \tag{25}$$

Proof: We consider $B_i^{\mathrm{G}} > 0$ and $B_i^{\mathrm{G}} = 0$ separately.

$\{B_i^{\mathrm{G}} = 0\}$ By substituting $B_i = 0$ in (18) we immediately get (25) for this case.

$\{B_i^{\mathrm{G}} > 0\}$ The right-hand side of (18) is a strictly increasing function of B_i and $S_{i,k}^{\mathrm{G}}(B_i)$ is therefore also a strictly increasing function of B_i. The largest value for $S_{i,k}^{\mathrm{G}}(B_i)$ is therefore found for the largest value of $B_i < B_i^{\mathrm{G}}$. Hence, $S_{i,k}^{\mathrm{G}}$ is given by

$$S_{i,k}^{\mathrm{G}} = \lim_{B_i \uparrow B_i^{\mathrm{G}}} S_{i,k}^{\mathrm{G}}(B_i). \tag{26}$$

Given Lemma 12 (see Appendix), we make the following derivation starting from this equation:

$$\begin{aligned}
S_{i,k}^{\mathrm{G}} &= \lim_{B_i \uparrow B_i^{\mathrm{G}}} \left(B_i + kC_i + \sum_{h:\pi_h > \pi_i} \left(\left\lfloor \frac{S_{i,k}^{\mathrm{G}}(B_i)}{T_h} \right\rfloor + 1 \right) C_h \right) \\
&= B_i^{\mathrm{G}} + kC_i + \sum_{h:\pi_h > \pi_i} \lim_{B_i \uparrow B_i^{\mathrm{G}}} \left(\left\lfloor \frac{S_{i,k}^{\mathrm{G}}(B_i)}{T_h} \right\rfloor + 1 \right) C_h \\
&= \{\text{Lemma 12}\} \quad B_i^{\mathrm{G}} + kC_i + \sum_{h:\pi_h > \pi_i} \left\lceil \lim_{B_i \uparrow B_i^{\mathrm{G}}} \frac{S_{i,k}^{\mathrm{G}}(B_i)}{T_h} \right\rceil C_h \\
&= \{(26)\} \quad B_i^{\mathrm{G}} + kC_i + \sum_{h:\pi_h > \pi_i} \left\lceil \frac{S_{i,k}^{\mathrm{G}}}{T_h} \right\rceil C_h.
\end{aligned}$$

Hence, the worst-case start time $S_{i,k}^{\mathrm{G}}$ is the smallest $x \in \mathbb{R}^+$ satisfying (25), which proves the lemma. □

4.3.2 Worst-case start time of the last sub-job of a job

Lemma 7: The worst-case start time S_{i,k,m_i}^{G} of sub-job m_i of job k with $0 \le k < wl_i^{\mathrm{G}}$ of task τ_i relative to the start of the level-i active period is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = \begin{cases} S_{i,k}^{\mathrm{G}} + C_i - C_{i,m_i} + \sum_{h:\pi_h > \theta_i} \left(\left\lceil \frac{x}{T_h} \right\rceil - \left\lceil \frac{S_{i,k}^{\mathrm{G}}}{T_h} \right\rceil \right) C_h & \text{for } B_i^{\mathrm{G}} > 0 \\ S_{i,k}^{\mathrm{G}} + C_i - C_{i,m_i} + \sum_{h:\pi_h > \theta_i} \left(\left\lfloor \frac{x}{T_h} \right\rfloor - \left\lfloor \frac{S_{i,k}^{\mathrm{G}}}{T_h} \right\rfloor \right) C_h & \text{for } B_i^{\mathrm{G}} = 0 \end{cases}. \tag{27}$$

Proof: The derivation of (27) from (19) is similar to the derivation of (25) from (18) by observing that $S_{i,k,m_i}^{\mathrm{G}}(B_i)$ is a strictly increasing function of B_i, describing S_{i,k,m_i}^{G} as

$$S_{i,k,m_i}^{\mathrm{G}} = \lim_{B_i \uparrow B_i^{\mathrm{G}}} S_{i,k,m_i}^{\mathrm{G}}(B_i), \tag{28}$$

and subsequently using Lemma 12. □

4.3.3 Worst-case finalization time of a job

Lemma 8: The worst-case finalization time $F^{\mathrm{G}}_{i,k}$ of job k with $0 \leq k < wl^{\mathrm{G}}_i$ of a task τ_i relative to the start of the level-i active period is given by the smallest $x \in \mathbb{R}^+$ satisfying

$$x = \begin{cases} S^{\mathrm{G}}_{i,k,m_i} + C_{i,m_i} + \sum\limits_{h:\pi_h > \theta_{i,m_i}} \left(\left\lceil \frac{x}{T_h} \right\rceil - \left\lceil \frac{S^{\mathrm{G}}_{i,k,m_i}}{T_h} \right\rceil \right) C_h & \text{for } B^{\mathrm{G}}_i > 0 \\ S^{\mathrm{G}}_{i,k,m_i} + C_{i,m_i} + & \\ \qquad \sum\limits_{h:\pi_h > \theta_{i,m_i}} \left(\left\lceil \frac{x}{T_h} \right\rceil - \left(\left\lfloor \frac{S^{\mathrm{G}}_{i,k,m_i}}{T_h} \right\rfloor + 1 \right) \right) C_h & \text{for } B^{\mathrm{G}}_i = 0 \end{cases}. \tag{29}$$

Proof: We consider $B^{\mathrm{G}}_i > 0$ and $B^{\mathrm{G}}_i = 0$ separately.

$\{B^{\mathrm{G}}_i = 0\}$ We simply substitute $S^{\mathrm{G}}_{i,k,m_i}(B_i)$ in (20) by S^{G}_{i,k,m_i} and we are done.

$\{B^{\mathrm{G}}_i > 0\}$ The right-hand side of (20) is a strictly increasing function of B_i and $F^{\mathrm{G}}_{i,k}(B_i)$ is therefore also a strictly increasing function of B_i. Hence, $F^{\mathrm{G}}_{i,k}$ is given by

$$F^{\mathrm{G}}_{i,k} = \lim_{B_i \uparrow B^{\mathrm{G}}_i} F^{\mathrm{G}}_{i,k}(B_i). \tag{30}$$

Given Lemma 11 (see Appendix), we can make the following derivation starting from this equation:

$$\begin{aligned} F^{\mathrm{G}}_{i,k} &= \lim_{B_i \uparrow B^{\mathrm{G}}_i} \left(S^{\mathrm{G}}_{i,k,m_i}(B_i) + kC_i + \sum_{h:\pi_h > \theta_{i,m_i}} \left(\left\lceil \frac{F^{\mathrm{G}}_{i,k}(B_i)}{T_h} \right\rceil - \right.\right. \\ &\qquad \left.\left. \left(\left\lfloor \frac{S^{\mathrm{G}}_{i,k,m_i}(B_i)}{T_h} \right\rfloor + 1 \right) \right) C_h \right) \\ &= \{(28)\} \quad S^{\mathrm{G}}_{i,k,m_i} + kC_i + \sum_{h:\pi_h > \theta_{i,m_i}} \lim_{B_i \uparrow B^{\mathrm{G}}_i} \left(\left\lceil \frac{F^{\mathrm{G}}_{i,k}(B_i)}{T_h} \right\rceil - \right. \\ &\qquad \left. \left(\left\lfloor \frac{S^{\mathrm{G}}_{i,k,m_i}(B_i)}{T_h} \right\rfloor + 1 \right) \right) C_h \\ &= S^{\mathrm{G}}_{i,k,m_i} + kC_i + \sum_{h:\pi_h > \theta_{i,m_i}} \left(\{\text{Lemma 11}\} \left\lceil \lim_{B_i \uparrow B^{\mathrm{G}}_i} \frac{F^{\mathrm{G}}_{i,k}(B_i)}{T_h} \right\rceil - \right. \\ &\qquad \left. \{\text{Lemma 12}\} \left\lceil \lim_{B_i \uparrow B^{\mathrm{G}}_i} \frac{S^{\mathrm{G}}_{i,k,m_i}(B_i)}{T_h} \right\rceil \right) C_h \\ &= S^{\mathrm{G}}_{i,k,m_i} + kC_i + \sum_{h:\pi_h > \theta_{i,m_i}} \left(\left\lceil \{(30)\} \frac{F^{\mathrm{G}}_{i,k}}{T_h} \right\rceil - \left\lceil \{(28)\} \frac{S^{\mathrm{G}}_{i,k,m_i}}{T_h} \right\rceil \right) C_h. \end{aligned}$$

Hence, the worst-case finalization time $F^{\mathrm{G}}_{i,k}$ is the smallest $x \in \mathbb{R}^+$ satisfying (29), which proves the lemma. □

4.3.4 *Worst-case response time of a job*

The worst-case response time $WR^{\mathrm{G}}_{i,k}$ of job $\tau_{i,k}$ can be expressed in terms of its worst-case finalization time $F^{\mathrm{G}}_{i,k}$ relative to the start of the level-i active period, i.e.

$$WR^{\mathrm{G}}_{i,k} = F^{\mathrm{G}}_{i,k} - kT_i. \tag{31}$$

Theorem 9: The worst-case response time WR^{G}_i of a task τ_i under FPGS is given by

$$WR^{\mathrm{G}}_i = \max_{0 \le k < wl^{\mathrm{G}}_i} (F^{\mathrm{G}}_{i,k} - kT_i). \tag{32}$$

Proof: Similar to Theorem 4. □

4.4 Concluding remarks

In this section, we presented worst-case response-time analysis for FPGS for a continuous scheduling model. We showed that the blocking time, critical instant, worst-case length of a level-i-active period, and worst-case response time for a task are suprema rather than maxima for all tasks, except for those tasks that do not experience blocking by a lower priority task. As a result, our analysis for the worst-case start-time $S^{\mathrm{G}}_{i,k}$ of job k (Lemma 6) and S^{G}_{i,k,m_i} of the last sub-job m_i of job k (Lemma 7), and the worst-case finalization-time $F^{\mathrm{G}}_{i,k}$ of job k (Lemma 8) of task τ_i is non-uniform.

To derive worst-case response times, we first determined maximum response times assuming a minimal amount δ that (a sub-job of) a lower priority task needs to start before the activation of task τ_i to cause blocking of τ_i. We subsequently derived worst-case response times by letting δ go to zero. By assuming $\delta = 1$, the intermediate results presented in Subsection 4.2 provide worst-case response-time analysis for a discrete scheduling model [19], i.e. where task parameters are restricted to integers and scheduling decisions are assumed to be taken at integral moments in time.

We therefore derived recursive equations for both a discrete as well a continuous scheduling model to determine (*i*) the worst-case start time of a job, see (18) in Lemma 1 and (25) in Lemma 6; (*ii*) the worst-case start time of the final sub-job, see (19) in Lemma 2 and (27) in Lemma 7; and (*iii*) the worst-case finalization time of a job, see (20) in Lemma 3 and (29) in Lemma 8. Each of these recursive equations can be solved by an iterative procedure by starting with an appropriate lower bound, similar to, e.g., the recursive equations of the level-i active period in (7) and the worst-case response time under FPDS in (12).

5 Improving Schedulability

In this section, we show how to maximize schedulability of a task set $\mathcal{T}$ under FPGS and its derivative FPS algorithms for given priorities π_i, computation times C_i, periods T_i, and deadlines D_i of all tasks τ_i of $\mathcal{T}$ by determining optimal thresholds θ_i, sub-job thresholds θ_{i,m_i}, and sub-job lengths C_{i,m_i} for each τ_i of $\mathcal{T}$. Our approach for FPGS is similar to the approach for non-preemptive regions presented by Bertogna et al. [14], i.e. we (*i*) convert the recursive response time equations to an exact schedulability test based on execution-request curves, (*ii*) minimize the response time of a task, while respecting so-called *blocking tolerances* [20] of higher priority tasks, and (*iii*) compute blocking tolerances of tasks. Our algorithm for FPGS specializes to algorithms for its derivatives, including the algorithm in [14]. Our algorithm differs significantly from the latter, however, for two reasons. Firstly, we need to select thresholds θ_i and θ_{i,m_i} next to sub-job lengths C_{i,m_i}, and secondly the highest θ_{i,m_i} does not necessarily result in a minimal response time under FPGS.

5.1 Alternative schedulability test for FPGS

Instead of calculating the worst-case start-time $S^{\mathrm{G}}_{i,k}$ of job k of τ_i, the worst-case start-time S^{G}_{i,k,m_i} of its final sub-job, and its worst-case finalization time $F^{\mathrm{G}}_{i,k}$, we formulate a schedulability test based on discontinuation points of cumulative execution-request functions, similar to [14]. The worst-case cumulative execution request $W(\pi,t)$ in an interval $[a,b)$ of length t of all tasks with a priority higher than π is given by

$$W(\pi,t) \overset{\text{def}}{=} \sum_{h:\pi_h>\pi} \left\lceil \frac{t}{T_h} \right\rceil C_h. \tag{33}$$

The modified execution request $W^*(\pi,t)$ in a closed interval of length t is given by

$$W^*(\pi,t) \overset{\text{def}}{=} \sum_{h:\pi_h>\pi} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h. \tag{34}$$

We now use $W(\pi,t)$ and $W^*(\pi,t)$ to define functions $\psi^{(1)}_{i,k}(x)$, $\psi^{(2)}_{i,k}(x,y)$, and $\psi^{(3)}_{i,k}(x,y,z)$, to refer to the right-hand side of the recursive equations (25) for $S^{\mathrm{G}}_{i,k}$, (27) for S^{G}_{i,k,m_i}, and (29) for $F^{\mathrm{G}}_{i,k}$, respectively, (excluding the blocking term), i.e.

$$\psi^{(1)}_{i,k}(x) = k \times C_i + \begin{cases} W(\pi_i,x) & \text{for } B^{\mathrm{G}}_i > 0 \\ W^*(\pi_i,x) & \text{for } B^{\mathrm{G}}_i = 0 \end{cases}, \tag{35}$$

$$\begin{aligned} \psi^{(2)}_{i,k}(x,y) = {} & \psi^{(1)}_{i,k}(x) + C_i - C_{i,m_i} \\ & + \begin{cases} W(\theta_i,y) - W(\theta_i,x) & \text{for } B^{\mathrm{G}}_i > 0 \\ W^*(\theta_i,y) - W^*(\theta_i,x) & \text{for } B^{\mathrm{G}}_i = 0 \end{cases}, \end{aligned} \tag{36}$$

$$\begin{aligned} \psi^{(3)}_{i,k}(x,y,z) = {} & \psi^{(2)}_{i,k}(x,y) + C_{i,m_i} \\ & + \begin{cases} W(\theta_{i,m_i},z) - W(\theta_{i,m_i},y) & \text{for } B^{\mathrm{G}}_i > 0 \\ W(\theta_{i,m_i},z) - W^*(\theta_{i,m_i},y) & \text{for } B^{\mathrm{G}}_i = 0 \end{cases}. \end{aligned} \tag{37}$$

Note that, following (29), we use $W(\theta_{i,m_i},z)$ in (37) and **not** $W^*(\theta_{i,m_i},z)$ for $B^{\mathrm{G}}_i = 0$.

The time points, representing the discontinuation points to be inspected, can be generalized as

$$\Pi_{i,k}(t_l, t_u) = (t_l,\ t_u] \bigcap \{h \times T_j \mid \forall h \in \mathbb{N},\ j \leq i\} \bigcup \{t_u\}, \tag{38}$$

where $\Pi_{i,k}(t_l, t_u)$ is a non-empty (i.e. it contains at least t_u) and finite set of time points. We now define specializations of $\Pi_{i,k}(t_l, t_u)$ for $\psi^{(1)}_{i,k}(x)$, $\psi^{(2)}_{i,k}(x,y)$ and $\psi^{(3)}_{i,k}(x,y,z)$. The set of time points $\Pi^{(1)}_{i,k}$ must contain a time t_1 at which a job has been started (or it exactly starts at t_1), i.e. $S^{\mathrm{G}}_{i,k} \leq t_1$, and is defined as

$$\Pi^{(1)}_{i,k} = \Pi_{i,k}(k \times T_i,\ k \times T_i + D_i - C_i). \tag{39}$$

Note that we always include the upper-bound of the time domain t_u in the set of time points. In special cases, e.g. $C_i = D_i$, this may include time 0 for job $k = 0$.

The set of time points $\Pi^{(2)}_{i,k}$ must contain a time t_2 at which the final sub-job of a job has been started (or it exactly starts at t_2), i.e. $S^{\mathrm{G}}_{i,k,m_i} \leq t_2$, and is defined as

$$\Pi^{(2)}_{i,k}(t_1) = \Pi_{i,k}(t_1,\ k \times T_i + D_i - C_{i,m_i}) \bigcup \{t_1\}. \tag{40}$$

The set of time points $\Pi^{(3)}_{i,k}$ must contain a time t_3 at which a job has been finished (or it exactly finishes at t_3), i.e. $F^{\mathrm{G}}_{i,k} \leq t_3$, and is defined as

$$\Pi^{(3)}_{i,k}(t_2) = \Pi_{i,k}(t_2,\ k \times T_i + D_i) \bigcup \{t_2\}. \tag{41}$$

Note that the inspected time points do not necessarily coincide with the exact worst-case start and finalization times of a (sub-) job. Once we find a time t_1 at which a job has been started, it might therefore be the case that the final sub-job has also been started. Similarly, once we find a time t_2 at which the final sub-job of job k has been started, it might be the case that the job has also finished its execution. Hence, we insert the time point t_1 in $\Pi^{(2)}_{i,k}(t_1)$ and we insert t_2 in $\Pi^{(3)}_{i,k}(t_2)$.

We are now ready to present an exact schedulability test for FPGS using execution-request curves.

Theorem 10: A task set is schedulable under FPGS, if and only if $\forall i\ : 1 \leq i \leq n\ :\ \forall k \in [0, wl^{\mathrm{G}}_i)\ :$

$$\begin{gathered}\Big(\exists t_1 \in \Pi^{(1)}_{i,k} : B^{\mathrm{G}}_i + \psi^{(1)}_{i,k}(t_1) \leq t_1 \bigwedge \\ \Big(\exists t_2 \in \Pi^{(2)}_{i,k}(t_1) : B^{\mathrm{G}}_i + \psi^{(2)}_{i,k}(t_1, t_2) \leq t_2 \bigwedge \\ \Big(\exists t_3 \in \Pi^{(3)}_{i,k}(t_2) : B^{\mathrm{G}}_i + \psi^{(3)}_{i,k}(t_1, t_2, t_3) \leq t_3\Big)\Big)\Big)\end{gathered} \tag{42}$$

and for every task τ_i with $B^{\mathrm{G}}_i = 0$, for each job k there exists a $t_1 \in \Pi^{(1)}_{i,k}$, $t_2 \in \Pi^{(2)}_{i,k}(t_1)$ and $t_3 \in \Pi^{(3)}_{i,k}(t_2)$ such that each of the following conditions holds:

1) $\left.\psi^{(1)}_{i,k}\right|_{(B^{\mathrm{G}}_i>0)}(t_1) < t_1$ or $\left.\psi^{(1)}_{i,k}\right|_{(B^{\mathrm{G}}_i=0)}(t_1) \leq t_1$;
2) $\left.\psi^{(2)}_{i,k}\right|_{(B^{\mathrm{G}}_i>0)}(t_1,t_2) < t_2$ or $\left.\psi^{(2)}_{i,k}\right|_{(B^{\mathrm{G}}_i=0)}(t_1,t_2) \leq t_2$;
3) $\left.\psi^{(3)}_{i,k}\right|_{(B^{\mathrm{G}}_i>0)}(t_1,t_2,t_3) < t_3$ or $\left.\psi^{(3)}_{i,k}\right|_{(B^{\mathrm{G}}_i=0)}(t_1,t_2,t_3) \leq t_3$.

Proof: The only-if direction ($\Leftarrow$) follows directly from Lemma 15, Lemma 16 and Lemma 17; see Appendix B.

If ($\Rightarrow$): Assume (42) is satisfied and there is a task τ_i which misses its deadline. This means that a job k in the level-i active period of τ_i ends at a time $t'_3 > k \times T_i + D_i$. By definition holds that $t'_3 \notin \Pi^{(3)}_{i,k}(t_2)$, so that (42) and condition 3 is falsified. □

5.2 Blocking tolerance

The blocking tolerance β_i of a task τ_i is the maximum amount of blocking that a lower priority task may induce to τ_i without hampering the feasibility of task τ_i. The blocking tolerance for a task τ_i is defined by the minimum blocking tolerance of all its jobs in a level-i active period:

$$\beta_i \quad = \min_{0\leq k< wl^{\mathrm{G}}_i} \left\{\beta_{i,k}\right\}, \tag{43}$$

where $\beta_{i,k}$ is the blocking tolerance of job k of τ_i.

Based on Theorem 10, the schedulability of job k of τ_i can be checked for $B^{\mathrm{G}}_i > 0$ by

$$\exists_{t_1\in\Pi^{(1)}_{i,k},\ t_2\in\Pi^{(2)}_{i,k}(t_1),\ t_3\in\Pi^{(3)}_{i,k}(t_2)} \Big(B^{\mathrm{G}}_i \leq t_1 - \psi^{(1)}_{i,k}(t_1) \wedge \tag{44}$$
$$B^{\mathrm{G}}_i \leq t_2 - \psi^{(2)}_{i,k}(t_1,t_2) \wedge B^{\mathrm{G}}_i \leq t_3 - \psi^{(3)}_{i,k}(t_1,t_2,t_3)\Big).$$

Equation (44) can be rewritten to

$$B^{\mathrm{G}}_i \leq \max_{t_1\in\Pi^{(1)}_{i,k},\ t_2\in\Pi^{(2)}_{i,k}(t_1),\ t_3\in\Pi^{(3)}_{i,k}(t_2)} \{\Phi(t_1,t_2,t_3)\} \tag{45}$$

where

$$\Phi(t_1,t_2,t_3) = \min\left\{\begin{array}{l} t_1 - \psi^{(1)}_{i,k}(t_1), \\ t_2 - \psi^{(2)}_{i,k}(t_1,t_2), \\ t_3 - \psi^{(3)}_{i,k}(t_1,t_2,t_3) \end{array}\right\}. \tag{46}$$

The blocking tolerance $\beta_{i,k}$ of job k of τ_i is now given by

$$\beta_{i,k} = \max_{t_1\in\Pi^{(1)}_{i,k},\ t_2\in\Pi^{(2)}_{i,k}(t_1),\ t_3\in\Pi^{(3)}_{i,k}(t_2)} \{\Phi(t_1,t_2,t_3)\}. \tag{47}$$

The definition for $\beta_{i,k}$ is only correct for a strictly positive blocking tolerance (i.e. $B_i^{\mathrm{G}} > 0$). In case that $\beta_{i,k} < 0$, we know that job k deems the task unschedulable. When $\beta_{i,k} = 0$, then we need to verify whether or not $\beta_{i,k}$ is really equal to 0 using the rules for $B_i^{\mathrm{G}} = 0$ in Theorem 10.

5.3 Minimizing response times

Similar to [5], the highest θ_i yields the minimal response time for a task τ_i under FPGS, because the amount of interference of tasks with a higher priority than τ_i is strictly non-increasing as a function of θ_i; see, e.g. (27). For a task τ_i, we can therefore select the highest θ_i respecting the blocking tolerances of higher priority tasks, i.e.

$$\theta_i = \max(\pi_i, \max(\pi_h : \forall_{j:\pi_h \geq \pi_j > \pi_i} C_i \leq \beta_j)). \tag{48}$$

The highest θ_{i,m_i} does not necessarily lead to a minimal response time, however, as will be illustrated by an example.

Example: Consider a task set $\mathcal{T}_{\mathrm{II}}$ with three tasks $\tau_i = (C_i, T_i, D_i)$: $\tau_1 = (1,7,2)$, $\tau_2 = (8,15,15)$ and $\tau_3 = (6,26,26)$. The blocking tolerance $\beta_1 = 1$, hence $\theta_2 = \pi_2$ using (48). Because task τ_2 is allowed to execute for a duration of $\beta_1 = 1$ time unit at priority level π_1, the optimal choice for $C_{2,m_2} = 1$ and $\theta_{2,m_2} = \pi_1$. The blocking tolerance $\beta_2 = 5$, hence $\theta_3 = \pi_3$ using (48). For task τ_3 we can either choose $C_{3,m_3} = 1$ with $\theta_{3,m_3} = \pi_1$ or we can choose $C_{3,m_3} = 5$ with $\theta_{3,m_3} = \pi_2$. For these two cases, we find $WR_3 = 26$ and $WR_3 = 17$, respectively. Moreover, the resulting blocking tolerances for τ_3 are $\beta_3 = 0$ and $\beta_3 = 3$, respectively.

This example shows that a lower threshold may allow for smaller response times and for a larger length C_{i,m_i}. A higher threshold may reduce the length of C_{i,m_i} due to a small blocking tolerance of (one of) the blocked tasks. Similar to [21], the last sub-job of a task can therefore be seen as a critical region with trade-offs in preemption level and execution length.

5.4 Optimization algorithms

The algorithm for determining the blocking tolerance is similar to the algorithm in [14], but in our case based on Theorem 10.

In Algorithm 1, determining the blocking tolerance of job k of task τ_i, we explicitly distinguish $\Phi|_{(B_i^{\mathrm{G}}>0)}$ and $\Phi|_{(B_i^{\mathrm{G}}=0)}$. This procedure is much more complicated than the one in [14] due to the validation of the predicates in $\Phi|_{(B_i^{\mathrm{G}}>0)}$ and $\Phi|_{(B_i^{\mathrm{G}}=0)}$.

After computing the blocking tolerance for the first job, $\beta_{i,0}$, we have an upper bound for the maximum blocking that a task may suffer. We can therefore use the value $\beta_{i,0}$ to give an upper bound on the worst-case length, see (23). Algorithm 2 simply computes the blocking tolerance β_i by taking the minimum over all jobs to be considered for a task, i.e. according to (43). Apart from the extra input parameters θ_i and θ_{i,m_i}, the procedure in Algorithm 2 is exactly the same as in [14] for FPDS.

Algorithm 1 jobTolerance($\mathcal{T}$, i, k, θ_i, θ_{i,m_i}, C_{i,m_i})

```
1: $\beta_{i,k} \leftarrow \max_{t_1 \in \Pi^{(1)}_{i,k},\ t_2 \in \Pi^{(2)}_{i,k}(t_1),\ t_3 \in \Pi^{(3)}_{i,k}(t_2)} \left\{ \Phi|_{(B^G_i > 0)}(t_1, t_2, t_3) \right\}$;
2: if $\beta_{i,k} = 0$ then
3:     $\beta_{i,k} \leftarrow \max_{t_1 \in \Pi^{(1)}_{i,k},\ t_2 \in \Pi^{(2)}_{i,k}(t_1),\ t_3 \in \Pi^{(3)}_{i,k}(t_2)} \left\{ \Phi|_{(B^G_i = 0)}(t_1, t_2, t_3) \right\}$;
4: end if
5: return $\beta_{i,k}$;
```

Algorithm 2 computeBlockingTolerance($\mathcal{T}$, i, θ_i, θ_{i,m_i}, C_{i,m_i})

```
 1: {Find the blocking tolerance for the first job:}
 2: $\beta_{i,0} \leftarrow$ jobTolerance($\mathcal{T}$, $i$, 0, $\theta_i$, $\theta_{i,m_i}$, $C_{i,m_i}$);
 3: if $\beta_{i,0} < 0$ then return $\beta_{i,0}$
 4: $\beta_i \leftarrow \beta_{i,0}$;
 5: compute $wl_i$ using $B^G_i = \beta_{i,0}$;
 6: {Find the minimum $\beta_{i,k}$ in the level-i active period:}
 7: for $k \leftarrow 1$; $k < wl_i$; $k \leftarrow k+1$ do
 8:     $\beta_{i,k} \leftarrow$ jobTolerance($\mathcal{T}$, $i$, $k$, $\theta_i$, $\theta_{i,m_i}$, $C_{i,m_i}$);
 9:     if $\beta_{i,k} < 0$ then return $\beta_{i,k}$
10:     $\beta_i \leftarrow \min(\beta_i;\ \beta_{i,k})$;
11: end for
12: return $\beta_i$;
```

The algorithm to determine the optimal thresholds, sub-job thresholds, and sub-job length generalizes the one by Bertogna et al. [14]. Essentially, Algorithm 3 loops through the tasks from high to low priority. First, it tries to maximize the threshold θ_i according to Lemma 14 (line 7-11). This limits the search space for an optimal threshold for the final sub-job, θ_{i,m_i}. Given a θ_{i,m_i}, we assign the largest length for C_{i,m_i} guaranteeing schedulability of all the earlier considered, higher priority tasks. The length of C_{i,m_i} is at most equal to the smallest blocking tolerance of any higher priority task with a priority lower than, or equal to, threshold θ_{i,m_i} (line 13-20).

If there does not exist a triple $(\theta_i, \theta_{i,m_i}, C_{i,m_i})$ that yields a non-negative blocking tolerance for task τ_i, then τ_i is infeasible (line 21). When the blocking tolerance is exactly zero, then we know that this limits the search space for valid thresholds for the lower priority tasks (line 23). Finally, line 24 adds the best configuration for task τ_i to the solution space $\mathcal{C}^{\texttt{opt}}$.

Run-time complexity: Algorithm 2 computes the blocking tolerance of a task with the same complexity as the worst-case response-time analysis, i.e. in pseudo-polynomial time. For each task τ_i, Algorithm 3 computes its blocking tolerance for at most i possible preemption thresholds. Hence, Algorithm 3 performs $O(n^2)$ iterations to determine the optimal threshold configuration for a task set $\mathcal{T}$.

5.5 Instantiating optimization algorithms

Algorithm 3 can be easily instantiated to compute optimal configurations for any of the FPS algorithms given in the overview of Figure 1. Below, we give a brief overview:

Algorithm 3 optimalFPGS($\mathcal{T}$)

```
1: $\mathcal{C}^{\texttt{opt}} \leftarrow \emptyset$;
2: $\beta_1 \ldots \beta_n \leftarrow -\infty$
3: $\theta_{i,m_i}^{\texttt{limit}} \leftarrow \pi_1$;
4: for $i \leftarrow 1;\ i \leq n;\ i \leftarrow i+1$ do
5:     {invariant: $\forall h < i$ the values $\beta_h$ have been computed.}
6:     $\theta_{i,m_i}^{\texttt{cur}} \leftarrow \theta_{i,m_i}^{\texttt{limit}}$;
7:     {Find the highest threshold $\theta_i$, where $\pi_1 \geq \theta_i \geq \pi_i$:}
8:     $\theta_i \leftarrow \pi_i$;
9:     for $h \leftarrow i-1;\ 1 \leq h \wedge \beta_h \geq C_i;\ h \leftarrow h-1$ do
10:        $\theta_i \leftarrow \pi_h$;
11:    end for
12:    {Find the best $\theta_{i,m_i}$ and corresponding $C_{i,m_i}$:}
13:    for $\left(\theta_{i,m_i}^{\texttt{cur}} \leftarrow \theta_i;\ \theta_{i,m_i}^{\texttt{limit}} \geq \theta_{i,m_i}^{\texttt{cur}};\ \theta_{i,m_i}^{\texttt{cur}} \leftarrow \theta_{i,m_i}^{\texttt{cur}}+1\right)$; do
14:        $C_{i,m_i} \leftarrow \min\left(C_i, \min\left\{\beta_h \,\middle|\, \theta_{i,m_i}^{\texttt{cur}} \geq \pi_h > \theta_i\right\}\right)$;
15:        $\beta_i^{\texttt{cur}} \leftarrow$ computeBlockingTolerance($\mathcal{T}$, $i$, $\theta_i$, $\theta_{i,m_i}^{\texttt{cur}}$, $C_{i,m_i}$);
16:        if $\beta_i^{\texttt{cur}} \geq \beta_i$ then
17:            $\beta_i \leftarrow \beta_i^{\texttt{cur}}$;
18:            $\theta_{i,m_i} \leftarrow \theta_{i,m_i}^{\texttt{cur}}$;
19:        end if
20:    end for
21:    if $\beta_i < 0$ then return $\emptyset$;
22:    {If $\beta_i = 0$, all next tasks $\tau_l$ are limited to $\pi_i > \theta_{l,m_l} \geq \theta_l$}
23:    if $\beta_i = 0$ then $\theta_{i,m_i}^{\texttt{limit}} \leftarrow \pi_{i+1}$;
24:    $\mathcal{C}^{\texttt{opt}} \leftarrow \mathcal{C}^{\texttt{opt}} \cup \{(\theta_i,\ \theta_{i,m_i},\ C_{i,m_i})\}$;
25: end for
26: return $\mathcal{C}^{\texttt{opt}}$;
```

- **FPTS:** the loop in line 12-20 reduces to a single case where $\theta_{i,m_i} = \theta_i$ and $C_{i,m_i} = C_i$. This is more efficient than [5], because in each iteration they recompute the response times for all tasks τ_j with $\theta_i \geq \pi_j > \pi_i$.
- **FPDS:** the loop in line 7-11 is discarded, i.e. $\theta_i = \pi_i$. The loop in line 12-20 reduces to a single case where $\theta_{i,m_i} = \pi_1$ and C_{i,m_i} is accordingly derived. This yields a similar algorithm as in [14].
- **FPNS:** apply the algorithm for FPDS and check whether or not the assigned values for C_{i,m_i} are equal to C_i.
- **FPTS$^+$:** similar to FPDS, the loop in line 7-11 is discarded, i.e. $\theta_i = \pi_i$.
- **FPDS$_\wedge$:** similar to FPDS, the loop in line 12-20 reduces to a single case where $\theta_{i,m_i} = \pi_1$ and C_{i,m_i} is accordingly derived. This firstly assigns an optimal threshold according to FPTS and then applies a similar algorithm as for FPDS.

6 Simulation results

We perform the same four simulation studies as [14] to compare FPS algorithms under various configurations of thresholds based on the number of schedulable task sets. The results are compared with an optimal scheduling algorithm, i.e. EDF. For each system, we assign deadline monotonic priorities to tasks and task periods T_i are uniformly drawn from the interval $[100, 10.000]$. The individual task utilizations U_i are generated using the UUnifast algorithm [22]. Using the task's utilization U_i and the randomly generated period T_i, we can derive the worst-case execution time C_i of a task τ_i, i.e. $C_i = U_i \times T_i$. For each experiment and for each parameter configuration, we generate a new set of 10.000 systems.

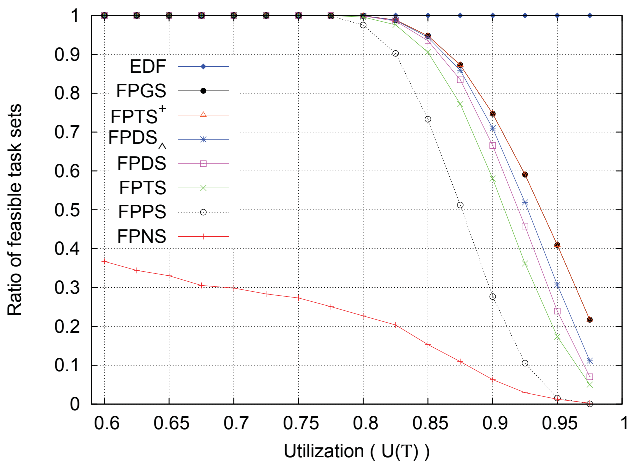

Figure 2. Ratio of feasible task sets versus the task-set's utilization, where the number of tasks is $n = 10$ and $D_i = T_i$.

In the first experiment, a system contains $n = 10$ tasks and we assume that task deadlines are equal to task periods, i.e. $T_i = D_i$. Figure 2 shows the results. It is interesting that FPDS$_\wedge$ gives a noticeable improvement in terms of schedulablity by adding a threshold θ_i to FPDS. The alternative generalization of FPDS, i.e. FPTS$^+$, shows to be superior to FPDS and FPDS$_\wedge$ in all our experiments. Surprisingly, FPGS only shows a negligible gain in terms of schedulability. In each of our experiments, there were between 5 to 15 systems out of the 10.000 generated systems that are only schedulable under FPGS and not by any of the other FPS algorithms.

In the second experiment, we generate task deadlines uniformly drawn from the range $[C_i + 0.5(T_i - C_i);\ T_i]$. Figure 3 shows the results. Under tighter deadlines, FPTS - as proposed by Wang and Saksena [5] - performs worse compared to FPDS relative to the results in Figure 2. This is a direct consequence of the smaller blocking tolerances of tasks, prohibiting to increase the threshold θ_i for the whole duration of C_i. The same effect hits FPDS$_\wedge$. For FPTS$^+$ and FPGS, however, we gain more margin than in the first experiment.

In the third experiment, we vary the range of the task deadlines using parameter α.

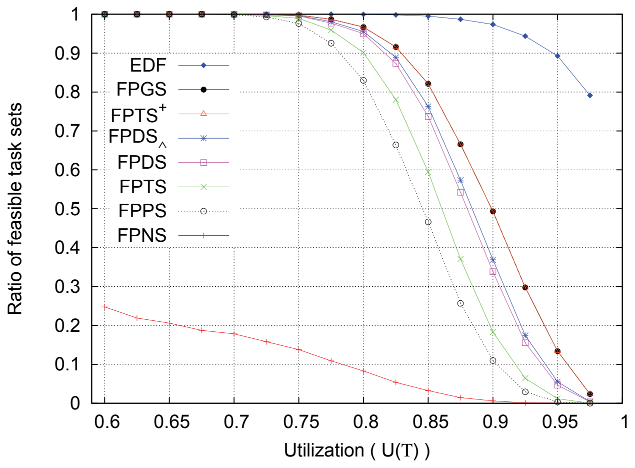

Figure 3. Ratio of feasible task sets versus the task-set's utilization, where the number of tasks is $n = 10$ and $D_i \leq T_i$.

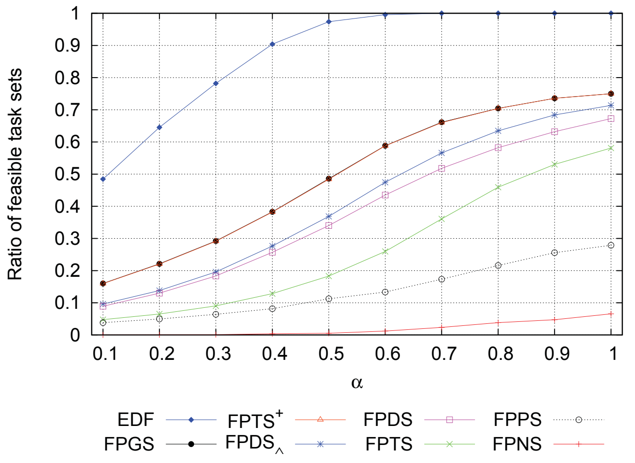

Figure 4. Ratio of feasible task sets versus the deadline distribution α, where the number of tasks is $n = 10$ and the task-set's utilization is $U^{\mathcal{T}} = 0.9$.

We generate task deadline uniformly drawn from the range $[C_i + \alpha(T_i - C_i);\ T_i]$. A low value of α allows tasks to have short deadlines relative to their computation time and $\alpha = 1$ means that deadlines are equal to periods. This experiment confirms the relation between the blocking tolerance and the relative performance of FPTS and FPDS$_\wedge$ to

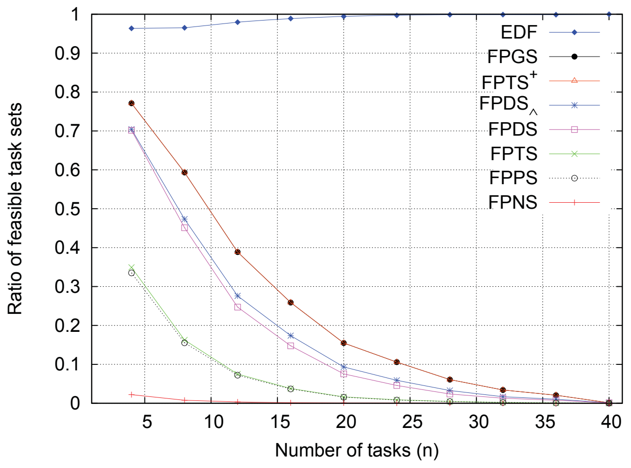

Figure 5. Ratio of feasible task sets versus the number of tasks, where the task-set's utilization is $U^{\mathcal{T}} = 0.9$ and $D_i \leq T_i$.

FPDS. Again, FPTS$^+$ and FPGS show a considerable improvement over FPDS.

In the fourth experiment, we increase the number of tasks within a range of [4, 40] with incremental steps of 4. Similar to [14], we observed that the gain in terms of schedulability becomes smaller for larger task sets. Intuitively, a high number of tasks (with arbitrary periods) which together have a high utilization, decrease the schedulabilty of a system. This is also shown in Figure 5. This differs from the results in [14], because in [14] the authors fix the computation times of task in a relatively small range and uniformly draw computations times rather than periods. As a result, for large task sets they generate many tasks with low utilizations and these tasks all have very large periods in a relatively small range. These task sets are therefore schedulable by almost any scheduling algorithm, even non-preemptive scheduling.

For the same reason, the performance of FPNS is considerable lower in all our experiments compared to the schedulability ratio in [14]. Note, however, that we were able to reproduce the facts in [14] using their method. By further changing the range of the task periods, we only observed small off-sets in the schedulability ratio. We confirm, however, that FPDS - as presented in [14] - indeed improves the schedulability compared to FPPS, FPTS and FPNS. Even more can be gained by our new results for FPDS$_\wedge$, FPTS$^+$ and FPGS.

Also, we considered the constraint $C_{i,m_i} > 0$ for FPDS and FPDS$_\wedge$. Bertogna et al. [14] created a scheduling class - *limited preemptive scheduling* - which unifies FPDS and FPPS when $C_{i,m_i} = 0$. We have evaluated both variants, allowing and disallowing $C_{i,m_i} = 0$, and we observed no difference in the schedulability ratio of FPDS and

FPDS$_\wedge$.

7 Conclusion

We presented FPGS, a generalized limited preemptive FPS scheme that combines scheduling with deferred preemptions (FPDS [3]) and scheduling with preemption thresholds (FPTS [5]). To obtain FPGS, we refined FPDS in two orthogonal dimensions, i.e. preemption thresholds for jobs (FPDS$_\wedge$) and preemption thresholds for sub-jobs (FPTS$^+$). FPGS generalizes existing FPS algorithms, such as fully preemptive scheduling (FPPS), non-preemptive scheduling (FPNS), and the limited-preemptive schemes proposed in [16, 8, 14]. We provided and proved an exact schedulability analysis for FPGS for both a discrete as well as a continuous scheduling model. Because FPGS generalizes existing FPS algorithms, its analysis specializes to the analysis for any of the other algorithms.

For a continuous scheduling model, we showed how to maximize the schedulability of a set of sporadic tasks under FPGS for given priorities, computation times, periods and deadlines of tasks by determining optimal preemption thresholds for jobs and sub-jobs, and length of final sub-jobs of tasks. Our approach for FPGS is inspired by, and refines to, the approach presented in [14]. We evaluated the effectiveness of FPGS by comparing the ratio of feasible task sets under various other FPS algorithms in a number of experiments. Compared to the existing algorithms, FPGS shows a significant gain in schedulability in all our experiments. The same holds for FPTS$^+$ and FPDS$_\wedge$.

Similar to [14], we neglected preemption overheads in our analysis for FPGS, in our approach to maximize schedulability and in our evaluation. With preemption overheads, the advantage of FPTS$^+$ over FPDS$_\wedge$ may be less pronounced, because the former allows preemptions of sub-jobs by tasks with a higher priority whereas the latter doesn't. Incorporating preemption overheads requires further study.

References

[1] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[2] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proc. 12th IEEE Real-Time Systems Symposium (RTSS)*, December 1991, pp. 129–139.

[3] A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," in *Advances in Real-Time Systems*, S. Son, Ed. Prentice-Hall, 1994, pp. 225–248.

[4] R.J. Bril, J. Lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, no. 1-3, pp. 63–119, 2009.

[5] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA)*, December 1999, pp. 328–335.

[6] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, December 2000, pp. 25–34.

[7] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS)*, December 2002, pp. 315–326.

[8] U. Keskin, R.J. Bril, and J.J. Lukkien, “Exact response-time analysis for fixed-priority preemption-threshold scheduling,” in *Proc. IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Work-in-Progress Session*, September 2010.

[9] Koninklijke Philips Electronics, “An enhanced method for handling preemption points,” United States Patent Application Publication US 2007/0022423 A1, January 2007, R.J. Bril and D.J.C. Lowet (inventors).

[10] P. Gai, G. Lipari, and M. Di Natale, “Minimizing memory utilizations of real-time task sets in single and multi-processor systems-on-a-chip,” in *Proc.* 22nd *IEEE Real-Time Systems Symposium (RTSS)*, December 2001, pp. 73–83.

[11] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, “Preemption points placement for sporadic task sets,” in *Proc.* 22th *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2010, pp. 251–260.

[12] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, “Optimal selection of preemption points to minimize preemption overhead,” in *Proc.* 23rd *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2011, pp. 217–227.

[13] G. Yao, G. Buttazzo, and M. Bertogna, “Bounding the maximum length of non-preemptive regions under fixed-priority scheduling,” in *Proc.* 15th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2009, pp. 351–360.

[14] M. Bertogna, G. Buttazzo, and G. Yao, “Improving feasibility of fixed priority tasks using non-preemptive regions,” in *Proc.* 32nd *IEEE Real-Time Systems Symposium (RTSS)*, December 2011, pp. 251–260.

[15] R. Koymans, “Specifying real-time properties with metric temporal logic,” *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, November 1990.

[16] M. Park, H. Yoo, and J. Chae, “Integration of preemption threshold and quantum-based scheduling for schedulability enhancement of fixed priority tasks,” in *Proc.* 15th *IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2009, pp. 503–510.

[17] G. Yao and G. Buttazzo, “Reducing stack with intra-task threshold priorities in real-time systems,” in *Proc.* 10th *Conference on Embedded Software (EMSOFT)*, October 2010, pp. 109–118.

[18] J. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” in *Proc.* 11th *IEEE Real-Time Systems Symposium (RTSS)*, December 1990, pp. 201–209.

[19] S. Baruah, L. Rosier, and R. Howell, “Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor,” *Real-Time Systems*, vol. 2, no. 4, pp. 301–324, November 1990.

[20] V. Lortz and K. Shin, “Semaphore queue priority assignment for real-time multiprocessor synchronization,” *IEEE Transactions on Software Engineering*, vol. 21, no. 10, pp. 834 – 844, October 1995.

[21] I. Shin, M. Behnam, T. Nolte, and M. Nolin, “Synthesis of optimal interfaces for hierarchical scheduling with resources,” in *Proc.* 29th *IEEE Real-Time Systems Symposium (RTSS)*, December 2008, pp. 209–220.

[22] E. Bini and G. Buttazzo, “Biasing effects in schedulability measures,” in *Proc.* 16th *Euromicro Conf. on Real-Time Systems (ECRTS)*, July 2004, pp. 196–203.

Appendix

In this appendix, Section A provides auxiliary lemmas for the proofs in Section 4. Section B presents preliminaries and auxiliary lemmas for the proof of Theorem 10 in Section 5.

Appendix A
Auxiliary lemmas for limits

The proofs of the following lemmas can be found in [4].

Lemma 11: When $\lim_{x\uparrow X} f(x)$ is defined, and $f(x)$ is strictly increasing in an interval $(X-\gamma, X)$ for a sufficiently small $\gamma \in \mathbb{R}^+$, then the following equation holds.

$$\lim_{x\uparrow X} \lceil f(x) \rceil = \left\lceil \lim_{x\uparrow X} f(x) \right\rceil \tag{49}$$

Lemma 12: When $\lim_{x\uparrow X} f(x)$ is defined, and $f(x)$ is strictly increasing in an interval $(X-\gamma, X)$ for a sufficiently small $\gamma \in \mathbb{R}^+$, then the following equation holds.

$$\lim_{x\uparrow X} \lfloor f(x) \rfloor = \left\lceil \lim_{x\uparrow X} f(x) \right\rceil - 1 \tag{50}$$

Appendix B
Preliminaries and auxiliary lemmas for Theorem 10

Lemma 13 (from [14]): For any time t there exists an arbitrarily small $\varepsilon > 0$ such that $W^*(\pi, t-\varepsilon) = W(\pi, t)$.

Lemma 14 (Lemma 5.2 from [5]): Given a task with a priority π_i, if setting the preemption threshold θ_i equal to the highest priority in the system can not make a task τ_i schedulable, then the task set is unschedulable.

Lemma 15: If a task τ_i is schedulable, then for each of its jobs k there exists a time $t_1 \in \Pi_{i,k}^{(1)}$ such that:

$$B_i^{\mathrm{G}} + \psi_{i,k}^{(1)}(t_1) \leq t_1 \qquad \text{if } B_i^{\mathrm{G}} > 0, \tag{51}$$

and for $B_i^{\mathrm{G}} = 0$ one of the following conditions holds:

1) $\psi_{i,k}^{(1)}\Big|_{(B_i^{\mathrm{G}}>0)}(t_1) < t_1$; or
2) $\psi_{i,k}^{(1)}\Big|_{(B_i^{\mathrm{G}}=0)}(t_1) \leq t_1$.

Proof: First we consider the case $B_i^{\mathrm{G}} > 0$. For a schedulable task τ_i, for each job k there must be a time instant t_1' where the processor becomes available. Moreover, t_1' can be at latest $D_i - C_i$ relative to the release of the task at time $k \times T_i$, otherwise task τ_i misses its deadline. Since the set $\Pi_{i,k}^{(1)}$ exactly contains all time points $t_1 \in (k \times T_i,\ k \times T_i + D_i - C_i]$ where the left-continuous right-hand side of (25) in Lemma 6 is discontinuous, for the smallest $t_1 \geq t'_1$ holds that $\psi^{(1)}_{i,k}(t_1) = \psi^{(1)}_{i,k}(t'_1)$. Hence, $\Pi^{(1)}_{i,k}$ must contain a time instant satisfying (51).

For the case $B^{\mathrm{G}}_i = 0$, we can follow a similar reasoning to find a time $t_1 \in \Pi^{(1)}_{i,k}$ which trivially satisfies the right-continuous function in case 2. We must show that the strict inequality in case 1 is also sufficient to guarantee the schedulability of task τ_i.

This case is almost the same as in [14]. Let $t'_1 \in (k \times T_i;\ k \times T_i + D_i - C_i]$ be a point satisfying case 1. Let t_1 be the smallest point in $\Pi^{(1)}_{i,k}$ such that $t_1 > t'_1$. Using Lemma 13 we know that for an arbitrary small $\varepsilon > 0$, $W^*(\pi_i, t'_1 - \varepsilon) = W^*(\pi_i, t_1) = W(\pi_i, t'_1)$. Hence, $\psi^{(1)}_{i,k}\Big|_{(B^{\mathrm{G}}_i > 0)}(t_1) = \psi^{(1)}_{i,k}\Big|_{(B^{\mathrm{G}}_i = 0)}(t'_1) \leq t'_1 < t_1$. □

Lemma 16: If a task τ_i is schedulable and given the smallest time $t_1 \in \Pi^{(1)}_{i,k}$ satisfying Lemma 15, then for each of its jobs k there exists a time $t_2 \in \Pi^{(2)}_{i,k}(t_1)$ such that:

$$B^{\mathrm{G}}_i + \psi^{(2)}_{i,k}(t_1, t_2) \leq t_2 \qquad \text{if } B^{\mathrm{G}}_i > 0, \tag{52}$$

and for $B^{\mathrm{G}}_i = 0$ one of the following conditions holds:

1) $\psi^{(2)}_{i,k}\Big|_{(B^{\mathrm{G}}_i > 0)}(t_1, t_2) < t_2$; or
2) $\psi^{(2)}_{i,k}\Big|_{(B^{\mathrm{G}}_i = 0)}(t_1, t_2) \leq t_2$.

Proof: Given a schedulable task τ_i: for each job k, which has started executing no later than the earliest discontinuous time point t_1 satisfying Lemma 15, there must be a time instant t'_2 where the processor becomes available to start the final sub-job C_{i,m_i}. Moreover, t'_2 can be at latest $k \times T_i + D_i - C_{i,m_i}$, otherwise task τ_i misses its deadline. Note that t_1 indicates the latest point that job k has been started. Since we only consider discontinuous points of the recursive equation, we may overshoot the actual start time of a job. Hence, t_1 and t'_2 may coincide. Since the set $\Pi^{(2)}_{i,k}(t_1)$ contains all time points $t_2 \in [t_1,\ k \times T_i + D_i - C_{i,m_i}]$ where the left-continuous right-hand side of (27) in Lemma 7 is discontinuous, for the smallest $t_2 \geq t'_2$ holds that $\psi^{(2)}_{i,k}(t_1, t_2) = \psi^{(2)}_{i,k}(t_1, t'_2)$. Hence, $\Pi^{(2)}_{i,k}(t_1)$ must contain a time instant satisfying (52).

For the case $B^{\mathrm{G}}_i = 0$, we follow exactly the same steps as in Lemma 15. □

Lemma 17: If a task τ_i is schedulable and given the smallest $t_1 \in \Pi^{(1)}_{i,k}$ satisfying Lemma 15 and the smallest time $t_2 \in \Pi^{(2)}_{i,k}(t_1)$ satisfying Lemma 16, then for each of its jobs k there exists a time $t_3 \in \Pi^{(3)}_{i,k}(t_2)$ such that:

$$B^{\mathrm{G}}_i + \psi^{(3)}_{i,k}(t_1, t_2, t_3) \leq t_3 \qquad \text{if } B^{\mathrm{G}}_i > 0, \tag{53}$$

and for $B^{\mathrm{G}}_i = 0$ one of the following conditions holds:

1) $\psi^{(3)}_{i,k}\Big|_{(B^{\mathrm{G}}_i > 0)}(t_1, t_2, t_3) < t_3$; or
2) $\psi^{(3)}_{i,k}\Big|_{(B^{\mathrm{G}}_i = 0)}(t_1, t_2, t_3) \leq t_3$.

Proof: Given a schedulable task τ_i: for each job k, which has started executing its final sub-job m_i no later than the earliest time t_2 satisfying Lemma 16, there must be a time instant t'_3 no later than $k \times T_i + D_i$. Note that t_2 indicates the latest time at which sub-job m_i has been started. Hence, t_2 and t'_3 may coincide. Since the set $\Pi^{(3)}_{i,k}$ contains all time points $t_3 \in [t_2,\ k \times T_i + D_i]$ where the left-continuous right-hand side of (29) in Lemma 8 is discontinuous, for the smallest $t_3 \geq t'_3$ holds that $\psi^{(3)}_{i,k}(t_1, t_2, t_3) = \psi^{(3)}_{i,k}(t_1, t_2, t'_3)$. Hence, $\Pi^{(3)}_{i,k}(t_2)$ must contain a time instant satisfying (53).

For the case where $B^{\mathrm{G}}_i = 0$, we have to consider the mixture of left-continuous and right-continuous operands in $\psi^{(3)}_{i,k}\Big|_{(B^{\mathrm{G}}_i > 0)}(t_1, t_2, t_3)$. By observing that terms that depend on t_1 and t_2 are constants (given by Lemma 15 and Lemma 16), we are again left with a right-continuous function of t_3. We can therefore again apply the same steps as in Lemma 15. $\square$

PAPER B:

VIRTUAL SCHEDULING FOR COMPOSITIONAL REAL-TIME GUARANTEES

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien

In proceedings of the 8th IEEE International Symposium
on Industrial Embedded Systems (SIES),
(to appear), June 2013

ABSTRACT

In this paper, we extend the compositional scheduling framework to enable the integration of an existing (legacy) application as a component on a faster processor which needs to be shared with other components. After admission of this application into the framework, the integrated component still has to satisfy its tasks' deadline constraints and it must execute jobs in the same order as it did previously on the dedicated processor, regardless of the actual supply of processor resources. We propose a method – called *virtual scheduling* – which establishes this by reconstructing an appropriate order of delivering events. Virtual scheduling is therefore independent of the component-level scheduling policy and it is transparent to a component, so that it is even applicable without making modifications to the code or specification of the application.

1 Introduction

Many industrial systems have various hardware and software applications for controlling the physical world through electronics and mechanics. These applications can be complex, distributed and interdependent. Examples can be found in a wide range of application domains, including manufacturing machines, highly dependable medical devices, intelligent transportation and in-vehicle control. There is a trend towards optimizing the overall system performance and creating a synergy between the different applications in a system. This makes it increasingly complex to take care of the real-time requirements and constraints of individual applications separately. Those application dependencies and requirements, e.g., end-to-end latencies and precedence constraints, are typically captured in application models and are translated to task attributes, e.g., deadlines and release offsets, by using off-line analysis (see [1] and [2]). After resolving those complex dependencies and requirements, the resulting components, each comprising a set of tasks with implicit constraints and a (local) task scheduler, are expected to execute (part of) the application's functionality properly on an off-the-shelf real-time platform. Such a platform consists of uni-processor hardware and a real-time operating system (RTOS). One component is then analyzed for only this dedicated processor that supplies resources continuously, i.e., without any disruptions due to execution of other components.

The traditional approach in the design and development of applications makes use of an architecture that simply composes self-contained hardware. In these so-called *federated architectures*, platforms are connected by several network buses and each set of inter-connected platforms typically hosts a single, independently developed application. The last few years have shown, however, that this federated architectural approach can no longer stand [3]. Since the number of electronic applications in embedded systems is ever increasing, it becomes too expensive to make dedicated hardware units for each application. Fortunately, modern hardware is able to run several components, derived from arbitrary applications, thereby taking advantage of the opportunities offered by today's better programmable and increasingly powerful platforms. However, the integration of components on the same platform inherently requires resource sharing between components, which may impact the input and output (I/O) delays experienced by the applications' tasks.

The real-time systems community has therefore proposed *compositional scheduling frameworks* (CSFs) to abstract from resource sharing between components and the corresponding scheduling delays associated with it. The main goal of a CSF [4] is to establish system-level timing properties by composing independently specified and analyzed component-level timing properties, thus, improving the re-usability of components. Current CSFs, e.g., [4]-[8], propose different models to combine and abstract timing constraints of tasks within a component as a single real-time constraint, called *a real-time interface*. A real-time interface constrains the variability and the discontinuities in the resource supply to a component, so that component-level timing

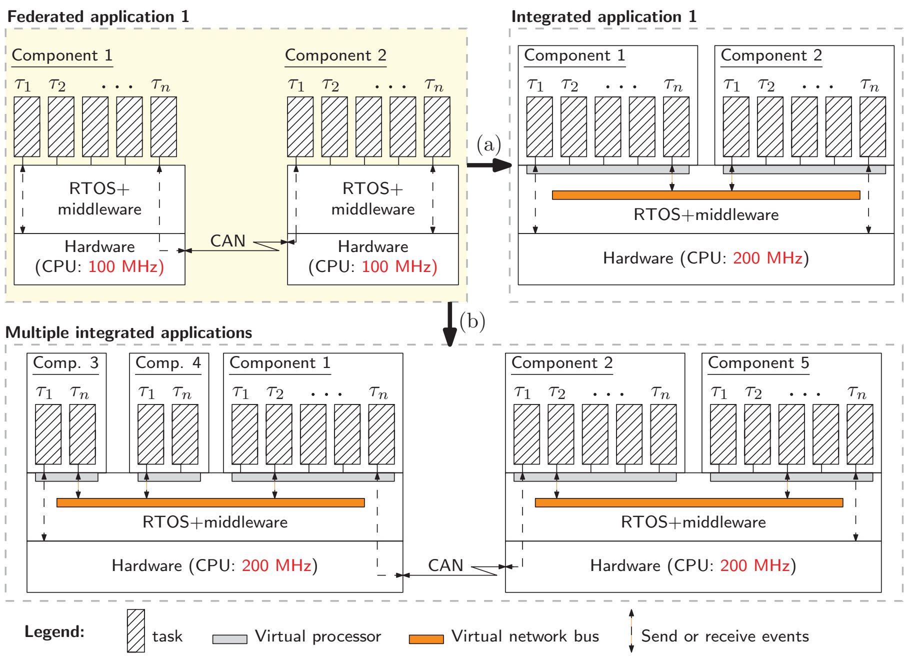

Figure 1. Whereas an application may be developed in isolation for a federated architecture, it may need to run together with other applications on integrated architectures in order to meet the demands of further limiting the growing number of increasingly powerful computing platforms embedded in today's real-time systems. For example, the components of one application can be integrated on a faster, shared platform, see (a), or the components of several applications can be integrated on a faster, shared platform, see (b). As a consequence of resource sharing between components in such integrated architectures, the events – either sent over a virtual network bus or sent by real hardware devices such as timers and the controller area network (CAN) – may arrive when the destined component is suspended and they may be handled by tasks in a different order in an integrated architecture than they were previously handled in a federated architecture. Moving from federated to integrated architectures, e.g., as described in [3], therefore requires additional validation of timing constraints at present.

properties can be established by the local scheduler. Each individual component may define its own scheduling policy to distribute its allocated resources to its tasks. A set of components can be composed by satisfying each of the individual resource supplies specified by the real-time interfaces.

1.1 Problem description and solution outline

A component in a CSF can be regarded as running on a dedicated processor at a speed proportional to its resource supply, i.e., the component is said to execute on a *virtual processor*. Two effects disturb this point of view. Firstly, resources are supplied discontinuously. Secondly, the actual processor is faster. If the component was validated under a continuous supply, as in legacy and development situations, the local deadlines of tasks and the precedence constraints of tasks might be violated. Even with an optimal local scheduling policy (like EDF) that admits delaying execution, the execution order of tasks may change by shifting the resource supply in time. An additional validation of local constraints is therefore required to prevent these artefacts. Since these local constraints might be implicit after the application's synthesis, re-validation of local constraints might be impossible.

In this paper, we look at the events associated to the real-time tasks of a component that was once developed and analyzed upon a federated architecture and that needs to be integrated upon a faster, shared platform. Figure 1 illustrates two scenarios. On a virtual processor, hardware resources are temporarily made unavailable to a component. Once the resources become available, multiple events might be pending and the resulting precedence order of jobs handling those events can be affected compared to a dedicated platform. We focus on synchronization between tasks located in arbitrary components by means of event-driven resource sharing and communication.

We extend CSFs with a mechanism to integrate components onto a shared processor based on their analysis on a dedicated processor, while replicating the same delivery order of events. By managing events appropriately, our only dependence on the resource-supply model then is whether or not the supply is timely enough for meeting deadlines of the tasks handling those events. This dependence is abstracted in a real-time interface and does not require an additional validation of the local deadlines and precedence constraints on the virtual processor.

We can now truly analyze a component independently, because a component can be analyzed as if it has the entire processor at its disposal. Hence, our goal is to implement a virtual processor which virtually supplies resources continuously. It must be (*i*) efficient to implement, (*ii*) transparent to a component and (*iii*) independent of the local scheduling policy.

1.2 Contribution

The contributions of this paper are as follows:

- We propose a *virtual-scheduling* policy to support synchronization on external events by tasks of one or more components that share the same processor. We therefore solve the problem of replicating the original delivery order of events on a virtual processor by means of a virtual-time release guard, thereby automatically satisfying the precedence constraints of tasks implicitly.

- We investigate the impact of virtual scheduling on the required budget of a component. We compare virtual scheduling to the periodic and bounded-delay resource-supply models in [6] and [4]. Virtual scheduling decouples the component's local analysis and its underlying resource-supply model. Despite our obtained abstraction, mapping virtually scheduled components onto the bounded-delay resource-supply model [6] is efficient and tight.
- We make task synchronization on external events transparent during the timing analysis of a component. We therefore give recommendations for managing events in a predictable manner using virtual timers and using a clear separation between internal and external events.

2 Related work

The increasing complexity of real-time systems led to a growing attention for compositional analysis [4]-[11]. One of the challenges of real-time composition is to derive a *real-time interface* for a component. The framework parameters that define a worst-case resource supply to a component, contemplated that timing constraints within an individual component are satisfied, together form a real-time interface. This real-time interface - sometimes also referred to as a *supply contract* [8] - abstracts the timing constraints of tasks into a single real-time constraint.

Wandeler and Thiele [7] calculate demand and service curves for components using real-time calculus. Shin and Lee [4] proposed the periodic resource model to guarantee periodic processor supplies to components. The explicit-deadline periodic (EDP) resource model [5] extends the periodic resource model in [4] by explicitly distinguishing a relative deadline for the allocation time of periodic guarantees. The bounded-delay model [6] describes linear service curves with a bounded initial service delay. Both Lipari and Bini [10] and Shin and Lee [11] have presented methods to convert a bounded-delay supply into a periodic resource supply.

Matic and Henzinger [12] showed how to abstract from data dependencies between tasks in dataflow graphs in two different formalisms (i.e., Real-Time Workshop (RTW) semantics of Mathworks and Logical Execution Time (LET) semantics) within the context of the periodic resource model [4]. In their model, tasks may have inter-dependencies with other tasks residing within both the same component or another component. Anand [13] has extended the recurring branching task model [14] with control variables at decision points within the context of the EDP resource model [5].

Both approaches in [12] and [13] abstract from task inter-dependencies at the level of composition. A drawback of their approaches is that they implement synchronization logic at each appointed synchronization point between tasks. Contrary, Wang et al. [8] reflect all task inter-dependencies within a component onto the resource supply. Although no further implementation logic is needed for task synchronization, task inter-dependencies are indirectly captured in the component's interface. This inherently comes at the cost of schedulability penalties at the level of composition.

Mutually exclusive execution of tasks within a component can be supported with standard semaphore-based protocols [9] such as the stack resource policy (SRP) [15]. The SRP can also implement precedence constraints between tasks [16]. A disadvantage of semaphores is that the correspondingly varying execution priorities of tasks, are hard to analyse [17]. This concern is especially important for highly critical systems which must go through a certification process.

2.1 This work and its significance

In this work we develop compositional techniques for predictable task scheduling upon virtual processors. At present, system designers manually schedule partitions based on information from the partition vendors [18]. Although Easwaran et al. [18] have improved the component-level timing analysis of the EDP resource model [5] to abstract from the scheduling offsets of tasks, their approach lacks mechanisms to enforce a predictable order of job execution upon a partition at run time. Also in this paper we assume that precedence constraints between the tasks of a single component are offline converted into release offsets (phasing). Contrary to [18], however, we assume that the component has the entire processor at its disposal when its schedule is constructed. Dobrin et al. [1] and Forget et al. [17] have developed such methods for fixed-priority scheduled tasks; Isovic and Fohler [2] developed similar methods for earliest-deadline-first (EDF) scheduled task sets. We propose a method to preserve those programmed task inter-dependencies in the context of CSFs, without the need to implement logic separately at each individual synchronization point (as [12], [13] and [16] do).

For this purpose we use a virtual-time release guard. Our way of separating job releases (events) looks similar to the release-guard protocol [19] or period-enforcement [20] of sporadic tasks. However, the protocols in [19] and [20] guarantee a minimal separation between subsequent jobs of the same task in real time while our approach guarantees a minimal separation in virtual time, i.e., relative to the consumed processor time by a component, between job releases of different tasks.

A key for predictable composition of components is managing the events associated to tasks, such as timers and other types of signals. When the processor is unavailable, multiple events may arrive to a component. It is the responsibility of a CSF to deliver these events to the destined tasks without hampering the timing of other components' tasks. As an example, Parmer et al. [21] have proposed *brands and up-calls* in an RTOS, called CompositeOS, that supports hierarchical composition of tasks and components. Their mechanisms – the brands and up-calls – have been invented to defer event handling until the receiving task of an event has the highest priority in the system. In this way, the overheads for handling events are charged to the associated task. Also hypervisors, like the SPIRIT μkernel [22], XtratuM [23], L4 [24] and VMware [25] by default follow a similar approach for buffering and deferring the propagation of events to partitions (components). However, none of these existing CSFs guarantees that the order of delivering events to a component is preserved compared to the order of delivery on a dedicated processor. We propose virtual scheduling for this, which has as an additional advantage that events are delivered during the execution of the destined component.

3 MODELS AND NOTATION

We consider a two-level scheduling framework, where a system comprises a set of N independent components that need to share a single processor of one-unit speed. A component C contains a set $\mathcal{T}$ of n recurring tasks τ_1, …, τ_n. A task τ_i generates an infinite sequence of jobs, $J_{i,1}, \ldots J_{i,k}$. Upon the *arrival* of a job $J_{i,k}$ at time $a_{i,k}$, this job $J_{i,k}$ has to complete its work no later than its absolute deadline $d_{i,k}$.

In fact, we allow any recurring task model which satisfies the following two properties: (i) in any time interval of arbitrary length t, each task must have an upper bound on the workload generated by its jobs; and (ii) the job arrivals of the tasks are triggered by an external event. Tasks may have internal deadlines [26], so that a part of its job's workload has to be completed prior to a relative deadline from its arrival[1].

Without further loss of generality and for ease of presentation, we focus this paper on periodic tasks that generate jobs triggered by timed events and a single deadline associated with each of them. The timing of a periodic task $\tau_i \in \mathcal{T}$ is specified by a triple $(T_i,\ E_i,\ D_i)$, where $T_i \in \mathbb{R}^+$ denotes its period, $E_i \in \mathbb{R}^+$ denotes its worst-case execution time and $D_i \in \mathbb{R}^+$ denotes its deadline relative to its job arrivals. We restrict to deadline-constrained tasks, i.e., $D_i \leq T_i$. The first job $J_{i,1}$ of a periodic task arrives with an arbitrary phasing ϕ_i and each job $J_{i,k}$, where $k \geq 1$, arrives at time $a_{i,k} = \phi_i + (k-1)T_i$ with a corresponding absolute deadline $d_{i,k} = a_{i,k} + D_i$.

A unique system-level (global) scheduler selects which component is executed on the shared processor. The component-level (local) scheduler decides which of the released jobs of the executing component is allocated the processor. The global scheduler and each of the local schedulers of individual components may apply different scheduling policies. The results contained in this paper apply to any task sets scheduled by a *work-conserving* scheduling policy, defined as follows.

Definition 1: A scheduling policy is work conserving if and only if it never idles the processor when there exists a job with pending execution requests.

We adapt our results for two example policies: earliest-deadline-first (EDF) scheduling of jobs – an optimal dynamic-priority uni-processor scheduling algorithm – and fixed-priority scheduling (FPS) of jobs – the de-facto standard in industrial systems for the scheduling of real-time tasks.

1. A deadline constrains the progress of a job relative to its arrival. Different ways of constraining the progress of job executions with respect to real time are disallowed, because those may disallow porting a component to a virtual processor with discontinuities in its supply at arbitrary moments in time.

3.1 Designing a bounded-delay partition recapitulated

Mok et al. [27] described how to modify the analysis of a task set that has been validated on a dedicated processor and needs to be scheduled on a shared processor with other components. They assume that a task set meets its deadline constraints on a slower dedicated α-speed processor, i.e., $\alpha \in (0,1]$. A task τ_i, with a WCET of E_i on a unit-speed processor, is assumed to have a WCET of $\frac{1}{\alpha}E_i$ on a dedicated α-speed processor.

Feng and Mok [6] extended their bounded-delay model with a layer of abstraction, so that if a task set is deemed schedulable on a slower processor of α speed and each task allows a maximum release delay of Δ time units on a dedicated α-speed processor without missing a deadline, then this task set is also schedulable on a virtual processor which gives a continuous α-fraction of a unit-speed processor after an initial delay of at most Δ time units. The parameter Δ is derived from the so-called *blocking tolerance* [28], β, of the task set $\mathcal{T}$.

Definition 2: Assume a task set $\mathcal{T} = \{\tau_i \mid 1 \le i \le n\}$ has an entire α-speed processor at its disposal. The blocking tolerance β_i of a task τ_i is the longest scheduling delay after the arrival of a job $J_{i,k}$ of task τ_i without missing its deadline $d_{i,k}$.
The blocking tolerance β_i of a task τ_i should be interpreted as follows. If a job $J_{i,k}$ of task τ_i is delayed for a duration of at most β_i time units, then the processing time of job $J_{i,k}$ may increase due to (*i*) this scheduling delay itself of at most β_i time units and (*ii*) any potential extra interference of higher-priority jobs that are released during this scheduling delay.

Definition 3: Assume a task set $\mathcal{T}$ has an entire α-speed processor at its disposal. The *blocking tolerance* β of task set $\mathcal{T}$ is the largest blocking that any task in $\mathcal{T}$ can tolerate without missing a deadline. Thus,

$$\beta \overset{\text{def}}{=} \min\{\beta_i \mid 1 \le i \le n\}. \qquad (1)$$

In the bounded-delay model of Feng and Mok [6], the value of β is computed upon a continuous α-speed processor and this value is then termed Δ. This principle allows α and Δ to be used as interface parameters of a component.

Since each component specifies an interface to abstract the resource requirements of its task set, this interface can be used to determine whether or not a component can be admitted together with other components on the same shared processor. If a component is admitted, a virtual processor - also referred to as a $(\alpha,\ \Delta)$-partition - guarantees and enforces the specified resources in the interface, so that the component keeps meeting its timing constraints on the shared processor as long as it behaves as specified. If it violates its interface specification, it may be penalized, but other components are temporally isolated from the malicious effects. In line with the bounded-delay model [6], we specify the interface of each component as a tuple (α, Δ). Parameter α represents the virtual processor speed and Δ is the maximum service delay.

3.2 Motivating example

Consider an example component comprising two EDF-scheduled periodic tasks: $\tau_1 = (8,1,4)$ and $\tau_2 = (8,1,8)$. By choosing a phasing $\phi_1 = 2$ and $\phi_2 = 0$, we enforce that job k of task τ_2 completes its execution prior to job k of task τ_1. This execution order of jobs is the reverse of their deadline ordering. Each job completes at least $\Delta = 2$ time units prior to its deadline on a $(\alpha = \frac{1}{2})$-speed processor. Although this task set satisfies all deadline constraints on a bounded-delay partition with parameters $(\frac{1}{2}, 2)$ of a unit-speed processor, the execution order of the jobs on this virtual processor may change. In the worst-case scenario, the first time unit of processor supply is provided at time 2. At time 2, τ_1 releases a job with a shorter absolute deadline of $d_{1,1} = 6$ than the deadline $d_{2,1} = 8$ of the pending job of τ_2, so that τ_1 immediately starts executing. This violates the precedence order $\forall k \ : \ k \geq 1 \ : \ J_{2,k}$ completes prior to $J_{1,k}$.

EDF guarantees that all tasks make their deadline on a virtual processor of the same reference speed, α, as the dedicated processor as long as the service delay Δ respects the maximum allowable blocking of tasks [6]. However, the execution order of jobs on a virtual processor may change compared to the dedicated reference processor.

3.3 Detailed problem description

Consider a component C, composed of a set of tasks $\mathcal{T}$ with fixed arrival times and absolute deadlines. The objective of virtual scheduling is as follows. If all tasks $\tau_i \in \mathcal{T}$ always complete their execution at least Δ time units prior to their deadlines when scheduled by a policy SP upon a dedicated α-speed processor, then

- all tasks make the same absolute deadlines with the same component-level scheduling policy SP on an (α, Δ)-partition of a unit-speed processor; and
- all the jobs execute in the same order on an (α, Δ)-partition of a unit-speed processor.

A virtual processor can deliver processor resources too early or too late (by a degree of Δ) compared to a dedicated processor with a continuous supply. First we show that the exact schedule can be replicated by constraining the virtual processor such that it supplies resources never too early (Section 4.1). Secondly, we relax this constraint on the resource supply (Section 4.3).

4 Virtual scheduling

In this section, we present the rules used by our compositional scheduling framework to make scheduling decisions. We first describe the global scheduling algorithm. Next, Section 4.2 presents a proof of correctness of our scheduling algorithm. Finally, Section 4.3 relaxes the assumptions in our system model on the resource supply.

4.1 System-level scheduling

At each instant, the system-level scheduler selects some partition for execution. For each partition, we define the least amount and the actual amount of processor resources that the hosted component C receives from the global scheduler.

Definition 4: The supply bound function, $\texttt{sbf}(t)$, returns the minimum amount of processor resources that a component receives in any arbitrary (sliding) time interval of length t. After a delay of Δ time units, the $\texttt{sbf}(t)$ supplies at least a continuous fraction, α, of the processor, i.e.,

$$\forall t: \quad t \geq 0: \quad \texttt{sbf}(t) = \max(0,\ \alpha(t - \Delta)). \tag{2}$$

From (2), we observe that the longest interval in which a component may receive no processor supply is Δ, i.e.,

$$\Delta \quad = \quad \max\{t \mid t \geq 0 \wedge \texttt{sbf}(t) = 0\}. \tag{3}$$

Definition 5: The actual supply function, $\texttt{asf}(t)$, returns the actual processor resources that a component has received in a designated time interval $[0, t)$; it is a non-decreasing function of time which must satisfy the following property:

$$\forall t: \quad t \geq 0: \quad \texttt{sbf}(t) \leq \texttt{asf}(t) \leq \texttt{sbf}(t + \Delta). \tag{4}$$

The restriction in (4) that $\texttt{asf}(t) \leq \texttt{sbf}(t + \Delta)$ guarantees that a virtual processor will never supply more processor time than a dedicated α-speed processor does. We recall that Section 4.3 removes this restriction on the $\texttt{asf}(t)$.

From (2) and Definition 5 the following corollary follows.

Corollary 1: After a longest interval without processor supply, $[t_s - \Delta,\ t_s)$, has ended at a time t_s, the $\texttt{asf}(t)$ continuously gives at least an α-share of the processor, i.e.,

$$\forall t \ : \ t \geq t_s \ : \ \texttt{asf}(t) - \texttt{asf}(t_s) \geq \alpha(t - t_s). \tag{5}$$

We introduce additional state associated with a partition. Each partition, hosting a component C, maintains

- a variable to keep track of virtual time, V;
- a variable R to track the absolute time of the most recent job release (i.e., an internal event);
- a variable A to track the absolute time of the most recent job arrival (i.e, an external event); and
- a buffer B to plan future releases of the stored job arrivals.

The key idea behind a virtual-scheduled partition is to distinguish the external event, i.e., the arrival of a job to a partition, (captured by variable A) and its corresponding internal event, i.e., the release of a job, (captured by variable R), whereas by default the arrival and the release are coinciding ($A = R$). Only after a job has been released, it can contend with earlier released jobs for the processor time supplied by its partition. In order to replicate the schedule of the jobs on a continuous processor, a (discontinuous) virtual processor must deliver the same amount of processor time in between the *releases* of two arbitrary jobs as a continuous processor does.

Whenever a component receives processor time, its virtual time, V, progresses with a rate proportional to its allocated processor share. We use $V(t)$ to retrieve the value of the virtual time, V, of component C at real time t. Buffer $B[0...x](t) = \{(J_{i,k},\ V_{i,k}) \mid a_{i,k} \leq t \wedge V_{i,k} > V(t)\}$ contains *a chronologically sorted set* of tuples denoting those arrived jobs that have a computed release at virtual time $V_{i,k}$ somewhere in the future.

To establish the desired separation between job releases on a partition, the variables V, R and A and buffer B are updated by virtual scheduling according to the following rules (where $t_{\texttt{cur}} \geq 0$ denotes the current real time):

1) Initially, at time instant $t_{\texttt{cur}} = 0$, we set:

$$\begin{array}{ll} V,\ R,\ A & \leftarrow 0; \\ B & \leftarrow \emptyset; \end{array} \tag{6}$$

2) Whenever a component C receives one unit of processor time, the virtual time progresses with $\frac{1}{\alpha}$ units, i.e.,

$$\frac{\mathrm{d}}{\mathrm{d}t}V = \begin{cases} \frac{1}{\alpha} & \text{if } C \text{ is executing;} \\ 0 & \text{else.} \end{cases} \tag{7}$$

3) Given a job $J_{i,k}$ arriving at time instant $t_{\texttt{cur}}$. If $B = \emptyset$ and $V(t_{\texttt{cur}}) - V(R) \geq t_{\texttt{cur}} - A$, then job $J_{i,k}$ is released immediately, accompanied by the actions:

$$R,\ A \quad \leftarrow t_{\texttt{cur}}; \tag{8}$$

4) Given a job $J_{i,k}$ arriving at time instant $t_{\texttt{cur}}$. If $B = \emptyset$ and $V(t_{\texttt{cur}}) - V(R) < t_{\texttt{cur}} - A$, then the release of job $J_{i,k}$ is buffered until time instant t' at which the virtual time reaches $V(t') = V_{i,k}$, where

$$\begin{array}{ll} V_{i,k} & \leftarrow t_{\texttt{cur}} - A + V(R); \\ A & \leftarrow t_{\texttt{cur}}; \\ B & \leftarrow B \cup \{(J_{i,k},\ V_{i,k})\}; \end{array} \tag{9}$$

5) Given a job $J_{i,k}$ arriving at time instant $t_{\texttt{cur}}$. Let $V^{\max} = \max\{V_{j,\ell} \mid (J_{j,\ell},\ V_{j,\ell}) \in B\}$ be the virtual time corresponding to the last job stored in B. If $B \neq \emptyset$, then the release of job $J_{i,k}$ is buffered until time instant t' at which the virtual time reaches $V(t') = V_{i,k}$, where

$$\begin{array}{ll} V_{i,k} & \leftarrow t_{\texttt{cur}} - A + V^{\max}; \\ A & \leftarrow t_{\texttt{cur}}; \\ B & \leftarrow B \cup \{(J_{i,k},\ V_{i,k})\}; \end{array} \tag{10}$$

6) If $B \neq \emptyset$ and $B[0] = \big(J_{i,k},\ V(t_{\mathtt{cur}})\big)$ hold at time $t_{\mathtt{cur}}$, then job $J_{i,k}$ is released and removed from the buffer, described by the following administrative actions:

$$\begin{aligned} B &\leftarrow B \setminus \{B[0]\}; \\ R &\leftarrow t_{\mathtt{cur}}; \end{aligned} \tag{11}$$

The above definitions and rules basically describe a bounded-delay partition [6]. One difference with the original bounded-delay partition is the restriction in (4) on the upper bound of $\mathtt{asf}(t)$, which has been introduced in order to guarantee that the virtual time cannot run ahead. The virtual time is directly related to, and serves as a short-hand notation for, the progress of the actual supply. Formally written, $\forall t_1, t_2 \ : \ 0 \leq t_1 \leq t_2 \ :$

$$V(t_2) - V(t_1) \quad = \quad \frac{1}{\alpha}\left(\mathtt{asf}(t_2) - \mathtt{asf}(t_1)\right). \tag{12}$$

Using our restriction in (4) on the $\mathtt{asf}(t)$, in the best case a virtual processor exactly gives an α share of the unit-speed processor without a service delay, so that the virtual processor exactly behaves as a dedicated processor of α speed. Since this constraint on the $\mathtt{asf}(t)$ might be hard to implement, in the next section we shall show how to remove this constraint.

The other difference with the original bounded-delay partition is the rules for arbitrating job releases. Rules 1, 3-6 guarantee that between any two job releases, with corresponding arrival times $a_{j,\ell} \geq a_{i,k}$, the virtual processor provisions at least $\alpha(a_{j,\ell} - a_{i,k})$ time units of processor supply to the component.

Example

Consider a task set comprising three tasks: $\tau_1 = (6,1,6)$, $\tau_2 = (8,1,8)$ and $\tau_3 = (8,1,8)$. Task τ_1 is assigned the highest priority and task τ_3 the lowest priority. The release offsets of these tasks are fixed by $\phi_1 = \phi_3 = 0$ and $\phi_2 = 2$. When scheduling this task set with the given fixed-priority assignment on a dedicated processor of $\alpha = \frac{1}{2}$ speed, each of these tasks complete their execution at least $\Delta = 2$ time units prior to their deadline, see Figure 2(a). After admission of this task set on a bounded-delay partition with parameters $\alpha = \frac{1}{2}$ and $\Delta = 2$ of a unit-speed processor, virtual scheduling makes sure that the schedule of jobs is replicated and at the same time all deadlines are met, see Figure 2(b).

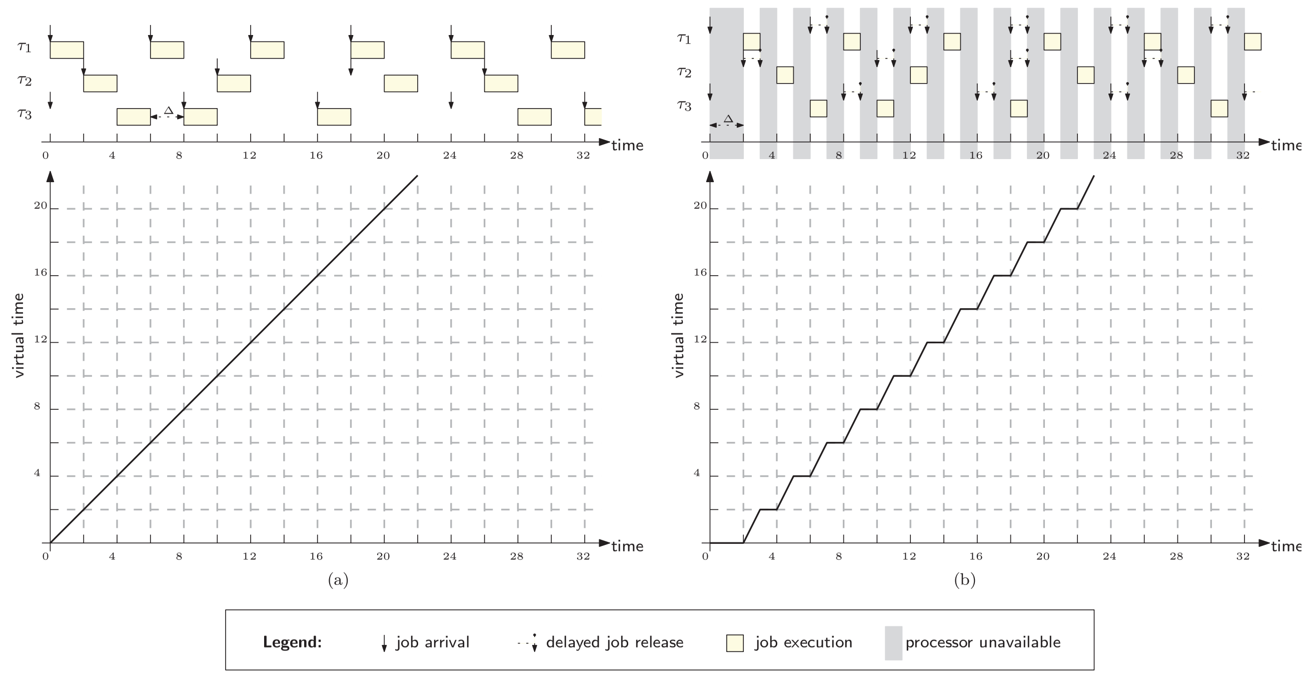

Figure 2. Example of a fixed-priority scheduled task set executing (a) on a dedicated $(\alpha = \frac{1}{2})$-speed processor and (b) on a $(\frac{1}{2},\, 2)$-bounded-delay partition of a unit-speed processor while virtually scheduled under a worst-case supply.

Under a best-case processor supply jobs are released at their arrival, because the virtual processor exactly behaves as a dedicated α-speed processor. According to the specified rules of virtual scheduling, the following scheduling decisions are made under a worst-case processor supply (see Figure 2(b)):

$t = 0$: $V,\ R,\ A \leftarrow 0$, $B \leftarrow \emptyset$ and both job $J_{1,1}$ and job $J_{3,1}$ are released immediately. (*Rule 1 and 3*)

$t = 2$: Job $J_{2,1}$ is buffered until virtual time 2, i.e., $V_{i,k} \leftarrow 2$, $A \leftarrow 2$ and $B \leftarrow \{(J_{2,1},\ 2)\}$. (*Rule 4*)

$t = 3$: Since $V(3) = 2$, job $J_{2,1}$ is released and removed from the buffer. We set $R \leftarrow 3$ and $B \leftarrow \emptyset$. (*Rule 6*)

...

$t = 18$: Job $J_{1,4}$ is buffered until virtual time 18, i.e., $V_{i,k} \leftarrow 18$, $A \leftarrow 18$ and $B \leftarrow \{(J_{1,4},\ 18)\}$; Next, job $J_{2,3}$ is buffered until virtual time 18, i.e., $V_{i,k} \leftarrow 18$, $A \leftarrow 18$ and $B \leftarrow \{(J_{1,4},\ 18), (J_{2,3},\ 18)\}$; (*Rule 4 and 5*)

$t = 19$: Since $V(19) = 18$, both job $J_{1,4}$ and job $J_{2,3}$ are released and removed from the buffer. We set $R \leftarrow 19$ and $B \leftarrow \emptyset$. (*Rule 6*)

$t = 24$: The hyper-period repeats itself.

4.2 Bounding the release delay of job arrivals

The scheduling rules described in the previous section may cause a delay in the release of job arrivals compared to a regular bounded-delay partition. This section analyses the durations of those release delays, which are directly related to the maximum allowable release delay Δ on a dedicated α-speed processor. We conclude that an $(\alpha,\ \Delta)$-partition exactly captures the required processor share of a virtually scheduled component.

Lemma 1: The absolute difference between the real time and virtual time is never larger than Δ, i.e.,

$$\forall t\ :\ t \geq 0\ :\ |V(t) - t| \leq \Delta. \tag{13}$$

Proof: We rewrite (13) by applying (12) as

$$\forall t\ :\ t \geq 0\ :\ \left|\frac{1}{\alpha}\texttt{asf}(t) - t\right| \leq \Delta. \tag{14}$$

The difference between $\texttt{asf}(t)$ and αt is at its maximum when $\texttt{asf}(t)$ is as small as possible, i.e., $\texttt{asf}(t) = \texttt{sbf}(t)$. Moreover, from the definition of $\texttt{sbf}(t)$ in (2), we know that $\texttt{sbf}(t) \geq \alpha(t - \Delta)$. Substituting $\texttt{asf}(t)$ in (14) for $\texttt{sbf}(t)$ gives

$$\forall t\ :\ t \geq 0\ :\ \left|\frac{1}{\alpha}\left(\alpha\left(t - \Delta\right)\right) - t\right| \leq \Delta. \tag{15}$$

From (15) we derive $\Delta \leq \Delta$, which proves this lemma. $\square$

Lemma 2: Given n deadline-constrained periodic tasks. If all n tasks meet their deadline on a virtually scheduled bounded-delay partition, then the length of buffer B is at most n.

Proof: Assume all n tasks are stored in the buffer. Prior to any subsequent job arrival $J_{i,k+1}$ of task τ_i, its previous job $J_{i,k}$ must have left the buffer. Otherwise job $J_{i,k}$ misses its deadline, thereby violating the premise. □

From Lemma 2 we conclude that the memory complexity of the scheduling queues is the same with and without virtual scheduling, because an arrived job is either represented in buffer B or it is is represented in the ready queue. We now bound the sojourn time of a job in buffer B prior to its transfer to the ready queue at its release.

Lemma 3: Let $b_{i,k}$ be the sojourn time of job $J_{i,k}$ in buffer B and let $s_{i,k}$ be the duration that the processor resources are unavailable to the component after the release of job $J_{i,k}$. For all jobs arbitrated by virtual scheduling, it holds $s_{i,k} + b_{i,k} \leq \Delta$.

Proof: From Definition 4 follows $s_{i,k} \leq \Delta$, so that a job not buffered prior to its release (Rule 3) fulfills this lemma.

Consider those jobs that are buffered in B according to Rule 4 ($B = \emptyset$) and Rule 5 ($B \neq \emptyset$) of virtual scheduling. Under the worst-case progress of virtual time, a service delay of Δ units happened prior to the release of $J_{i,k}$. By virtue of Corollary 1, it then holds that $s_{i,k} = 0$ and we must now bound $b_{i,k}$ to Δ.

Rule 4: Assume $B = \emptyset$, so that $R \geq A \geq 0$. Consider a job $J_{i,k}$ arrives at time $a_{i,k}$. Hence, job $J_{i,k}$ is released when the virtual time violates the premise of Rule 4, i.e., at a time t' s.t.

$$\begin{aligned} & V(t') - V(R) && = a_{i,k} - A \\ \equiv\ & V(t') - a_{i,k} && = V(R) - A. \end{aligned} \tag{16}$$

If the buffer B is currently empty, then the previous job release at time R has occurred earlier at a time where $A = V(R)$.
Hence,

$$V(t') \quad = \quad a_{i,k} \tag{17}$$

Using (17) we obtain $b_{i,k} = t' - a_{i,k} = t' - V(t')$. Thus, according to Lemma 1 we conclude that $b_{i,k} \leq \Delta$.

Rule 5: Assume $B \neq \emptyset$ and consider job $J_{i,k}$ arrives at time $a_{i,k}$. We prove this rule by induction in the length of buffer B (bound by Lemma 2), where Rule 4 covers the base step.

Consider a buffer of length ℓ where the most recently inserted job has a a virtual timer expiring at time $V^{\texttt{max}} = A$. Job $J_{i,k}$ is released when (at a time $t' \geq a_{i,k}$) the virtual time reaches

$$\begin{aligned} & V(t') && = a_{i,k} + V^{\texttt{max}} - A \\ \equiv\ & V(t') && = a_{i,k}. \end{aligned} \tag{18}$$

Hence, the $(\ell+1)$-th inserted value inside buffer B is (at most) $V^{\texttt{max}} \leftarrow a_{i,k}$. Since $V^{\texttt{max}} = a_{i,k}$ happens no later than Δ time units after $a_{i,k}$ (see Lemma 1), the proof for Rule 5 follows by induction. Combining Rule 3, Rule 4 and Rule 5 proves this lemma. □

Given Lemma 3 and the rules of virtual scheduling which separate any two job releases by at least an α share of the time between their absolute arrival distance, we can now directly apply the known result from Mok et al. [27].

Theorem 1 (Theorem 6 in [27]): Given a component C and a bounded-delay partition $(\alpha,\ \Delta)$, let S_n denote a valid schedule on a dedicated processor with speed α, and S_p the schedule of the component on partition $(\alpha,\ \Delta)$ according to the same execution order and amount as S_n. Also let $\overline{\Delta}$ denote the largest amount of time such that any job of component C is completed at least $\overline{\Delta}$ time units before its deadline. S_p is a valid schedule if and only if $\overline{\Delta} \geq \Delta$.

Theorem 1 tells that if all tasks finish their execution at least Δ time units prior to their deadline on a dedicated processor of α speed, then the task set is schedulable on a $(\alpha,\ \Delta)$ bounded-delay partition provided that the same execution order and amount is replicated. Literature did not describe how to realize a replicated schedule, which we have presented in this section.

4.3 Sustainable relaxations

The scheduling of hard real-time tasks must be predictable, so that all critical timing constraints of the tasks are satisfied in advance. However, in many cases it is irrelevant for the predictability of the system to know exactly who is executing. Especially in the composition of off-line scheduled task sets, it is hard to establish the exact distance between the execution of different jobs [8]. Fortunately, replicating the precedence constraints of jobs is easier than replicating the entire schedule of jobs on a dedicated processor onto a virtual processor.

To establish the latter, virtual scheduling requires an initial delay of the processor supply to prevent that the virtual time within a component runs ahead of the real time. Replicating just the precedence order allows to lift this implementation requirement, assuming that a component implements its timing constraints in a sustainable manner. A sustainable system [29] that satisfies its timing constraints under its worst-case specifications, keeps satisfying its timing constraints when its real behavior is better than it is in the worst-case. In line with this notion of a sustainable system by Baruah and Burns [29], we define a sustainable order of the execution of jobs.

Definition 6: The precedence order of a job $J_{i,k}$ and a job $J_{j,\ell}$ is sustainable, if both jobs $J_{i,k}$ and $J_{j,\ell}$ execute in the same precedence order independent of whether or not one or more jobs $J_{x,y}$ execute less than their WCET.

Theorem 2: Let the precedence order of a job $J_{i,k}$ and a job $J_{j,\ell}$ be sustainable. Also let all the jobs of a component C finish their execution at least Δ time units prior to their deadline on a dedicated α-speed processor. Then, component C will execute the jobs $J_{i,k}$ and $J_{j,\ell}$ in the same order on a virtual-scheduled bounded-delay partition with parameters $(\alpha,\ \Delta)$ that satisfies $\texttt{asf}(t) \geq \alpha(t-\Delta)$ without missing any deadline.

Proof: According to Theorem 1, this theorem holds if $\texttt{asf}(t)$ is constrained by (4). Now let us violate (4) by assuming $\texttt{asf}(t) > \texttt{sbf}(t+\Delta) = \alpha t$, i.e., the average speed

γ of the virtual processor is $\alpha < \gamma \leq 1$ over time interval $[0,\ t]$. The WCET of any job of the tasks $\tau_i \in \mathcal{T}$ that execute in $[0,\ t]$ may therefore decrease from $\frac{1}{\alpha}E_i$ time units to $\frac{1}{\gamma}E_i$ time units. Since the precedence order of $J_{i,k}$ and $J_{j,\ell}$ is sustainable, both jobs execute in the same order when one or more jobs execute less than their WCET (Definition 6). Hence, $J_{i,k}$ and $J_{j,\ell}$ execute also in the same order under virtual scheduling, if $\mathtt{asf}(t) \geq \mathtt{sbf}(t)$, compared to a dedicated α-speed processor. $\square$

Reconsider the example in Figure 2: the precedence order of the jobs $J_{2,1}$ and $J_{3,1}$ is not sustainable, because their execution order may change when job $J_{1,1}$ executes less than its WCET. The same happens on a continuous processor, however. The relative execution order of the jobs $J_{2,1}$ and $J_{3,1}$ can therefore be considered irrelevant for the correctness of the schedule. Reconsidering the motivating example in Section 3.2, the order of all jobs are sustainable. Many off-line schedules implicitly implement the precedence constraints in this way, e.g., [1], [2] and [17] are therefore by definition sustainable.

According to Theorem 2, resources can be supplied earlier without affecting the precedence order of jobs, if the execution order of tasks is fully determined by (*i*) the timely supply of resources in order to meet deadlines and (*ii*) the arrival times of jobs. In other words, sustainable precedence relations remain valid when the processor speed is increased. Virtual scheduling then takes care of those precedence relations in the presence of delays in the processor supply.

5 Component-level scheduling

The rules of virtual scheduling, as specified in the previous section, are responsible for separating the events relative to their occurrences in real time and the delivered processor supply between them. Only after an event is delivered to the destined partition, a corresponding job is released and that job can subsequently contend with earlier released jobs for the processor time supplied by its partition. Given a task set that has programmed separations of events associated to the execution of its tasks, virtual scheduling then replicates the precedence order of jobs of the local tasks upon a partition under any work-conserving scheduling policy. This section discusses the local schedulers that can be used by the individual components.

Consider a component that can be scheduled successfully upon a bounded-delay partition (α, Δ). We recall that the bounded-delay model of Feng and Mok [6] determines Δ from the blocking tolerance β of a task set while it is running upon a continuous α-speed processor. From Theorem 1 we already concluded that this component then also makes its deadlines upon an (α, Δ) partition when we add virtual scheduling to it. The remaining question considered in this section is how to derive an appropriate timing interface that summarizes the processor requirements of a component. We distinguish two aspects here: (*i*) the choice of an interface representation, i.e., determined by the choice of a resource-supply model, and (*ii*) the selection of the interface parameters under the auspices of the chosen resource-supply model and the scheduling policy.

For these purposes, we first derive an exact condition for virtual scheduling based on the *deadline tolerance* of tasks[2] under an arbitrary work-conserving local scheduling policy.

Definition 7: Assume a task set $\mathcal{T}$ has an entire α-speed processor at its disposal. The *deadline tolerance* δ_i of a task $\tau_i \in \mathcal{T}$ is given by $\delta_i = D_i - WR_i^{\alpha}$, where WR_i^{α} denotes the worst-case response time of task τ_i upon this α-speed processor.

Stated differently, the deadline tolerance δ_i is the amount that the deadline D_i of task τ_i can be reduced without missing a deadline upon a continuous α-speed processor. The deadline tolerance of a task set $\mathcal{T}$ is then given by

$$\delta \stackrel{\text{def}}{=} \min\{\delta_i \mid 1 \leq i \leq n\}. \tag{19}$$

We observe that Theorem 1 already makes use of the deadline tolerance implicitly. Using Definition 7 of the deadline tolerance and Definition 3 of the blocking tolerance, the following two corollaries follow directly from Theorem 1.

Corollary 2: Under any work-conserving scheduling policy, the blocking tolerance β_i of a task τ_i is at most equal to its deadline tolerance δ_i, i.e. $\forall i : 1 \leq i \leq n : \beta_i \leq \delta_i$.

Corollary 3: Under any work-conserving scheduling policy, the blocking tolerance β of the task set $\mathcal{T}$ is at most equal to its deadline tolerance δ.

Theorem 3: Given a component C comprised of a task set $\mathcal{T} = \{\tau_i \mid 1 \leq i \leq n\}$ and a work-conserving scheduling policy SP. Component C can execute on a (α, Δ) bounded-delay partition under the auspices of virtual scheduling without missing any deadlines, if and only if $\Delta \leq \delta$.

Proof: Only-if direction: delaying a job of task τ_i for a longer duration than $D_i - WR_i^{\alpha}$ time units may obviously lead to deadline misses upon an α-speed processor, no matter whether processor time is supplied continuously or discontinuously.

If direction: Theorem 1 assures the validity of the schedule under virtual scheduling and a corresponding release delay of jobs of at most δ time units. □

The scheduling condition derived in Theorem 3 uses the response times of tasks upon a continuous α-speed processor in order to derive the component's interface (α, Δ). The way of computing the response time WR_i^{α} depends on the scheduling policy SP of the component and it assumes that the processor supply progresses linearly and continuously with a slope α over time. Firstly, we tailor Theorem 3 for more specific work-conserving scheduling policies (EDF and FPS). Secondly, we look at resource-supply models that represent a discontinuous processor differently than our linear approximation does.

2. In literature, e.g., [6], the term *jitter tolerance* may refer either to our term *blocking tolerance* or to our term *deadline tolerance*. In any case, we find the term jitter confusing. If a job $J_{j,\ell}$ would indeed have a jitter as large as δ_j, then lower priority jobs $J_{i,k}$ may miss their deadline. Contrary to the jitter tolerance, the blocking tolerance of a job remains valid even if other jobs are blocked maximally and, similarly, the deadline tolerances of other tasks remain valid even if one task's deadline is decreased with δ time units.

5.1 Priority-based scheduling policies

For optimal uni-processor scheduling policies (like EDF) replicating the execution order is unnecessary for satisfying deadline constraints. Bertogna et al. [30] also observed that replicating the same execution order and amount of execution time of jobs on a virtual processor compared to a dedicated processor is just sufficient to satisfy deadline constraints. As shown by the example in Section 3.2, however, even optimal scheduling policies that satisfy deadline constraints regardless of a discontinuous or continuous resource supply require a mechanism to preserve the precedence order of jobs as well.

Since EDF is work-conserving and an optimal scheduling policy, it can utilize its allocated processor share entirely, regardless of any bounded delays in the processor supply.

Theorem 4: Given a set of n deadline-constrained sporadic tasks, $\mathcal{T} = \{\tau_i \mid 1 \leq i \leq n\}$, and an EDF scheduler. The task set $\mathcal{T}$ can be *virtually scheduled* upon an (α, Δ) partition by EDF without missing any deadline if and only if

$$\forall t \ : t \geq 0 \ : \texttt{dbf}(t) \quad \leq \quad \alpha(t - \Delta), \tag{20}$$

where the cumulative processor demand bound is defined as

$$\texttt{dbf}(t) \quad = \quad \sum_{1 \leq i \leq n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor E_i. \tag{21}$$

Proof: We prove this theorem with the help of Lemma 4. Since Lemma 4 shows that $\delta = \beta$ for EDF scheduling of tasks upon a continuous processor, this theorem follows from the regular bounded-delay criterion presented by [6]. □

Lemma 4: The deadline tolerance, δ, of a task set $\mathcal{T}$ is equal to its blocking tolerance, β, upon a continuous α-speed processor scheduled by EDF. In short, $\delta = \beta$ under EDF.

Proof: We prove this lemma by contradiction. From Corollary 3 it follows that $\beta \leq \delta$. Thus, suppose $0 \leq \beta < \delta$ and there exists a job that misses a deadline when its release is delayed for δ time units, and it does not do so when it is blocked for β time units, upon a continuous α-speed processor.

Let a job $J_{i,k}$ arrive at time $a_{i,k}$ with a release $r_{i,k} = a_{i,k} + \delta$, so that $d_{i,k} = r_{i,k} + D_i - \delta$. If job $J_{i,k}$ misses its deadline $d_{i,k}$, then by virtue of EDF holds $\texttt{dbf}(d_{i,k} - a_{i,k}) = \texttt{dbf}(D_i) > \alpha(D_i - \delta)$. Since all jobs of τ_i complete their execution at least δ time units prior to their deadline D_i upon a continuous α-speed processor (see Definition 7), the cumulative execution demand $\delta + \frac{1}{\alpha}\texttt{dbf}(L)$ of task set $\mathcal{T}$ in any arbitrary time window of length L cannot exceed L, i.e., $\forall L \ : L \geq 0 : \ \delta + \frac{1}{\alpha}\texttt{dbf}(L) \leq L$. Using Definition 3 of β and $0 \leq \beta \leq \delta$, the lemma follows. □

Lemma 4 presents a unique property of the analysis of feasible bounded-delay partitions scheduled optimally by EDF, independent of our support for virtual scheduling.

Under non-optimal scheduling policies, deadline constraints can also be violated when a component is moved from a continuous to a discontinuous resource supply. In the example of Figure 2(b), job $J_{3,1}$ of task τ_3 misses its deadline when the release of job $J_{1,2}$ coincides with its arrival at time $t = 6$. Since discontinuities in the resource supply may block the execution of jobs, under some scheduling policies (like FPS) one might need to increase the speed of the virtual processor compared to the speed of the dedicated processor in order to satisfy deadline constraints of tasks. For example, [10] and [11] take this approach by redoing the local schedulability analysis of a component and taking Δ as a blocking factor, β, on the target unit-speed processor. However, a faster virtual processor with service delays does still not guarantee the same precedence order of jobs; this is where virtual scheduling comes into play.

Virtual scheduling satisfies deadline constraints even without increasing the virtual processor speed, i.e., it avoids that deadline constraints must be validated again on a virtual processor. Hence, at least the same task sets can be scheduled without deadline misses on a bounded-delay partition with and without virtual scheduling. Moreover, our rules to guard the releases of jobs in virtual time replicate the execution order of jobs on a dedicated processor onto a virtual processor and these rules are transparent for the component.

5.2 Periodic versus bounded-delay resource-supply models

Virtual scheduling reconstructs the releases of jobs on a continuous α-speed processor upon an (α, Δ) bounded-delay partition. For this purpose, it may delay the release of a job in order to compensate for delays in the processor supply to earlier released jobs. The progress of the resource supply to the tasks of a component therefore appears as linear over time.

Different from the bounded-delay resource supply that progresses linearly, the periodic resource model $\Gamma(\Pi, \Theta)$ – as proposed by Shin and Lee [4] – supplies Θ time units of processor time every period of Π time units and Θ is supplied no later than Π time units from the start of the period Π.

Independent of the use of virtual scheduling and no matter the local scheduling policy, a bounded-delay interface (α, Δ) can be converted into a periodic interface $\Gamma(\Pi, \Theta)$ as follows [6]:

$$\Pi = \frac{\Delta}{2(1-\alpha)}. \tag{22}$$

This period assignment implicitly assigns a budget $\Theta = \alpha \cdot \Pi$. The obtained $\Gamma(\Pi, \Theta)$ can be convenient in the context of an RTOS with support for a CSF. We recall that the way of determining Δ may depend on whether or not virtual scheduling is applied. Since virtual scheduling may allow for larger Δ (Corollary 3), the sizes of period Π may also become larger.

The role of a resource-supply model is important for meeting deadlines, because it bounds the delays and the received processor supply. This can be done in different ways, e.g., [11] and [10] compare the bounded-delay and the periodic resource model.

However, the relative performance of resource models is non-trivial, regardless of virtual scheduling.

With our virtual-scheduling support, a task set requires a non-increasing processor bandwidth on its bounded-delay partition. Sometimes, it requires even less budget to satisfy task deadlines than the periodic resource model requires. In Figure 2, the task set needs a periodic resource $\Gamma(2, 1.2)$ according to Shin and Lee [4]. With virtual scheduling $\Gamma(2, 1)$ suffices, which reduces the allocated processor bandwidth with 20%. Although virtual scheduling can use the bounded-delay model tightly, we cannot always gain bandwidth compared to other resource-supply models, because the relative strengths of the resource-supply models are unchanged by virtual scheduling.

6 Conclusions

A virtual platform abstracts all resources required by the hosted component, so that they appear to be available without the need of sharing them with other components. Several virtual platforms can then be scheduled together on the physical resources. On such a shared platform, however, the actual processor is faster than the virtual processor and the virtual processor delivers the processor time discontinuously. This disturbs the desired view of a virtual processor, because the order in which external events are handled within the component during run time may change with respect to the execution order on a continuous processor. This paper proposed a solution for this problem, called *virtual scheduling*.

By using virtual scheduling, a component executes on a virtual platform and it only experiences that the processor speed is different from the actual processor speed. A legacy component, which was once validated on that dedicated slower processor, can therefore be integrated on a shared processor without requiring access to the code or specification of its internal tasks to repeat the component-level analysis. The reason is that virtual scheduling guarantees that the integrated component satisfies its tasks' deadline constraints and that it executes jobs in the same order as on the dedicated slower processor, regardless of the actual supply of processor resources. Unfortunately, it is hard to abstract entirely from the resource-supply model during the timing analysis of a component, because delays in the processor supply must be bounded. With virtual scheduling, however, we can focus on the traditional scheduling problems of meeting precedence constraints and deadline constraints of tasks on a continuous uni-processor.

As a second advantage, virtual scheduling delivers external events to a component when it is executing on the processor. This prevents the handling of events destined for suspended components; thus, we avoid that event-handling consumes resources allocated to other components. As a result, a component developer can be unaware that the processor and other event-driven resources may need to be shared with other real-time components. Because of these two major advantages, we believe that virtual scheduling can considerably improve the design flow of hardware and software products applying virtualization techniques, e.g, hypervisors hosting and separating software components with different levels of timing criticality.

REFERENCES

[1] R. Dobrin, G. Fohler, and P. Puschner, "Translating off-line schedules into task attributes for fixed priority scheduling," in *RTSS*, Dec. 2001, pp. 225–234.

[2] D. Isović and G. Fohler, "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints," in *RTSS*, Nov. 2000, pp. 207–216.

[3] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proc. of the IEEE*, vol. 98, no. 4, pp. 603–620, April 2010.

[4] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM TECS*, vol. 7, no. 3, pp. 1–39, 2008.

[5] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *RTSS*, Dec. 2007, pp. 129–138.

[6] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in *RTSS*, Dec. 2002, pp. 26–35.

[7] E. Wandeler and L. Thiele, "Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling," in *EMSOFT*, Sept. 2005, pp. 80–89.

[8] W. Wang, A. K. Mok, and G. Fohler, "Pre-scheduling," *RTSJ*, vol. 30, pp. 83–103, 2005.

[9] L. Almeida and P. Peidreiras, "Scheduling with temporal partitions: response-time analysis and server design," in *EMSOFT*, Sept. 2004.

[10] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *JEC*, vol. 1, no. 2, pp. 257–269, April 2005.

[11] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *RTSS*, Dec. 2004, pp. 57–67.

[12] S. Matic and T. Henzinger, "Trading end-to-end latency for composability," in *RTSS*, Dec. 2005, pp. 98–110.

[13] M. Anand, "Conditional models for compositional design of real-time embedded systems," Ph.D. dissertation, Univ. of Pennsylvania, May 2008.

[14] S. Baruah, "Feasibility analysis of recurring branching tasks," in *ECRTS*, June 1998, pp. 138–145.

[15] T. Baker, "Stack-based scheduling of realtime processes," *RTSJ*, vol. 3, no. 1, pp. 67–99, March 1991.

[16] M. Spuri and J. Stankovic, "How to integrate precedence constraints and shared resources in real-time scheduling," *IEEE TC*, vol. 43, no. 12, pp. 1407–1412, Dec. 1994.

[17] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *RTAS*, April 2010, pp. 301–310.

[18] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A compositional scheduling framework for digital avionics systems," in *RTCSA*, Aug. 2009, pp. 371–380.

[19] J. Sun and J. Liu, "Synchronization protocols in distributed real-time systems," in *ICDCS*, May 1996, pp. 38–45.

[20] R. Rajkumar, "Dealing with suspending periodic tasks." *IBM Thomas J. Watson Research Center*, June 1991.

[21] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS*, Dec. 2008, pp. 232–243.

[22] D. Kim, Y.-H. Lee, and M. Younis, "Spirit-μkernel for strongly partitioned real-time systems," in *RTCSA*, Dec. 2000, pp. 73–80.

[23] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *EDCC*, April 2010, pp. 67–72.

[24] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *ECRTS*, July 2005, pp. 89–97.

[25] VMware, "Timekeeping in VMware virtual machines - VMware ESX 4.0/ESXI 4.0, VMware Workstation 7.0," *Information guide*, May 2010.

[26] A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," in *Advances in Real-Time Systems*, S. Son, Ed. Prentice-Hall, 1994, pp. 225–248.

[27] A. Mok, X. Feng, and D. Chen, “Resource partition for real-time systems,” in *RTAS*, May 2001, pp. 75–84.
[28] V. Lortz and K. Shin, “Semaphore queue priority assignment for real-time multiprocessor synchronization,” *IEEE TSE*, vol. 21, no. 10, pp. 834 – 844, Oct. 1995.
[29] S. Baruah and A. Burns, “Sustainable schedulability analysis,” in *RTSS*, Dec. 2006, pp. 159–168.
[30] M. Bertogna, N. Fisher, and S. Baruah, “Resource-sharing servers for open environments,” *IEEE TII*, vol. 5, no. 3, pp. 202–219, Aug. 2009.

PAPER C:

OPAQUE ANALYSIS FOR RESOURCE-SHARING COMPONENTS IN HIERARCHICAL REAL-TIME SYSTEMS

M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte

Submitted to the real-time systems journal.

Abstract

Hierarchical scheduling frameworks (HSFs) have been developed to enable composition and reuse of independently developed and analyzed real-time components in complex systems. In practice, these components share more resources than just the processor, requiring arbitration through a global (system-level) synchronization protocol.

In this paper we propose opaque analysis methods to integrate resource-sharing components into uni-processor HSFs. A local (component-level) schedulability analysis is opaque if it is independent (or agnostic) of the global synchronization protocol. An individual component can therefore be analyzed as if all resources are entirely dedicated to it. This locality of the analysis obtained from opacity enables us to derive a computationally tractable method for exploring and selecting the design parameters of resource-sharing components with the objective to minimize the system load. Moreover, given a real-time interface of a component that is derived by means of an opaque analysis, the component can be used with an arbitrary global synchronization protocol. Hence, opacity extends the independence of a component of the global scheduling model, thereby effectively increasing the reusability of the component.

To arbitrate resource access between components, we consider four existing protocols: SIRAP, BROE and HSRP - comprising overrun with payback (OWP) and overrun without payback (ONP). We classify local analyses for each synchronization protocol based on the notion of opacity and we develop new analysis for those protocols that are non-opaque.

Finally, we compare different analyses for SIRAP, ONP, OWP and BROE by means of an extensive simulation study. From the results we derive guidelines for selecting a global synchronization protocol.

1 Introduction

Hierarchical scheduling frameworks (HSFs) have been developed to enable composition and reusability of real-time components in complex systems. The increasing complexity of those systems demands a decoupling of (*i*) development and analysis of individual components and (*ii*) integration of components on a shared platform, including analysis at the system level. An HSF provides temporal isolation between components by allocating a *processor budget* to each component. A component that is validated to meet its timing constraints when executed in isolation will continue meeting its timing constraints after integration (or admission) on a shared uni-processor platform. The HSF is therefore a promising solution for industrial standards which more and more specify that an underlying operating system should prevent timing faults in any component to propagate to other components on the same processor.

The main goal of compositional real-time scheduling frameworks (Shin and Lee, 2008) is establishing global (system level) timing properties by composing independently specified and analyzed local (component level) timing properties, thus, improving the reusability of components. Local timing properties are analyzed by assuming a worst-case supply of processor resources to a component. The way of modeling the processor provisioning to a component is defined by a resource-supply model, e.g., the periodic resource model by Shin and Lee (2008) or the bounded-delay model by Feng and Mok (2002). These models make it possible to combine deadline constraints of all tasks within a component and abstract from the way tasks are locally scheduled, so that a component can be represented by a single real-time constraint, called *a real-time interface*. Components can be composed by combining a set of real-time interfaces, which will treat each component as a single task by itself.

The global scheduling environment (a parent component) can provide more resources to its (child) components than just processor resources. For example, components may use operating system services, memory mapped devices and shared communication devices requiring mutually exclusive access. An HSF with support for resource sharing makes it possible to share serially accessible resources (from now on referred to as resources) between arbitrary tasks, which are located in arbitrary components, in a mutually exclusive manner. A resource that is only shared by tasks within a single component is a *local shared resource*. A resource that is used in more than one component is a *global shared resource*. Any access to a resource is assumed to be arbitrated by a synchronization protocol.

If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of its budget, excessive blocking periods can occur which may hamper the correct timeliness of other components. To prevent such budget depletion during global resource access[1] (see Figure 1), four synchronization protocols have been proposed based on the stack resource policy (SRP) by Baker (1991). These are based on two general mechanisms:

1) *self-blocking* when the remaining budget is insufficient to complete a global resource access entirely - having two flavors called (*i*) the subsystem integration and resource allocation policy (SIRAP) by Behnam et al. (2007) and (*ii*) the bounded-delay resource open environment (BROE) by Bertogna et al. (2009b) - or
2) *overrun* the budget until the resource is released - called the hierarchical stack resource policy (HSRP) by Davis and Burns (2006). HSRP has two flavors: overrun with payback (OWP) and overrun without payback (ONP). The term *without payback* means that the additional amount of budget consumed during an overrun does not have to be returned in the next budget period.

Although these protocols prevent budget depletion during global resource access, in order to do so, they may need to deliver processor resources differently. This, on its turn, may add constraints to the supply of processor resources in order to preserve local deadline constraints of tasks.

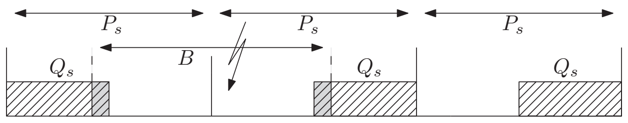

Figure 1. When the budget Q_s (allocated every period P_s) of a task depletes while a task executes on a global resource, tasks in other components may experience excessive blocking durations, B.

In practical situations, a component developer is typically unconcerned about the sharing scope of resources. A component may access resources for which just local usage or shared global usage is determined only upon integration of components onto a shared processor. Fortunately, the syntax of the primitives for accessing local and global resources can be the same, even though the synchronization protocols are different. The actual binding of function calls to scope-dependent synchronization primitives, that arbitrate either global or local resource access, can be done at compile time or when the component is loaded. Dynamic binding of primitives makes it possible to decouple the specification of global resources in the interface from their use in

1. For the same reason, SRP's task model by Baker (1991) requires that the same job of a task that accesses a resource also releases it. In order to be able to reuse the schedulability analysis of tasks in the hierarchy of components (i.e., we treat a component as a task), an access to a resource must be followed by a release within the same job of the component.

the implementation. This flexible decoupling of the sharing scope of resources in the application's programming interface is called *opacity* by López Martinez et al. (2010) and it abstracts whether or not a resource is global in the system.

This paper presents an extension of this notion of opacity to component analysis and the corresponding derivation of a real-time interface of a component. Opacity requires that the implementation of a component, as well as the way in which interface parameters are derived (the local analysis), are unaware of the global synchronization protocol, so that components cannot make use of any knowledge about the constraints and modifications in the processor supply to a component imposed by the global synchronization protocol. By definition of opacity, all computed interface parameters of a component are made independent of a global synchronization protocol. This effectively increases the reusability of the component.

In order to create a clean separation between local and global scheduling analysis, we call the local analysis of a component *opaque* with respect to its global (parent) component, if

1) it is unaware of global resource arbitration;
2) in the local analysis, global resources are treated the same as local resources;

An opaque local analysis remains parameterized by a resource-supply model, but, contrary to existing compositional models with non-opaque local analysis, it allows us to post-pone the classification of shared resources into global and local until component integration time. Moreover, an opaque analysis makes it possible to defer the choice for a global synchronization protocol (if global sharing is necessary at all) until component integration time.

Contributions: The first and main contribution of this paper is introducing opaque analysis and leveraging the concept of an opaque analysis to a computationally efficient methodology for exploring an overall resource-efficient system configuration. Our methodology consists of three steps: a step where the component analysis is abstracted from the resource-supply model and two optimization steps where local blocking can be traded for global blocking.

The notion of opaque local analysis yields the following other contributions:

- We survey the existing analyses for HSFs in the presence of shared resources and we characterize the compliance to opacity of each of the analysis.
- We develop new local analysis for those protocols that are non-opaque, i.e,
  - We develop a new analysis for OWP. Our analysis reduces the pessimism of OWP, it is opaque and, in most cases, it is better than ONP.
  - Given the non-opaque analysis of SIRAP, we show that the overruns of ONP can be used as an upper bound for the self-blocking durations of SIRAP regardless of the local scheduling policy. As a consequence, an

opaque analysis for ONP provides also an opaque analysis technique for resources arbitrated by an implementation of SIRAP.

- We are the first to present an extensive experimental comparison of the different analysis techniques for BROE, SIRAP, OWP and ONP. We do not only evaluate the individual protocols, we also evaluate the effect of using an opaque analysis for them.
- Finally, we derive new guidelines for selecting a synchronization protocol in hierarchical real-time systems.

Organization: The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes our system model. Section 4 recapitulates the mechanisms for global sharing of the synchronization protocols considered in this paper (i.e., HSRP, SIRAP and BROE). Section 5 defines the notion of opacity and accordingly classifies the existing compositional analysis for HSFs in the presence of shared resources. Section 6 presents an improved, simplified and opaque analysis for overrun with payback (OWP). Section 7 presents an opaque local analysis for SIRAP. Section 8 presents a methodology which shows how an opaque analysis allows for a tractable design-space exploration of resource-sharing components. Section 9 evaluates the different analyses and the different protocols for global resource sharing by means of a simulation study. We investigate how global resource sharing impacts the schedulability of an individual component and, subsequently, how it impacts the schedulability of an entire system. From these results we derive guidelines for selecting a global synchronization protocol. Finally, Section 10 concludes this paper.

2 RELATED WORK

In hierarchically scheduled systems, a group of recurring tasks, forming a component, is mapped on a reservation; reservations were originally introduced by Mercer et al. (1994) and Rajkumar et al. (1998). We first review existing works on hierarchical scheduling of independent components. Secondly, we lift the assumption on the independence of components. We discuss how tasks may share resources with other tasks, either within the same component or located in other components. This means that resource sharing expands across reservations which calls for specialized resource access protocols. Finally, we discuss the relation and difference between two important quality properties of real-time synchronization protocols, i.e., transparency and opacity.

2.1 Timing interfaces of independent real-time components

The increasing complexity of real-time systems led to a growing attention for compositional analysis. Deng and Liu (1997) proposed a two-level HSF for open systems, where components may be independently developed and validated. The corresponding schedulability analysis have been presented in Kuo and Li (1999) for fixed-priority preemptive scheduling (FPPS) and in Lipari and Baruah (2000) for earliest-deadline-first (EDF) global schedulers.

One of the challenges of real-time composition is deriving a timing interface for a component, i.e., separate the component's internals from its global timing parameters by allocating a guaranteed processor share. Wandeler and Thiele (2005) calculate demand and service curves for components using real-time calculus. Shin and Lee (2008) proposed the periodic resource model to specify periodic processor allocations to components. The explicit-deadline periodic (EDP) resource model by Easwaran et al. (2007) extends the periodic resource model of Shin and Lee (2008) by distinguishing the relative deadline for the allocation time of budgets explicitly. The bounded-delay model by Feng and Mok (2002) describes linear service curves with a bounded initial service delay.

Many works presented approximated (e.g., Almeida and Peidreiras, 2004; Lipari and Bini, 2005; Fisher and Dewan, 2012) and exact (e.g., Shin and Lee, 2008; Easwaran et al., 2007; Lipari and Bini, 2005) budget allocations for the bounded-delay and periodic resource models under preemptive scheduling policies. Both Lipari and Bini (2005) and Shin and Lee (2004) have presented methods to convert the bounded-delay model into a periodic resource model.

Unlike the models by Shin and Lee (2008), Easwaran et al. (2007) and Feng and Mok (2002), various models deviate from the principle of locality of schedulability analysis as they make assumptions on the resource supply of other components in the system in order to derive response times, e.g., see the models by Davis and Burns (2005); Bril et al. (2006); Balbastre et al. (2009). Although these alternative models tighten the response-time analysis of tasks, unfortunately, the complexity of the analysis also increases for these models.

2.2 Task synchronization in hierarchically scheduled systems

In literature several alternatives are presented to accommodate resource sharing between tasks in reservation-based systems. de Niz et al. (2001) support this in their fixed-priority preemptively scheduled (FPPS) Linux/RK resource kernel based on the immediate priority ceiling protocol (IPCP) by Sha et al. (1990) and de Niz et al. (2001) propose a mechanism for temporal protection against misbehaving critical sections. Steinberg et al. (2005) implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach is described in Lipari et al. (2004) for earliest-deadline-first (EDF)-based systems and termed bandwidth-inheritance (BWI).

BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol (PIP) by Sha et al.

(1990) and when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. BWI does not require a-priori knowledge of tasks, i.e., no ceilings need to be precalculated. BWI-like protocols are not very suitable for arbitrating hard real-time tasks in HSFs, because the worst-case interference of all tasks in other components that access global resources needs to be added to a component's budget at integration time in order to guarantee its internal tasks' schedulability. This leads to pessimistic budget allocations for hard real-time components.

A prerequisite to enable independent analysis of each of the resource-sharing components and their integration is the knowledge of which resources a task will access (Shin and Lee, 2008). To accommodate such resource sharing, three synchronization protocols have been proposed based on the SRP from Baker (1991), i.e., HSRP (Davis and Burns, 2006), SIRAP (Behnam et al., 2007) and BROE (Bertogna et al., 2009b). Unlike HSRP's and SIRAP's analysis, however, the global schedulability analysis of BROE is limited to EDF and cannot be generalized to include other scheduling policies.

The overrun mechanism (with payback) was first introduced by Ghazalie and Baker (1995) in the context of aperiodic servers. This mechanism was later re-used in HSRP in the context of two-level HSFs by Davis and Burns (2006) and complemented with a variant *without* payback. Although the analysis presented by Davis and Burns (2006) does not integrate in HSFs due to the lacking support for independent analysis of components, this limitation is lifted by Behnam et al. (2010c).

The idea of self-blocking has also been considered in different contexts, e.g., see Caccamo and Sha (2001) for supporting soft real-time tasks and see Holman and Anderson (2002) for a zone-based protocol in a pfair scheduling environment. The SIRAP by Behnam et al. (2007) uses self-blocking for hard real-time tasks in HSFs on a single processor and its associated analysis supports composability. In Behnam et al. (2010a) the original SIRAP analysis by Behnam et al. (2007) has been significantly improved for arbitrating multiple shared resources. We will show that the strength of SIRAP's analysis comes from its detailed system model, making it difficult to analyze components opaquely with little timing characteristics.

The original SIRAP analysis by Behnam et al. (2007) and the compositional HSRP analysis by Behnam et al. (2010c) have been analytically compared with respect to their impact on the system load for various component parameters by Behnam et al. (2009a). The performance of each protocol heavily depends on the chosen system parameters. Moreover, these results suggest that HSRP's overrun mechanism with payback (OWP) is hardly beneficial compared to overrun without payback (ONP). This observation is contradictory with the recommendations of Davis and Burns (2006). Our new analysis makes the results in Behnam et al. (2009a) obsolete and we develop new guidelines, which consider also BROE, to select a synchronization protocol in two-level HSFs.

2.3 Opacity versus transparency

Transparency means that synchronization primitives of one protocol can be exchanged by another protocol without introducing new kernel primitives in the application's programming interface (Buttazzo, 2005). The traditional example for transparent resource sharing in real-time systems is the priority-inheritance protocol (PIP) by Sha et al. (1990) which transparently improves the predictability of blocking durations compared to binary semaphores. For example, the PCP by Sha et al. (1990) and the SRP by Baker (1991) are non-transparent with respect to binary semaphores, because resource ceilings need to be precalculated even to obtain the basic protocol functionality of mutual exclusive resource access.

Although the traditional notion of transparency does not consider a hierarchy of schedulers, BWI makes it possible for tasks located in arbitrary components to share resources transparently. Looking only at SRP-compliant protocols, van den Heuvel et al. (2012) have implemented HSRP, SIRAP and BROE in a real-time micro-kernel by means of transparent primitives, enabling the system integrator to select a synchronization protocol to optimize the overall system performance. Transparent synchronization primitives for these SRP-based protocols hide for the components which flavor of the SRP is used at the global level to arbitrate shared resources.

Whereas transparency is a property of a protocol which mainly influences the development cycle of a component, opacity influences both the development cycle (e.g., see López Martinez et al., 2010; van den Heuvel et al., 2012) and the analysis cycle of a component (the topic of this paper). From a development perspective, opaque synchronization primitives hide the sharing scope of resources during the component development and postpone binding of scope-dependent primitives until integration time. At the end, the kernel primitives for synchronizing local and global resources need to be different, because local resource sharing should never interfere with the global schedule. From an analytical perspective, opacity introduces an orthogonal level of abstraction with respect to protocol transparency which is similar to the abstraction of scheduling policies of independent tasks in HSFs by Deng and Liu (1997). That is, each component can be deployed with a different synchronization protocol, e.g., PIP, PCP or SRP, to arbitrate access to local resources and the local protocols are independent and unconscious of the global synchronization protocol.

If, and only if, a global resource is actually shared between components, it must be arbitrated by a global synchronization protocol. The inherent changes in the supply of processor resources by a particular synchronization protocol, which prevent budget depletion during global resource access, can lead to untruthful interface specifications under a non-opaque component-level analysis.

For example, consider a component manufacturer which delivers a component with a corresponding interface description specifying the component's resource requirements. We shall show (see Example 2) that the manufacturer may create artificial dependencies on global resources which allow him to optimize the required processor bandwidth of the component by means of - what we call - a non-opaque analysis. Even if the system integrator identifies no other component accesses the same global resources, it might be unclear whether and how the local analysis has affected the processor requirements captured in the component's interface. In other words, the local optimizations based on the specified resource requirements of a component may give an impression of resource efficiency and, nevertheless, the system integrator may encounter global scheduling penalties for the artificially created global dependencies.

Moreover, if a component developer would make use of properties specific to a global synchronization protocol, then the resulting component may violate its timing properties after the global synchronization protocol is changed. Since the analysis of components is different for each global synchronization protocol, we need to develop a common notion of opacity which yields a determinate way of deriving component interfaces[2]. These interfaces should be independent of the global scheduling environment in order to be general enough to select an arbitrary global synchronization protocol.

3 Real-time scheduling model

A system contains a single processor and a set $\mathcal{R}$ of M serially accessible global resources R_1, ..., R_M. The processor and these resources need to be shared by N components, C_1, ..., C_N, each comprising a sporadic, deadline-constrained task set. A unique system-level (global) scheduler selects which component, and when a component, is executed on the shared processor. The component-level (local) scheduler decides which of the tasks of the executing component is allocated the processor. The global scheduler and each of the local schedulers of individual components may apply different scheduling policies. As scheduling policies, we consider earliest deadline first (EDF), an optimal dynamic uniprocessor scheduling algorithm, and the deadline-monotonic (DM) algorithm, an optimal fixed-priority preemptive scheduling (FPPS) algorithm.

2. Opacity does not change the amount of parameters contained within an interface (i.e., the size of the interface is unchanged); but it makes the way of computing the interface parameters of a component independent of the global synchronization protocol.

The stack resource policy (SRP) by Baker (1991) is used to arbitrate access to shared resources between components at the global level; similarly, the SRP is used at the local level to arbitrate access to shared resources between tasks locally.

Since each component specifies an interface to abstract its task-set's resource requirements, this interface can be used to determine whether or not a component can be admitted together with other components on the same shared processor. If a component is admitted, a server guarantees and enforces the periodic availability of the specified resources in the interface, so that the component keeps meeting its timing constraints on the shared processor as long as it behaves as specified. If it violates its interface, it may be penalized, but other components are temporally isolated from the malignant effects.

An HSF truly executes different components in isolation in the absence of shared resources. However, if components share resources, they cannot be completely isolated from each other. For example, one component may violate its interface by executing longer on a shared resource than it has specified, so that all other components suffer from unexpectedly long blocking durations. There are strategies for containment of temporal faults to those components that share resources, e.g., de Niz et al. (2001) consider priority-inheritance-based protocols and van den Heuvel et al. (2011) consider the same SRP-based protocols as in this paper. Such strategies are beyond the scope of this paper; we look at different methods for allocating sufficient processor time to an individual component regardless of any other component, such that all the resource requirements of a component are met according to its specification.

3.1 Component and task model

Each component C_s has a dedicated budget which specifies its periodically guaranteed fraction of the processor. The timing interface of a component C_s is specified by means of a triple $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$, where $P_s \in \mathbb{R}^+$ denotes its period, $Q_s \in \mathbb{R}^+$ denotes its budget, and $\mathcal{X}_s$ denotes the set of resource holding times to global resources. The maximum value in $\mathcal{X}_s$ is denoted by X_s, where $0 \leq X_s \leq P_s$. The set $\mathcal{R}_s$ denotes the subset of global resources accessed by component C_s, where the cardinality of $\mathcal{R}_s$ is denoted by m_s (just like the cardinality of $\mathcal{X}_s$). The maximum time that a component C_s executes on the processor while accessing resource $R_\ell \in \mathcal{R}_s$ is called the resource holding time which is denoted by $X_{s\ell}$, where $X_{s\ell} \in \mathbb{R}^+ \cup \{0\}$ and $X_{s\ell} > 0 \Leftrightarrow R_\ell \in \mathcal{R}_s$.

Each component C_s contains a set $\mathcal{T}_s$ of n_s sporadic tasks $\tau_{s1}, \ldots, \tau_{sn_s}$. The timing characteristics of a task $\tau_{si} \in \mathcal{T}_s$ are specified by means of a quadruple $(T_{si}, E_{si}, D_{si}, \mathcal{H}_{si})$, where $T_{si} \in \mathbb{R}^+$ denotes its minimum inter-arrival time, $E_{si} \in \mathbb{R}^+$ its worst-case execution time (WCET), $D_{si} \in \mathbb{R}^+$ its (relative) deadline (where $0 < E_{si} \leq D_{si} \leq T_{si}$) and $\mathcal{H}_{si}$ denotes the set of its WCETs of critical sections. We assume that period P_s of component C_s is selected such that $2P_s \leq T_{si} (\forall \tau_{si} \in \mathcal{T}_s)$, because this efficiently assigns a budget to component C_s in the context of the periodic resource model of Shin and Lee (2008). For notational convenience, tasks (and components) are given in deadline-monotonic order, i.e., τ_{s1} has the smallest deadline and τ_{sn_s} has the largest deadline.

The WCET of task τ_{si} within a critical section accessing global resource R_ℓ (i.e., a value contained in $\mathcal{H}_{si}$) is denoted $h_{si\ell}$, where $h_{si\ell} \in \mathbb{R}^+ \cup \{0\}$, $E_{si} \geq h_{si\ell}$ and $h_{si\ell} > 0 \Leftrightarrow R_\ell \in \mathcal{R}_s$. The relation between the WCET of a critical section ($h_{si\ell}$) and the resource holding times ($X_{s\ell}$) of a component is further explained in Section 3.4.

3.2 Resource-supply models

The *processor supply* refers to the amount of processor resources that a component C_s can provide to its workload in order to satisfy internal deadline constraints. The supply bound function $\texttt{sbf}_{\Omega_s}(t)$ of the EDP resource model $\Omega_s = (P_s, Q_s, D_s, \mathcal{X}_s)$, that returns the minimum supply for any interval of length t, is given by Easwaran et al. (2007):

$$\texttt{sbf}_{\Omega_s}(t) = \max \left\{ \begin{array}{l} 0, \\ (h(t) - 1)Q_s, \\ t - (h(t) + 1)(P_s - Q_s) + (P_s - D_s) \end{array} \right\}, \tag{1}$$

with $h(t) = \left\lceil \frac{t - (D_s - Q_s)}{P_s} \right\rceil$. Intuitively, an EDP resource $\Omega_s = (P_s, Q_s, D_s, \mathcal{X}_s)$ gives Q_s time units of processor time every period of P_s time units and Q_s is provisioned no later than D_s time units from the start of the period P_s.

The periodic resource model $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ is a specialization of the EDP resource Ω_s with characteristics $(\Gamma_s = (P_s, Q_s, \mathcal{X}_s)) \equiv (\Omega_s = (P_s, Q_s, P_s, \mathcal{X}_s))$. The deadline parameter D_s of the EDP resource model makes it possible to express relative priorities between components. Since we refrain from making any assumptions on other components than the one under consideration, i.e., also relative priorities are unknown, we will consider the periodic resource model unless explicitly denoted differently. In this way, the resource requirements of component C_s, captured in interface Γ_s by the local analysis, are satisfied by a periodic resource supply of $\texttt{sbf}_{\Gamma_s}(t)$.

The longest interval a component may receive no processor supply on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ is named the *blackout duration*, i.e.,

$$BD_s \quad = \max\{t \mid \texttt{sbf}_{\Gamma_s}(t) = 0\} \quad = 2(P_s - Q_s). \tag{2}$$

The linear lower bound of the periodic resource with parameters $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ is given by Shin and Lee (2008):

$$\texttt{lsbf}_{\Gamma_s}(t) = \frac{Q_s}{P_s}\left(t - 2\left(P_s - Q_s\right)\right). \tag{3}$$

The $\texttt{lsbf}_{\Gamma_s}(t)$ in (3) is not only a linear approximation of the $\texttt{sbf}_{\Gamma_s}(t)$ in (1), it also models a bounded-delay resource supply as defined by Feng and Mok (2002) with a continuous, fractional provisioning of $\frac{Q_s}{P_s}$ of the shared processor (also referred to as the virtual processor speed) and a longest initial service delay of BD_s time units.

A budget parameter Q_s of interface Γ_s should be sufficient to meet deadline constraints of tasks and no other constraints, e.g., related to global synchronization, should influence the size of Q_s. The set of resource holding times - denoted in Γ_s by $\mathcal{X}_s$ - defines the amount of execution time on global resources that a component receives for an access to a resource. In other words, if component C_s is granted access to resource R_ℓ, it receives $X_{s\ell}$ time units of execution time on resource R_ℓ prior to deadline P_s.

3.3 Synchronization protocol

This paper focuses on arbitrating *global* shared resources using the SRP. To be able to use the SRP for synchronizing global resources, its associated ceiling terms need to be extended.

3.3.1 Preemption levels

Each task τ_{si} has a static preemption level equal to $\pi_{si} = 1/D_{si}$. Similarly, a component has a preemption level equal to $\Pi_s = 1/P_s$, where period P_s serves as a relative deadline. If components (or tasks) have the same calculated preemption level, then the smallest index determines the highest preemption level.

3.3.2 Resource ceilings

With every global resource R_ℓ two types of resource ceilings are associated; a *global* resource ceiling RC_ℓ for global scheduling and a *local* resource ceiling $rc_{s\ell}$ for local scheduling. These ceilings are statically calculated values, which are defined as the highest preemption level of any component or task that shares the resource. According to the SRP, these ceilings are defined as:

$$RC_\ell = \max(\Pi_N, \max\{\Pi_s \mid R_\ell \in \mathcal{R}_s\}), \tag{4}$$

$$rc_{s\ell} = \max(\pi_{sn_s}, \max\{\pi_{si} \mid h_{si\ell} > 0\}). \tag{5}$$

We use the outermost $\max$ in (4) and (5) to define RC_ℓ and $rc_{s\ell}$ in those situations where no component or task uses R_ℓ. The values of the local and global ceilings as defined in (4) and (5) by definition guarantee mutual exclusive access to their corresponding resource R_ℓ by the sharing tasks and components and, therefore, the values of these ceilings cannot be further decreased. In some situations - as further investigated in Section 8 - it might be desirable to limit preemptions more than is strictly required for mutual exclusive resource access, which can be established by increasing the value of the resource ceilings in (4) and (5) artificially.

3.3.3 System and component ceilings

The system ceiling and the component ceiling are dynamic parameters that change during execution. The system ceiling is equal to the highest global resource ceiling of a currently locked resource in the system. Similarly, the component ceiling is equal to the highest local resource ceiling of a currently locked resource within a component. Under the SRP a task can only preempt the currently executing task if its preemption level is higher than its component ceiling. A similar condition for preemption holds for components.

3.4 Resource holding times

The value of the local resource ceiling $rc_{s\ell}$ influences the *resource holding times* (introduced by Bertogna et al., 2007), i.e., the value of $X_{si\ell}$ represents the amount of processor time supplied to component C_s from the access until the release of task τ_{si} to resource R_ℓ. The resource holding time, $X_{si\ell}$, includes the cumulative processor requests of tasks within the same component C_s that can preempt τ_i while it is holding resource R_ℓ.

The way of computing resource holding times of tasks depends on the local scheduling policy, for example, see the method under local FPPS by Bertogna et al. (2007) and the method for local EDF by Bertogna et al. (2009a). Moreover, various system assumptions in the description of a particular global synchronization protocol may affect the way of computing resource holding times (e.g., see Behnam et al., 2007; Bertogna et al., 2009b; Behnam et al., 2010c). However, all these methods can be simplified and unified (independent of the local scheduling policy and the global synchronization protocol) by assuming that the component's period is smaller than the tasks' periods. The main observation leading to this simplification is that an access to a global resource must be followed by a release of the resource in the same component period, e.g., established by the self-blocking mechanisms or the overrun mechanisms considered in real-time literature; otherwise, components do not comply with the seminal model of Liu and Layland (1973) of periodic tasks in which each job (an instance) of a particular task (or component) must be independent of any of its preceding jobs. Violating this assumption would mean that we are unable to reuse the SRP analysis at the global scheduling level, yielding an analytical situation which is obviously undesirable.

If a resource must be accessed and released in the same component period which is smaller than the task periods, then we can limit the number of preemptions within a critical section and this, on its turn, will lead to a smaller resource holding time. The next lemma generalizes Lemma 1 in Behnam et al. (2010b) and Lemma 3 in Behnam et al. (2010c) for the global SRP:

Lemma 1: Given $P_s < T_s^{\mathtt{min}}$ and $T_s^{\mathtt{min}} = \min\{T_{si} \mid 1 \leq i \leq n_s\}$, all tasks τ_{sj} that are allowed to preempt a critical section accessing a global shared resource R_ℓ, i.e., $\pi_{sj} > rc_{s\ell}$, can preempt at most once during an access to resource R_ℓ when using any global SRP-compliant protocol, independent if the local scheduler is EDF or FPPS.

Proof: If a task, having a period of at least $T_s^{\mathtt{min}}$, preempts two or more times inside a critical section of resource R_ℓ, then the resource is also locked during a period of at least a length $T_s^{\mathtt{min}}$; thus, $X_{si\ell} > T_s^{\mathtt{min}}$. Since $P_s < T_s^{\mathtt{min}}$, this would mean that $X_{si\ell} > P_s$. According to the SRP (Baker, 1991), a global resource should be accessed and released by the same instance of a component, i.e., within period P_s. However, $X_{si\ell} > P_s$ yields a contradiction by requiring a component utilization of $U_s \geq \frac{X_{si\ell}}{P_s} > 1$, making the component infeasible. $\square$

Lemma 1 makes it possible to compute the *resource holding time*, $X_{si\ell}$ of task τ_{si} to resource R_ℓ as follows:

$$X_{si\ell} \quad = \quad h_{si\ell} + \sum_{\pi_{sj} > rc_{s\ell}} E_{sj}, \tag{6}$$

and the maximum resource holding time within a component C_s is computed as $X_{s\ell} = \max\{X_{si\ell} \mid 1 \leq i \leq n_s\}$.

The computed values of $X_{s\ell}$ are included in the set $\mathcal{X}_s$ which is part of the component's interface, Γ_s. We recall that opacity requires that the way of computing the interface parameters Q_s and $\mathcal{X}_s$ of a component is independent of the global synchronization protocol; Lemma 1 establishes this requirement for the set of resource holding times, $\mathcal{X}_s$, of a component.

3.5 Overview

Table 1 summarizes the notation adopted in this paper.

4 Global synchronization: prevent excessive blocking

In this section, we give a brief overview of the run-time mechanisms employed by the synchronization protocols considered in this paper. Each of the protocols applies straightforward resource arbitration by the SRP at the local level, for both local and global resources. This means that when a task has started its execution and tries to access a resource, irrespective of any other protocol specific actions for global synchronization, the local component ceiling is updated immediately as if resource access is granted.

To prevent budget depletion while a task executes on a shared resource, HSRP (Davis and Burns, 2006) allows overrunning the budget until the task

Table 1
Notation throughout this paper.

Symbol	Description
N	Number of components in the system
U	Total utilization of allocated processor bandwidth
M	Number of global resources
$\mathcal{R}$	Set of global resources
R_ℓ	ℓ-th global resource
RC_ℓ	Global resource ceiling of R_ℓ
C_s	s-th component
Π_s	Preemption level of C_s
P_s	Period of the resource allocation to component C_s
D_s	Relative deadline for the resource allocation to component C_s
$d_s(t)$	Absolute deadline at time t for the resource allocation to C_s
Q_s	Periodically allocated processor time for C_s
U_s	Allocated processor bandwidth of C_s
$\mathcal{R}_s$	Set of global resources accessed by C_s
$\mathcal{X}_s$	Set of holding times to global resources accessed by C_s
$X_{s\ell}$	the resource holding time of C_s for R_ℓ
X_s	Maximum of the resource holding times of C_s
O_s	Processor time for C_s merely dedicated to prevent excessive blocking
$rc_{s\ell}$	Local resource ceiling of R_ℓ
Γ_s	Interface of C_s defining periodic resource demands of C_s
Ω_s	Interface Γ_s of C_s constrained by deadline D_s
BD_s	Longest duration for C_s without any processor supply
$\texttt{sbf}(t)$	Minimum processor supply in any sliding window of length t
$\texttt{lsbf}(t)$	Linear lower bound of $\texttt{sbf}(t)$
$\mathcal{IC}_s$	Matrix of partial interface candidates of C_s
$\mathcal{IC}_s^{(\ell)}$	ℓ-th row of $\mathcal{IC}_s$ with partial interfaces of C_s for accessing R_ℓ
$\Gamma_{s\ell}$	Partial interface defining C_s' demands for a given resource ceiling $rc_{s\ell}$
$\mathcal{T}_s$	Task set of a component
$U(\mathcal{T}_s)$	Utilization of task set $\mathcal{T}_s$
n_s	Number of tasks composing component C_s
m_s	Number of global resources accessed by C_s
τ_{si}	i-th task of component C_s
π_{si}	Preemption level of τ_{si}
T_{si}	Minimal inter-arrival time of task τ_{si}
E_{si}	WCET of τ_{si}
D_{si}	(Relative) deadline of τ_{si}
$\mathcal{H}_{si}$	Set of WCETs of task τ_{si} on resources
I_{si}	Idle time (self-blocking) by SIRAP experienced by task $\tau_{s1} \dots \tau_{si}$
$h_{si\ell}$	WCET of τ_{si}'s largest critical section to R_ℓ
$X_{si\ell}$	Largest resource holding time of τ_{si} to R_ℓ

releases the resource. Alternatively, SIRAP (Behnam et al., 2007) and BROE (Bertogna et al., 2009b) each employ a self-blocking mechanism to prevent budget overruns by only granting resource access when there is sufficient budget left to complete the resource access.

4.1 HSRP: Budget overruns

HSRP uses an overrun mechanism when a budget depletes during a critical section. Several flavors of overrun have been compared by Behnam et al. (2010c). If a task $\tau_{si} \in \mathcal{T}_s$ has locked a global resource when its component's budget Q_s depletes, then component C_s can continue its execution until task τ_{si} releases the resource.

To distinguish this additional amount of required budget from the *normal budget* Q_s, we refer to X_s as an *overrun budget*. HSRP has two flavors: overrun with payback (OWP) and overrun without payback (ONP). The term *without payback* means that the additional amount of budget consumed[3] during an overrun does not have to be returned in the next budget period.

Although some papers on budget overruns assume $X_s \leq Q_s$ (e.g., Davis and Burns, 2006), we refrain from this assumption. In return, we assume that pay back may span multiple budget replenishments, if $X_s > Q_s$. In other words: no budget becomes available in the subsequent budget replenishments after the overrun happened until all remaining debts are paid back.

Budget overruns cannot take place across replenishment boundaries. For each component C_s, the analyses of overrun protocols require $Q_s + X_s$ processor time before its relative deadline P_s, for example, see Davis and Burns (2006) and Behnam et al. (2010c).

4.2 SIRAP: task-level self-blocking

With SIRAP (Behnam et al., 2007, 2010a) a task is only allowed to access a global resource when it has sufficient budget to complete the entire critical section. If a task attempts to access a resource and the remaining budget is insufficient, then the task blocks itself until the budget is replenished. SIRAP guarantees access to a global resource in the replenished budget subsequent to self-blocking. Essentially, a self-blocked task τ_{si} consumes at most $X_{si\ell}$ amount of idle time from the component's budget while the task is waiting for its budget to replenish. The cumulative overhead of self-blocking experienced by task τ_{si} over a time interval of length t can be computed as follows.

The SIRAP analysis by Behnam et al. (2010a) constructs a multi-set $G_{si}^{\texttt{sort}}(t)$ of self-blocking durations that a task τ_{si} may experience in a time interval of length t. The (cumulative) self-blocking term $I_{si}(t)$ of a task τ_{si} is defined as:

$$I_{si}(t) = \sum_{1 \leq l \leq z(t)} G_{si}^{\texttt{sort}}(t)[l], \tag{7}$$

3. The actually consumed amount of processor time is by definition smaller than or equal to the worst-case resource holding time $X_{si\ell}$.

where $z(t) = \left\lceil \frac{t}{P_s} \right\rceil$ defines an upper bound to the number of self-blocking occurrences within a time interval of length t and $G^{\texttt{sort}}_{si}(t)$ defines a multi-set (a set including duplicates of values $X_{si\ell}$) of self-blocking lengths that a task τ_{si} may experience by itself and other tasks τ_{sj} in the same component.

This multi-set contains the extra blocking that a task may suffer due to self-blocking by lower priority tasks:

$$I^{low}_{si} = \max\{X_{sj\ell} \mid \pi_{si} > \pi_{sj} \wedge rc_{s\ell} \geq \pi_{si}\}. \tag{8}$$

In addition, the multi-set contains the self-blocking durations of task τ_{si} itself and the interference caused by self-blocking of higher priority tasks, so that we can define the multi-set $G_{si}(t)$ as follows:

$$G_{si}(t) = \{I^{low}_{si}\} \cup \left(\bigcup_{(1 \leq j \leq i)} \bigcup_{\left(1 \leq k \leq \left\lceil \frac{t}{T_{sj}} \right\rceil\right)} \bigcup_{(R_\ell \in \mathcal{R}_s)} \bigcup_{(1 \leq a \leq m_{sj\ell})} \{X_{sj\ell}\} \right). \tag{9}$$

The term $\bigcup_{(j \leq i)}$ iterates over all tasks τ_{sj} with a higher priority than task τ_{si} and includes the self-blocking by task τ_{si} itself when $i = j$; the term $\bigcup_{\left(1 \leq k \leq \left\lceil \frac{t}{T_{sj}} \right\rceil\right)}$ considers all the k activations of task τ_{sj} in an interval of length t; the term $\bigcup_{(R_\ell \in \mathcal{R}_s)}$ considers all resources R_ℓ accessed by task τ_{sj} and, finally, the term $\bigcup_{(1 \leq a \leq m_{sj\ell})}$ iterates over *the number of resource accesses* to resource R_ℓ by task τ_{sj}. In other words: during each job-activation a task τ_{sj} may access a shared resource R_ℓ for $m_{sj\ell}$ times and it can self-block at any of these attempts. For each attempt of τ_{si} to access resource R_ℓ, a duplicate of the value $X_{si\ell}$ is included in the multi-set $G_{si}(t)$.

Finally, the multi-set $G^{\texttt{sort}}_{si}(t)$ is obtained by sorting the values in $G_{si}(t)$ in non-increasing order. Equation (7) contributes a number of $z(t)$ largest self-blocking occurrences that a task τ_{si} may experience in an interval of length t, i.e., the first $z(t)$ elements of $G^{\texttt{sort}}_{si}(t)$.

It is important to note that the above analysis bounds the number of self-blocking occurrences in any interval of length t by $z(t)$ based on the minimum ratio between task periods and the component period. After a self-blocking occurrence has caused budget-depletion, tasks with a higher priority than the local resource ceiling ($\pi_{sj} > rc_{s\ell}$) may arrive. Those jobs will be pushed through to the next budget period and those are not always accounted for in the resource holding time.

Example 1: Consider a component C_1 with a local fixed-priority scheduler and with two tasks τ_{11} and τ_{12}. Task τ_{12} accesses a global shared resource R_ℓ and τ_{11} is independent, so that the local resource ceiling $rc_{s\ell} = \pi_{12}$. Now the following scenario can happen:

1) task τ_{12} starts its execution and upon its attempt to access resource R_ℓ, it encounters insufficient remaining budget to fit a processor request of $X_{12\ell}$ time units. Task τ_{12} therefore initiates self-blocking.
2) a high priority task τ_{11} arrives just after budget depletion; Hence, it starts executing as soon as the component's budget is replenished and becomes available.
3) After τ_{11} has finished its execution, the remaining budget is again insufficient to fit $X_{12\ell}$ time units.

The scenario is illustrated in Figure 2 and it shows that if two jobs of a single task with higher priority than any of the local resource ceilings can execute within one budget period, then this requires special attention.

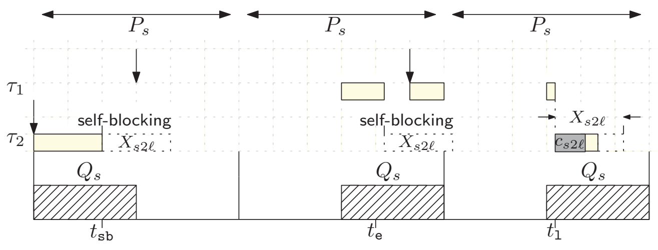

Figure 2. SIRAP disallows that task τ_2 self-blocks two times to prevent budget overruns before access to resource R_ℓ is granted.

In order to accomodate for the extra budget requirement of tasks executing twice within one budget Q_s, it is sufficient that

$$X_s + \sum_{j \; : \; T_{sj} < 2P_s} C_{sj} \quad \leq \quad Q_s. \tag{10}$$

Note that a deadline-constrained task τ_{si} with a period satisfying $2P_s \leq T_{si}$ can execute only once within one budget Q_s. Hence, those tasks are by definition taken into account in the term X_s (see Lemma 1). For simplicity, we have therefore assumed $2P_s \leq T_{si}$ in our system model, so that the summation in (10) returns zero.

Behnam et al. (2009b) have traded off the amount of compensating budget in (10) versus the amount of worst-case local self-blocking by refining SIRAP with an additional local resource ceiling to regulate preemptions during self-blocking. Without loss of generality and for simplicity of the analysis[4], in this paper

4. It has been shown by Shin and Lee (2008) that the assumption $2P_s \leq T_s^{\mathtt{min}}$ allocates an efficient budget for a periodic resource model. Moreover, for relatively small budget periods compared to task periods, the bounded-delay approximation of the periodic resource model is tighter (see Lipari and Bini, 2005).

we adopt the assumption by Behnam et al. (2007, 2009a, 2010a) to constrain resource periods to $2P_s \leq T_s^{\mathtt{min}}$, so that (10) simplifies to $X_s \leq Q_s$.

Contrary to SIRAP, BROE - which also uses a self-blocking mechanism - does not require resource access in the next budget replenishment after the first self-blocking occurrence.

4.3 BROE: component-level self-blocking

Bertogna et al. (2009b) have proposed an alternative method of self-blocking compared to SIRAP - called BROE - which uses EDF scheduling of tasks and components. An analysis for BROE under task-level FPPS is presented by Behnam et al. (2010b). Although the given analyses are opaque, BROE is restricted to global EDF scheduling of components and the bounded-delay model of Feng and Mok (2002).

Contrary to the other protocols, BROE's resource-sharing overhead is left implicit in its local analysis, because the bounded-delay model estimates the worst-case processor supply to a component sufficiently pessimistic. It is therefore unnecessary to account for the largest overrun of each task (as with HSRP) and BROE also refrains from idling the processor to prevent budget overruns (as with SIRAP). A comparison of different synchronization protocols is therefore dependent on the underlying resource-supply model.

BROE uses a hard constant bandwidth server (H-CBS) by Abeni et al. (2009) to provide its allocated processor bandwidth to a component C_s. Apart from period P_s and maximum budget Q_s, defining its utilization $U_s = \frac{Q_s}{P_s}$, at each time t a H-CBS is characterized by an absolute server deadline $d_s(t)$ and a remaining budget $Q_s^{\mathtt{rem}}(t)$. Like with other servers, all pending jobs are contending for processor resources at the server's deadline $d_s(t)$ and whenever a job executes, the budget $Q_s^{\mathtt{rem}}(t)$ is decreased by the received execution time of that job. The rules (1-5) of a BROE server, with respect to the current time t, are as follows (Bertogna et al., 2009b):

1) Initially, $Q_s^{\mathtt{rem}}(0) = 0$ and $d_s(0) = 0$.

2) When a new job of a task τ_i arrives at time t, if the server is idle and if $Q_s^{\mathtt{rem}}(t) \geq (d_s(t) - t)U_s$, then the server budget is replenished to the maximum value Q_s and the server deadline is set to $d_s(t) \leftarrow t + P_s$.

3) Let $t_r = d_s(t) - \frac{1}{U_s} Q_s^{\mathtt{rem}}(t)$. When a new job of a task τ_i arrives at time t, if the server is idle and if $t < t_r$, then the server budget is suspended until time t_r. At time t_r the server budget is replenished to the maximum value Q_s and the server deadline is set to $d_s(t) \leftarrow t_r + P_s$.

4) When $Q_s^{\mathtt{rem}}(t) = 0$, the server is suspended until time $d_s(t)$, so that pending jobs cannot contend for processor resources during time interval $[t,\ d_s(t)]$. At time $d_s(t)$, the server budget is replenished to the maximum value Q_s and the server deadline is set to $d_s(d_s(t)) \leftarrow d_s(t) + P_s$.

5) Whenever a pending task wishes to access a global resource R_ℓ at a time t, it must perform a budget check. I.e., if the remaining budget $Q_s^{\mathtt{rem}}(t) \geq X_{s\ell}$, then there is enough budget to complete the resource access prior to server deadline $d_s(t)$. Then, the task is granted access to resource R_ℓ. Otherwise, the server will replenish its budget no later than time $t_r \leftarrow d_s(t) - \frac{1}{U_s} Q_s^{\mathtt{rem}}(t)$. If $t_r \leq t$, this results in an immediate replenishment of the server budget to the maximum value Q_s and the server deadline is set to $d_s(t) \leftarrow t_r + P_s$. If $t_r > t$, the server is suspended until time t_r. Next, at time t_r the server budget is replenished to the maximum value Q_s and the server deadline is set to $d_s(t) \leftarrow t_r + P_s$.

Rule 1, 2 and 4 describe a H-CBS, see Abeni et al. (2009) and Kumar et al. (2011). BROE adds Rule 3 to the H-CBS to guarantee a fully replenished budget when the server continues after a duration of idle time. Rule 2 and Rule 3 are mutually exclusive. Rule 2 applies when the remaining budget $Q_s^{\mathtt{rem}}(t)$ until deadline $d_s(t)$ would require that the server supplies more processor resources in the interval until deadline $d_s(t)$ than the server utilization U_s. Otherwise, Rule 3 applies, i.e., the supply by the server is still running ahead with respect to its guaranteed processor utilization. Rule 5 adds resource arbitration to the modified H-CBS. For any continuously backlogged interval of length t, i.e., the BROE server has pending jobs, BROE behaves as a conventional H-CBS extended with resource arbitration (Rule 5). A request to access a global shared resource only causes a server self-suspension if there is insufficient budget to complete the critical section and if - similar to Rule 3 - the supplied budget by the server is running ahead with respect to its guaranteed processor utilization.

5 Compositional two-level analysis

In this section we recapitulate the existing compositional analysis for hierarchically scheduled systems in the presence of global shared resources. Firstly, we consider the global integration test which implements the admission control for components based on the resource requirements specified in their interfaces.

Secondly, we consider the local schedulability analysis of a component. Using this analysis an interface of a component is derived. The main contribution of this section is that we define and discern opaque and non-opaque local analysis of components.

5.1 Global schedulability analysis

Since the global scheduler is responsible for arbitrating access to global resources, the global analysis explicitly takes into account the corresponding penalties for global resource sharing which depend on the applied synchronization protocol. These penalties include (*i*) blocking between components and (*ii*) protocol specific penalties for BROE, ONP, OWP or SIRAP. Dependent on the chosen global synchronization protocol, the latter influences the processor requests by a component or it influences the processor supplies to a component. To analyse these scheduling penalties appropriately, we assume that *during component-integration time* the synchronization protocol is known.

The following sufficient schedulability condition holds for top-level EDF-scheduled systems (Baruah, 2006):

$$\forall t > 0 \quad : \quad B(t) + \mathtt{DBF}(t) \leq t. \tag{11}$$

The blocking term, $B(t)$, is defined as Baruah (2006):

$$B(t) = \max\{X_{u\ell} \mid \exists s : R_\ell \in \mathcal{R}_u \cap \mathcal{R}_s \wedge P_s \leq t < P_u\}. \tag{12}$$

The demand bound function $\mathtt{DBF}(t)$ computes the total processor demand of all components in the system for every time interval of length t, i.e.,

$$\mathtt{DBF}(t) \quad = \sum_{1 \leq s \leq N} \left(O_s(t) + \left\lfloor \frac{t}{P_s} \right\rfloor Q_s \right), \tag{13}$$

where $O_s(t)$ defines the additional amount of budget that a component C_s requires under a certain global synchronization protocol in order to prevent excessive blocking durations for other components in the system.

With ONP a component can request for an additional amount of X_s time units of processor time each period P_s. Hence, the term $O_s(t)$ is defined by Behnam et al. (2010c):

$$O_s(t) \quad = \quad \left\lfloor \frac{t}{P_s} \right\rfloor X_s. \tag{14}$$

With OWP a component can only request an additional amount of X_s time units of processor time once. Hence, the term $O_s(t)$ is defined by Behnam et al. (2010c):

$$O_s(t) \quad = \quad \begin{cases} X_s & \text{if } t \geq P_s \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

For both SIRAP and BROE, it is required that $X_s \leq Q_S$ in order to be able to complete an entire critical section within a single budget of size Q_s. Hence, we increase Q_s with $O_s(t)$ time units if it is too small to fit X_s time units contiguously, where $O_s(t)$ is defined as follows:

$$O_s(t) \quad = \quad \left\lfloor \frac{t}{P_s} \right\rfloor \max\left(0, X_s - Q_s\right). \tag{16}$$

A global admission of EDF-scheduled components with the $\mathtt{DBF}(t)$-based analysis by Baruah (2006) has not been proven for BROE. Instead, Bertogna et al. (2009b) have modified the analysis by Baruah (2006) in order to make it applicable to BROE and they derived a sufficient utilization-based test, i.e.,

$$\forall w \ : \ 1 \le w \le N \ : \ \frac{B(P_w)}{P_w} + \sum_{1 \le s \le w} \frac{Q_s + O_s(P_s)}{P_s} \le 1. \tag{17}$$

The test in (17) is also sufficient for SIRAP, ONP and OWP. Although this utilization-based test is less tight than the demand-bound test, its linear time complexity in the number of components can be interesting for open systems where the admission test has to be executed online.

For global FPPS of components - by definition disallowing BROE - the following sufficient schedulability condition holds (Lehoczky et al., 1989):

$$\forall 1 \le s \le N : \ \exists t \in (0, P_s] : \ \texttt{RBF}(t, s) \le t. \tag{18}$$

The $\texttt{RBF}(t, s)$ denotes the worst-case cumulative processor request of C_s and all higher priority components for a time interval of length t, i.e.,

$$\texttt{RBF}(t, s) = \ B_s + \sum_{1 \le r \le s} \left(O_r(t) + \left\lceil \frac{t}{P_r} \right\rceil Q_r \right). \tag{19}$$

where $O_r(t)$ defines the additional amount of budget that a component C_s requires under a certain global synchronization protocol in order to prevent excessive blocking durations for other components in the system.

Similar to EDF, under global FPPS and ONP the term $O_r(t)$ is defined by Behnam et al. (2010c):

$$O_r(t) \ = \ \left\lceil \frac{t}{P_r} \right\rceil X_r. \tag{20}$$

Also under FPPS, a component arbitrated by OWP can only request an additional amount of X_s time units of processor time once. Hence, the term $O_r(t)$ becomes time independent and it is defined by Behnam et al. (2010c):

$$O_r(t) \ = \ X_r. \tag{21}$$

Taking into account the requirement of SIRAP that $X_r \le Q_r$, we define the term $O_r(t)$ in order to enforce that Q_r is artificially increased if it is too small to fit X_r time units contiguously within one period P_r, i.e., as follows:

$$O_r(t) \ = \ \left\lceil \frac{t}{P_r} \right\rceil \max\left(0, X_r - Q_r\right). \tag{22}$$

The blocking term, B_s, is defined according to Baker (1991):

$$B_s = \max\{X_{u\ell} \mid \Pi_u < \Pi_s \le RC_\ell\}. \tag{23}$$

5.2 Local schedulability analysis

After developing a component and before publishing it to a framework integrator, a component is packaged as a re-usable entity. This includes deriving a timing interface to abstract from internal deadline constraints of tasks. Such an abstraction requires an explicit choice for a resource-supply model, capturing the processor supply to a component. Moreover, a component specifies what it needs in terms of resources and exposes those resources that may be shared globally in its interface. Whether or not a global resource is actually used by other components is unknown within the context of a component.

There are several ways to account for local scheduling penalties due to global resource sharing. One might assume that each resource is global and, subsequently, account for the worst-case overhead inside the local analysis, e.g., SIRAP's analysis by Behnam et al. (2007, 2010a) and OWP's analysis by Davis and Burns (2006) and by Behnam et al. (2010c). Alternatively, one may assume that all resources are local during the local analysis and compensate for sharing between components at integration time, e.g., ONP's analysis by Behnam et al. (2010c).

The latter alternative presents the same view as during component development, i.e., a component has the entire platform at its disposal and all resources. Whenever a synchronization protocol for global resources is used that is compliant with a synchronization protocol for local resources, the local analysis of a component can be based on local properties only. We call such a local analysis *opaque*, because it separates local and global resource arbitration.

Definition 1: An opaque analysis provides a sufficient local schedulability condition for an individual component. It treats all resources accessed by the component as local, so that, even under global sharing, properties of the global synchronization protocol do not influence the computed interface parameters.

The key consequence of an opaque local analysis is the absence of resource holding times in the equations that validate local schedulability. Section 3.4 already showed how resource holding times can be computed without making assumptions on the global synchronization protocol. The opaque analyses presented in this section accomplish that parameter Q_s in the interface of the component can also be derived regardless of the global synchronization protocol. Table 2 gives an overview of local analyses found in literature by indicating their opacity. The remainder of this section distinguishes opaque and non-opaque local schedulability analyses under various global synchronization protocols.

5.2.1 Opaque local analysis

Traditional protocols such as the PCP by Sha et al. (1990) and the SRP by Baker (1991) can be used for *local* resource sharing in HSFs (see Almeida and

Peidreiras, 2004). With an opaque local analysis, we can re-use the same local analysis in the presence of global shared resources. The local analysis of ONP by Behnam et al. (2010c) satisfies the notion of opacity, because it uses a simple overrun upon integration and nothing else locally.

By filling in task characteristics in the demand bound $\texttt{DBF}$ of (11) or the request bound $\texttt{RBF}$ of (18) and replacing their right-hand sides by (1), i.e., replace t by $\texttt{sbf}_{\Gamma_s}(t)$, the same schedulability analysis holds for tasks executing within a component as for components at the global level.

We first perform these steps for local EDF of tasks for which the following sufficient schedulability condition holds:

$$\forall t \ : \ t \geq 0 \ : \ \texttt{dbf}_s(t) \leq \texttt{sbf}_{\Gamma_s}(t), \tag{24}$$

where $\texttt{dbf}_s(t)$ denotes the cumulative processor demand of all tasks of component C_s for a time interval of length t.

For BROE, ONP and our new OWP analysis, the $\texttt{dbf}_s(t)$ is fully compliant to the schedulability analysis for task sets on a dedicated unit-speed processor, i.e.,

$$\texttt{dbf}_s(t) \quad = \quad b(t) + \sum_{1 \leq i \leq n_s} \left\lfloor \frac{t - D_{si} + T_{si}}{T_{si}} \right\rfloor E_{si}. \tag{25}$$

The blocking term, b_{si}, is defined according to Baruah (2006):

$$b(t) = \max\{h_{sj\ell} \mid \exists k : h_{sk\ell} > 0 \wedge D_{sk} \leq t < D_{sj}\}. \tag{26}$$

Secondly, we perform the same steps for local FPPS of tasks for which the following sufficient schedulability condition holds:

$$\forall 1 \leq i \leq n_s : \exists t \in (0, D_{si}] : \ \ \texttt{rbf}_s(t, i) \leq \texttt{sbf}_{\Gamma_s}(t), \tag{27}$$

where $\texttt{rbf}_s(t, i)$ denotes the worst-case cumulative processor request of τ_{si} for a time interval of length t.

Table 2
Overview of the synchronization protocol's support for integrating resource-sharing components into the HSF with opaque analysis.

Analysis of resource-sharing strategies	**Authors**	**Opacity**
BROE	Bertogna et al. (2009b)	yes
HSRP - overrun without payback (ONP)	Davis and Burns (2006)	no
HSRP - overrun without payback (ONP)	Behnam et al. (2010c)	yes
Enhanced overrun	Behnam et al. (2010c)	no
Improved overrun without payback (IONP)	Behnam et al. (2011)	no
HSRP - overrun with payback (OWP)	Davis and Burns (2006)	no
HSRP - overrun with payback (OWP)	Behnam et al. (2010c)	no
SIRAP	Behnam et al. (2007, 2010a)	no

For BROE, ONP and our new OWP analysis, the $\mathtt{rbf}_s(t,i)$ is fully compliant to the schedulability analysis for task sets on a dedicated unit-speed processor, i.e.,

$$\mathtt{rbf}_s(t,i) \quad = \quad b_{si} + \sum_{1\le j\le i} \left\lceil \frac{t}{T_{sj}} \right\rceil E_{sj}. \tag{28}$$

The blocking term, b_{si}, is defined according to Baker (1991):

$$b_{si} = \max\{h_{sj\ell} \mid \pi_{sj} < \pi_{si} \le rc_{s\ell}\}. \tag{29}$$

In Section 8 we shall show that BROE's analysis requires to replace the $\mathtt{sbf}_{\Gamma_s}(t)$ with $\mathtt{lsbf}_{\Gamma_s}(t)$ in the local schedulability conditions of (24) and (27). Contrary to the other protocols, BROE explicitly assumes the bounded-delay resource-supply model by Feng and Mok (2002). Since BROE's underlying resource-supply model captures the processor supply to a component sufficiently pessimistic, all resources can be treated as local in the local analysis. This makes BROE's local analysis opaque.

5.2.2 Non-opaque local analysis

The local analysis of a component under resource arbitration by OWP or by SIRAP does not regard global resources as local, thereby violating our definition of opacity. Since global resources may need to be shared with tasks in other components, the idea of SIRAP is to use the resource holding times of local tasks to tighten the analysis of wasted resources. OWP penalizes a component for (potentially) forcing changes in the processor supply $\mathtt{sbf}_{\Gamma_s}(t)$ due to accesses to global resources. The properties of SIRAP and OWP are therefore reflected on the computed value of budget Q_s of a component.

A component using SIRAP demands more resources in its worst-case scenario than an independent component demands. Using the analysis presented by Behnam et al. (2010a), we therefore need to add a term, $I_{si}(t)$, to account for *self-blocking* to the $\mathtt{rbf}_s(t,i)$ (see Section 4.2). The self-blocking term I_{si} of a task τ_{si} is defined in terms of $z(t) = \left\lceil \frac{t}{P_s} \right\rceil$, representing an upper bound to the number of self-blocking occurrences within a time interval of length t, and a multi-set $G_{si}^{\mathtt{sort}}(t)$ which comprises all self-blocking lengths $X_{si\ell}$ that a task τ_{si} may experience by itself and other tasks τ_{sj} in the same component in a non-decreasing order. The sufficient FPPS condition for a task set $\mathcal{T}_s$ on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ is given by Behnam et al. (2010a):

$$\forall \tau_i \in \mathcal{T}_s : \exists t \in (0, D_{si}] : \mathtt{rbf}_s(t,i) + I_{si}(t) \le \mathtt{sbf}_{\Gamma_s}(t), \tag{30}$$

where $\mathtt{rbf}_s(t,i)$ is defined in (28).

For local EDF we need to add the self-blocking occurrences of all tasks in the component rather than the FPPS solution which considers the candidates for self-blocking of each task separately, i.e., the set of self-blocking occurrences under local EDF can be represented by $I_{sn_s}(t)$ and can be derived from $G^{\mathtt{sort}}_{sn_s}(t)$. We recall that $G^{\mathtt{sort}}_{si}(t)$ stores all values $X_{si\ell}$ in a non-decreasing order and includes a value for each individual resource access by a job of task τ_{si} to resource R_ℓ. Section 4.2 showed how to construct such a multi-set. The sufficient EDF condition for a task set $\mathcal{T}_s$ on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ is now given by:

$$\forall t \ : \ \ t \geq 0 \ : \ \ \mathtt{dbf}_s(t) + I_{sn_s}(t) \leq \mathtt{sbf}_{\Gamma_s}(t), \tag{31}$$

where $\mathtt{dbf}_s(t)$ is defined in (25).

To demonstrate the effect of using properties of the global synchronization protocol for optimizing the parameters of the component's interface in the local analysis, we consider a simpler non-opaque analysis than SIRAP. Behnam et al. (2011) improved ONP by observing that the normal budget Q_s of a component C_s has to be served at least before relative deadline $P_s - X_s$, resulting in a processor requirement which can be met by the EDP model $\Omega_s = (P_s, Q_s, P_s - X_s, \mathcal{X}_s)$. This improved ONP (IONP) analysis is non-opaque, because it uses resource holding times to tighten the deadline $D_s = P_s - X_s$ in order to obtain a better budget parameter, Q_s (thereby violating Definition 1).

Example 2: Consider a component C_2 with a period $P_2 = 10$ and a single task $\tau_{21} = (27, 5, 27, \{0.5\})$ which specifies an access to a global resource R_ℓ for a duration of $h_{21\ell} = X_{21\ell} = X_2 = 0.5$ time units. We use ONP for arbitrating access to global resources. By making assumptions in the local analysis on how ONP changes the processor supply to a component, a manufacturer may give an untruthful impression on the efficiency of the component.

According to the IONP analysis where the resource holding time of 0.5 time units is exploited to tighten the deadline for budget Q_2, it is sufficient to allocate $Q_2 = 2.5$ time units every period of 10 time units. This budget allocation is derived from the EDP resource $\Omega_2 = (P_2,\ Q_2,\ P_2 - X_2,\ \mathcal{X}_2) = (10,\ 2.5,\ 9.5,\ \{0.5\})$ and it is computed based on the assumption that an additional amount of 0.5 time units may need to be supplied within one component period to complete resource access by means of a budget overrun.

If resource R_ℓ turns out to be local to component C_2, i.e., component C_2 is entirely independent of other components in the system, then budget overruns are unnecessary for accessing resource R_ℓ. An independent component C_2 would have required a periodic budget of $Q_2 = \frac{8}{3}$ time units every period of 10 time units.

If a component developer would have provided the interface computed by means of the non-opaque IONP analysis intentionally, then the component may look more attractive, because it requires an optimal processor bandwidth - meaning that reducing the deadline D_2 further than 9.5 does not yield lower bandwidth requirements - of only 2.5 time units instead of the larger quantum of $\frac{8}{3}$ time units every period of 10 time units. We recall, however, the 2.5 time units must be supplied within 9.5 time units from the budget's release, leading to a density of processor allocations of $\frac{2.5}{9.5}$. This density under the tightened EDP constraint is higher than the one without resource sharing satisfying the periodic resource model $\Gamma_2 = (10, \frac{8}{3})$, i.e., $\frac{8/3}{10} < \frac{2.5}{9.5}$. In other words, at the level of composition one may admit a component into the system requiring $\frac{8}{3}$ time units every 10 time units while one may need to reject a component requiring 2.5 time units before relative deadline 9.5 every 10 time units. Hence, a non-opaque analysis may give the illusion of resource efficiency by artificially creating resource dependencies.

The above example clearly shows the importance of opaque analyses for open environments. The only protocols satisfying these properties are (the basic version of) ONP and BROE. Surprisingly and contrary to both ONP, the current local OWP analysis is non-opaque, but for OWP the violations of the opacity property do not lead to an improved analysis for components (as it does with IONP), irrespective of whether or not global resources are actually shared or not.

According to Behnam et al. (2009a, 2010c), OWP has additional pessimism at the local scheduling level compared to overrun without payback (ONP). They have therefore modified the $\texttt{sbf}(t)$ compared to the definition given in (1), see Behnam et al. (2010c). Firstly, due to payback a component may supply less resource within a component period. Secondly, the payback increases the blackout duration of a component. The current local OWP analysis is non-opaque, because it needs to know which resource are shared globally in order to modify the $\texttt{sbf}(t)$ accordingly. OWP is therefore hardly beneficial in its current form compared to ONP. Should overrun with payback therefore be considered obsolete based on these observations, or not?

6 SRP WITH BUDGET OVERRUNS: TO PAYBACK OR NOT TO PAYBACK?

We reconsider the problem of resource sharing across budgets. Ghazalie and Baker (1995) recognized that when tasks access resources across their budget with the SRP, their budget may deplete during resource access so that other components may experience an excessive blocking duration. As a solution, they proposed to overrun the budget Q_s until the critical section completes and they subsequently deduct the amount of overrun from the next budget replenishment of the corresponding component. Their (global) analysis corresponds to the analysis by Davis and Burns (2006) and Behnam et al. (2010c) in the sense that we need to account for additional interference to all other components due to a worst-case over-provisioning of X_s budget which facilitates the overrun. This results in the sufficient schedulability condition under global EDF and FPPS of components as defined in (11) and (18).

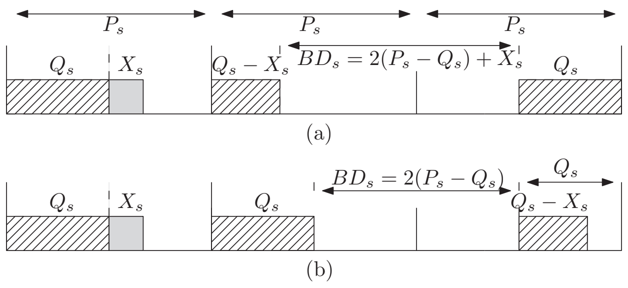

Figure 3. Worst-case characterization of the periodic processor supply for the SRP with mechanisms for overrun and payback, as presented in Behnam et al. (2010c).

We now need to characterize the worst-case resource supply to the tasks serviced by component C_s. Behnam et al. (2010c) distinguish two cases to represent the worst-case processor supply, see Figure 3. The worst-case scenario happens after the first budget supply of Q_s has overrun with an amount of X_s. This leads to a payback in one of the subsequent component periods. A payback in the second period, as shown in Figure 3(a), means that (*i*) the amount of overrun X_s is deducted from the next replenishment of Q_s; and (*ii*) the next replenishment of Q_s is serviced as late as possible before the deadline P_s. The longest blackout of the processor supply is $BD_s = 2(P_s - Q_s) + X_s$.

Alternatively, the component may overrun its budget again in the second period, see Figure 3(b), so that a payback happens in the third period. The budget in the third period is again supplied as late as possible, taking into account that there must be enough time until the deadline to accommodate for another overrun. This scenario has a smaller worst-case processor blackout of $BD_s = 2(P_s - Q_s)$.

Since component deadlines are assumed to be equal to their period P_s, it is sufficient to consider the response time of the first activation of each component, see (19). Furthermore, the schedulability test in (18) guarantees that an amount of $Q_s + X_s$ budget can be provisioned within a period P_s. As a consequence, the latest start time of that budget provisioning is $P_s - (Q_s + X_s)$. This is independent of whether or not an overrun has taken place, as shown in Figure 4.

We can now derive the following lemma:

Lemma 2: A component C_s following the periodic resource model $\Gamma_s =$

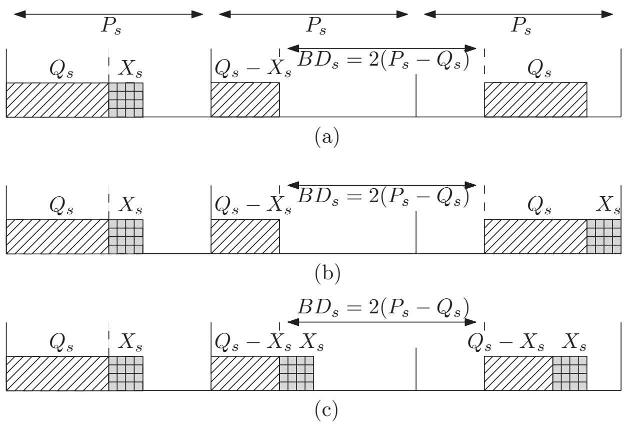

Figure 4. The latest starting time of the processor supply in each period is independent of whether or not an overrun takes place in that period.

$(P_s, Q_s, \mathcal{X}_s)$, arbitrating global shared resource using the OWP mechanism, cannot experience more than the regular blackout duration of $BD_s = 2(P_s - Q_s)$.

Proof: Following the periodic resource model Shin and Lee (2008), shown in Figure 4, the latest time that a budget of at least $Q_s - X_s$ will be provisioned is at time $P_s - (Q_s + X_s)$, because there must be sufficient time between the finishing time of the normal budget Q_s and the period boundary P_s to accommodate for an overrun situation. Hence, the $P_s - X_s$ is an implicit deadline for the normal budget Q_s, so that the blackout for two consecutive budget supplies is at most $BD_s = 2(P_s - Q_s)$. □

Contrary to ONP, we cannot make the implicit deadline $P_s - X_s$ of budget Q_s explicit for OWP by applying the EDP model by Easwaran et al. (2007), because this would further reduce the blackout duration to $BD_s = 2(P_s - Q_s) - X_s$, see Figure 5. Although this is obviously optimistic for OWP, this explicit deadline improves the local analysis of ONP (see Behnam et al., 2011). This improved ONP (IONP) analysis is non-opaque, because it uses resource holding times to tighten the computed budget parameter of a component.

The result of Lemma 2 is the same as the analysis derived by Davis and Burns (2006), although they do not support a compositional analysis. Behnam et al. (2010c) came up with an improved OWP method - called *enhanced overrun* - to improve the blackout duration assumed by their initial OWP analysis, see Figure 6. They improve their analysis by postponing the next replenishment of a component, i.e., contrary to Lemma 2 they postpone the start time of the budget provisioning. However, their alternative (*i*) requires modifications in the implementation of the overrun mechanism, since it alters the periodicity of

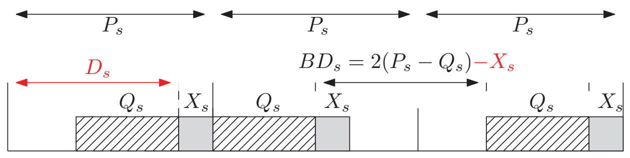

Figure 5. Since budget Q_s must be provisioned before deadline $P_s - X_s$, the EDP resource model Easwaran et al. (2007) enables a tighter, non-opaque ONP analysis.

budget releases and (*ii*) still assumes a pessimistic budget supply of at most $Q_s - X_s$ in an interval of length $2P_s$.

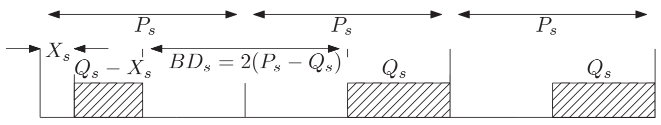

Figure 6. In Behnam et al. (2010c) the extra blackout due to payback is reduced by introducing a flexible release off set for budget $Q_s - X_s$, i.e., illustrated by the initial delay of X_s.

The latter source of pessimism is inherited from the analysis by Davis and Burns (2006), which considers the effect of *push-through blocking* due to an overrun with payback. This effect is shown in Figure 4(c), where a task arrives just after depletion of budget $Q_s - X_s$. Although the task is *pushed through* to the next budget replenishment, the blackout duration of the processor supply remains $BD_s = 2(P_s - Q_s)$. Using the periodic resource model by Shin and Lee (2008), however, we already assume an initial delay of BD_s followed by a periodic supply of a budget of size Q_s.

We also recall that the overrun budget X_s is merely for global reasons, because the task set does not need an extra budget of X_s, i.e., it is already feasible with a budget of Q_s every period P_s. The remaining question is: given that a fixed-priority-scheduled task set using a plain SRP-based resource arbitration is schedulable on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$, is there any task that may experience insufficient budget after a payback of at most X_s budget?

The analysis by Behnam et al. (2010c) is based on the point of view that the minimum resource supply in an interval of length P_s must be assumed to be equal to $Q_s - X_s$, as is suggested by Figure 3. We will show that the model in Behnam et al. (2010c) is indeed overly complex and pessimistic. The main reasoning behind this claim is that the task set as a whole actually receives Q_s budget in an interval of length P_s, but the *individual resource supply to a task activation* has changed. An overrun advances exactly the amount of budget of at most X_s to complete the critical section. The task activations that have consumed this overrun cannot claim again processor time in the next budget supply, so that a potential subsequent overrun cannot be caused by them. The overrun budget in Figure 4 is grid-marked to indicate its partial availability.

Lemma 3: Given that a task set $\mathcal{T}_s$ under SRP-based resource arbitration is schedulable on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$, a task $\tau_{si} \in \mathcal{T}_s$ cannot miss its deadline when adding an overrun with payback mechanism.

Proof: We only need to consider the case where an overrun situation has taken place subsequently causing a payback at the next budget replenishment. Otherwise, the resource supply is unchanged compared to the $\mathtt{sbf}_{\Gamma_s}$ for independent components, see (1).

We observe that an overrun situation can only be caused by a resource lock by any of the tasks $\tau_{si} \in \mathcal{T}_s$. Assume that task τ_{si} locks resource R_ℓ, so that the component ceiling is at least equal to the resource ceiling $rc_{s\ell}$. Furthermore, budget Q_s depletes during resource access. This means that component C_s may overrun its normal budget Q_s for at most an amount of $X_{s\ell}$ processor time, which allows completing the critical section initiated by task τ_{si}.

We prove by contradiction that no task $\tau_{sj} \in \mathcal{T}_s$ will miss a deadline due to the payback of $X_{s\ell}$ budget at the next replenishment of the normal budget Q_s, i.e., assume that there exists a task $\tau_{sj} \in \mathcal{T}_s$ that will miss a deadline after an overrun.

We tackle this proof obligation by distinguishing four cases: tasks that may preempt the critical section ($\pi_{sj} > rc_{s\ell}$), tasks that are blocked during the critical section ($rc_{s\ell} \geq \pi_{sj} > \pi_{si}$), the resource-locking task τ_{si} itself ($\pi_{si} = \pi_{sj}$) and tasks that have a lower priority than the resource-locking task ($\pi_{si} > \pi_{sj}$).

1) $\pi_{sj} > rc_{s\ell}$: these tasks may preempt the critical section. Moreover, these tasks contribute to the length of $X_{s\ell}$ for at most a single preemption (Lemma 1). This means that if the task arrives after depletion of Q_s and an overrun takes place, then it will execute in the overrun budget. Contrary to the assumptions in Behnam et al. (2010c), this task *will actually consume* the overrun budget when it is available. An activation of task τ_{sj} which consumes E_{sj} of overrun budget cannot request the same amount of budget in the next budget period P_s, because it has already finished its execution during the overrun. And vice versa: if an activation of task τ_{sj} requests for E_{sj} of normal budget, then it did not execute during a possible overrun in the previous budget period. An overrun in the previous period could therefore have at most a length of $X_{s\ell} - E_{sj}$. If E_{sj} of the overrun has not been consumed, then the next budget supply will also not be reduced with this amount of payback. Thus, the resources requested by the current activation of task τ_{sj}, i.e., E_{sj}, will be available before task τ_{sj} will miss a deadline. Hence, no higher priority task τ_{sj} where $\pi_{sj} > rc_{s\ell}$ will miss a deadline due to a payback.

2) $rc_{s\ell} \geq \pi_{sj} > \pi_{si}$: these tasks are blocked during the critical section by the resource ceiling. When we do not advance the overrun budget $X_{s\ell}$ compared to plain SRP-based resource arbitration, these tasks are schedulable. The reason is that the blocking duration of at most $X_{s\ell}$ is already accounted in the $\texttt{rbf}_s(t,j)$ of task τ_{sj}. A new periodic supply cannot start with local blocking, because blocking should already start in the previous provisioning and use the overrun (if needed). Hence, OWP does not cause a deadline miss for any of the tasks τ_{sj} that are blocked by the resource-accessing task τ_{si}.

3) $\pi_{si} = \pi_{sj}$: for the resource locking task τ_{si} itself the same reasoning holds as for the first case: it either consumes an amount of $h_{si\ell}$ of the overrun budget in the previous budget period or it consumes $h_{si\ell}$ from the normal budget Q_s in the current budget period. Both cases are mutually exclusive and cannot cause a deadline miss.

4) $\pi_{si} > \pi_{sj}$: these tasks have a lower priority than the resource-locking task and have already accounted $X_{s\ell}$ as interference in their $\texttt{rbf}_s(t,j)$. Hence, similarly to case 3, these tasks cannot assume that any budget would be immediately available after replenishment of Q_s in case of plain SRP-arbitration. The OWP mechanism does therefore not cause a deadline miss to any task τ_{sj} where $\pi_{si} > \pi_{sj}$.

By contradiction we have proven that advancing the resource supply of X_s due to overrun with payback does not hamper the schedulability of task set $\mathcal{T}_s$ compared to plain SRP-based resource arbitration. $\square$

From both Lemma 2 and Lemma 3 we directly obtain the following results:

Theorem 1: The local schedulability analysis in (27) for a task-set $\mathcal{T}_s$ on an SRP+fixed-priority-scheduled periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ can be applied when arbitrating global shared resources using overrun with payback (OWP).

Theorem 2: The local schedulability analysis in (24) for a task-set $\mathcal{T}_s$ on an SRP+EDF-scheduled periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ can be applied when arbitrating global shared resources using overrun with payback (OWP).

We believe these theorems yield an interesting result, because they show that the local schedulablity analysis of overrun with and without payback are exactly the same. In particular, we can reuse the sufficient schedulability condition for ONP as presented in (27) and (24).

Finally, we answer the main question of this section: to payback or not to payback? The global schedulability analysis for components arbitrated by overrun with payback is unchanged and was already considerably better than the global analysis of overrun without payback. In addition, we have improved the local schedulability analysis, such that there is no difference between ONP and OWP. Hence, there is no reason to deploy overrun without a payback mechanism from an opacity perspective.

7 Opaque local analysis for SIRAP

In this section we derive an opaque local analysis for SIRAP. With the non-opaque SIRAP analysis by Behnam et al. (2010a), one must know the number of accesses to any global resource by each individual job. Although this is unnecessary for HSRP and BROE, it makes SIRAP superior to ONP in case each of those resources are actually shared with at least one other component.

HSRP accounts for a worst-case overrun in each component period, while an actual overrun does not necessarily happen each period. However, exposing a multi-set of resource holding times to the global schedulability test (similar to SIRAP) is impossible for HSRP, because this breaks the independent analysis of components due to the dependency of $G_{si}^{\mathtt{sort}}(t)$ on the time values t in the testing set of the tasks in $\mathcal{T}_s$.

Since each element in the set $G_{si}^{\mathtt{sort}}(t)$ is at most of length X_s, ONP only performs equally well when a self-blocking of approximately X_s is deducted in each component period. SIRAP is therefore always superior to ONP, so that the ONP analysis can be safely used to implement a SIRAP system.

Theorem 3: If a task set $\mathcal{T}_s$ is deemed schedulable on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ using the ONP analysis, then it is also feasible on a periodic resource $\Gamma'_s = (P_s, Q_s + X_s, \mathcal{X}_s)$ using a SIRAP implementation.

Proof: The sufficient FPPS condition for a task set $\mathcal{T}_s$ on a periodic resource $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$ is given by (30):

$$\forall \tau_i \in \mathcal{T}_s : \exists t \in (0, D_{si}] : \mathtt{rbf}_s(t,i) + I_{si}(t) \leq \mathtt{sbf}_{\Gamma_s}(t), \tag{32}$$

where $\mathtt{rbf}_s(t,i)$ is defined in (28), $\mathtt{sbf}_{\Gamma_s}(t)$ is defined in (1) and the exact construction of $I_{si}(t)$ is given in (7), see Section 4.2. By definition it holds that $\forall e \in G_{si}^{\mathtt{sort}}(t) : e \leq X_s$. Hence, the schedulability condition in (32) is implied by:

$$\forall \tau_i \in \mathcal{T}_s : \exists t \in (0, D_{si}] : \quad \mathtt{rbf}_s(t,i) + \left\lceil \frac{t}{P_s} \right\rceil X_s \leq \mathtt{sbf}_{\Gamma_s}(t). \tag{33}$$

Since within one budget period a self-blocking occurrence can only happen at the end of a supply due to insufficient budget to complete a critical section, we can remove the dependency on t provided that we add X_s extra budget in each component period. In other words, for a given P_s the smallest conservative budget Q'_s can be derived by:

$$X_s + \left(\min Q_s : \left(\forall \tau_i \in \mathcal{T}_s : \exists t \in (0, D_{si}] : \mathtt{rbf}_s(t,i) \leq \mathtt{sbf}_{\Gamma_s}(t)\right)\right). \tag{34}$$

The right-hand term of (34) is the same as schedulability condition for ONP, see (27), which concludes our proof for local FPPS. Since our proof is based on the observation that every server period P_s an at most e idle time is inserted ($\forall e \in G_{si}^{\mathtt{sort}}(t) : e \leq X_s$), the same steps as above can be straightforwardly applied to the $\mathtt{dbf}_s(t)$-based test in (31) for local EDF of tasks, which concludes our proof. □

Given Theorem 3, we make it possible to integrate a component validated by an opaque analysis for both SRP+FPPS and SRP+EDF into the HSF, while using SIRAP for global resource arbitration.

8 A DESIGN METHODOLOGY

So far in this paper, we have shown that ONP and BROE have opaque analysis and we have developed a new opaque analysis for OWP and SIRAP. In this section, we propose a three-step approach for optimizing the overall resource-requirements of a system at integration time.

The objective of our approach is as follows. We are given the following parameters of a component: (*i*) a task set $\mathcal{T}$ with predefined task characteristics $(T_{si}, E_{si}, D_{si}, \mathcal{H}_{si})$, (*ii*) a local scheduling policy (FPPS or EDF) and (*iii*) the period of the component, P_s. Given these component parameters, we want to compute the remaining two parameters of the component's interface Γ_s, i.e., we compute the smallest budget Q_s that satisfies local deadline constraints of tasks and we compute the set of resource holding times $\mathcal{X}_s$. These parameters must be computed independent of the global synchronization protocol, such that the local (opaque) schedulability analysis is satisfied and the overall impact on the global system load is minimized. In order to reach our objective, we define three steps.

Firstly, one must select a resource-supply model, which may significantly impact the allocated budget to a component - even if a component shares no global resources. In the presence of global shared resources, a synchronization protocol may limit the choice of a resource-supply model. Hence, we consider the problem of making this dependence opaque, so that the choice of a resource-supply model can also be deferred until integration time (just like the synchronization protocol).

Secondly, a local resource ceiling must be selected for each shared resource. Contrary to local shared resources, for global resources an artificial increase of the local resource ceiling compared to (5) may improve schedulability. On the one hand, an explicit assumption on local resource ceilings affects the local analysis non-opaquely, because a component gives up its local view on resource sharing. On the other hand, since opacity allows one to defer the choice of a global synchronization protocol until system integration, binding of synchronization primitives may come with globally selected local ceilings. For example, Davis and Burns (2006) disable all local preemptions during global resource access. We consider the problem of finding the local resource ceilings that allow for a maximum reuse at the level of composition in terms of global schedulability, i.e., we capture each of those component configurations in an interface candidate.

Thirdly, given a set of components that need to be executed on a set of resources and a processor, we search for a method to select one of those interface candidates (i.e., we fix the resource-supply model and we choose the appropriate local resource ceilings) that minimize the system load.

The second and the third step are inspired by the methods by Shin et al. (2008) and Behnam et al. (2010b) of deriving and selecting interface candidates. Their approach cannot be applied directly, because they implicitly assume that all local resources are globally shared between components. If this assumption is violated, then their method may not lead to optimal solutions. Lifting their assumption is difficult, because it increases the design space of a system exponentially. That is, each of the m_s resources accessed by a component can be globally shared (or not) and all 2^{m_s} combination should be considered. Fortunately, an opaque analysis enables us to traverse this exponentially sized design space in $\mathcal{O}(MNn_s^{\mathtt{max}})$ time and space complexity, where $n_s^{\mathtt{max}} = \max\{n_s \mid 1 \leq s \leq N\}$. Compared to Shin et al. (2008) and Behnam et al. (2010b), the complexity of our method is only a factor M higher. By exploiting the key concept of opacity that all resources are assumed to be local in the local analysis, we obtain a tighter integration of resource-sharing components than Shin et al. (2008) and Behnam et al. (2010b) did.

We conclude this section with a case study. It demonstrates our design methodology and it compares our algorithms to the algorithms by Shin et al. (2008) and Behnam et al. (2010b).

8.1 Choosing a resource-supply model

Each of the global synchronization protocols considered in this paper has a period constraint P_s. For any access to a global resource granted by the global SRP, the period P_s serves as a relative deadline for completion of a resource access. For SIRAP and BROE, the period P_s also bounds the waiting time of a task that wishes to access a global resource.

Given a period constraint P_s, the bounded-delay model gives a linear lower bound $\mathtt{lsbf}_{\Gamma_s}(t)$ of the actually supplied resources by a periodic resource $\mathtt{sbf}_{\Gamma_s}(t)$ with the same period parameter (see Lipari and Bini, 2005). For this reason, the schedulability analysis for OWP, ONP and SIRAP using the bounded-delay model is sufficient but pessimistic. BROE must use the bounded-delay model and a period constraint P_s, while it is non-compliant with the periodic resource model by Shin and Lee (2008); this is the main weakness of BROE compared to the other protocols.

Example 3: Consider component C_3 with a period $P_3 = 10$ and two tasks: $\tau_{31} = (1000, 2, 29, \{0.5\})$ and $\tau_{32} = (1000, 1, 1000, \emptyset)$. Task τ_{31} accesses a global resource R_1 for a duration of $h_{311} = X_{31} = 0.5$ time units; task τ_{32} is independent. The smallest budget satisfying the local schedulability condition is the same for both deadline-monontonic scheduling as well as for EDF scheduling of tasks. Using the local schedulability condition in (27), we derive $Q_3 = 1$, yielding an interface $\Gamma_3^{(\texttt{PRM})} = (10, 1, \{0.5\})$. Without any global resource sharing, the required processor bandwidth of component C_3 is therefore 0.1, see Figure 7(a). When arbitrating resource R_1 with BROE, however, a budget of $Q_3 = 1$ is insufficient, see Figure 7(b). According to BROE's bounded-delay criteria, i.e., using (3), component C_3 requires a budget of $Q_3 = 1.63$. The corresponding bounded-delay interface $\Gamma_3^{(\texttt{BDM})} = (10, 1.63, \{0.5\})$ yields a bandwidth of 0.163.

To compare: arbitrating resource R_1 with ONP or OWP would allocate an overrun budget of 0.5 time units, so that the allocated processor bandwidth for C_3 becomes 0.15. In this example, BROE requires more processor bandwidth than ONP, OWP or - by virtue of Theorem 3 - SIRAP.

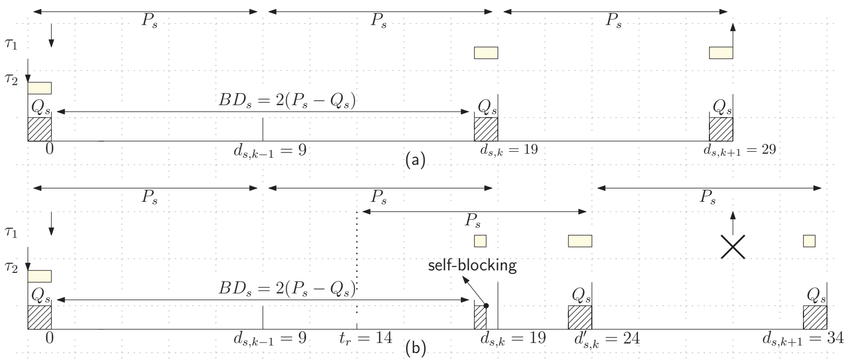

Figure 7. The periodic resource model is inapplicable to BROE.

Although it has been shown by Kumar et al. (2011) that a conventional H-CBS without support for resource sharing complies to the periodic resource model, Example 3 shows BROE's pessimism compared to a conventional H-CBS at both the local and global scheduling level. Firstly, BROE cannot guarantee at least $\texttt{sbf}_{\Gamma_s}(t)$ processor resources to its task set in any interval of length t within a backlogged period. Secondly, there are many possible server deadlines. An important difference of BROE compared to other protocols is that the worst-case processor supply to a component changes dependent on both (*i*) the size of the statically computed resource holding times $X_{s\ell}$ and (*ii*) the actual time at which a task attempts to access resource R_ℓ. With the other protocols the processor supply merely changes dependent on the size of the statically computed $X_{s\ell}$ values. The latter difference indicates an infinite amount of possibilities for absolute deadlines within a backlogged server period. The lack of a finite set of server deadlines complicates an integration of BROE servers into the HSF by using the enhanced demand-bound test of Baruah (2006) for EDF. Bertogna et al. (2009b) have therefore proven a sufficient utilization-based integration test, as presented in (17).

We conclude that a BROE server is non-compliant with the periodic resource supplies of Shin and Lee (2008) and Kumar et al. (2011). Inherent to the rules of BROE, however, the server has a period P_s (see Bertogna et al., 2009b). Given a period constraint P_s, the bounded-delay model always gives a linear lower bound $\texttt{lsbf}_{\Gamma_s}(t)$ of the actually supplied resources $\texttt{sbf}_{\Gamma_s}(t)$ by a periodic resource Γ_s with the same period and budget parameters. BROE's pessimism in the allocation of processor bandwidth is inherited from the bounded-delay model and the amount of pessimism compared to the periodic resource model heavily depends on the timing characteristics of tasks and the interface parameters of the comprising component.

Clearly, for the arbitration of accesses to global resources BROE cannot get seamlessly attached to any arbitrary component represented by its periodic interface Γ_s. This requires that the budget parameter Q_s in interface Γ_s is linearly estimated by means of the bounded-delay model. Since the resource-supply model is enforced by BROE, the global synchronization protocol influences the computed budget Q_s, thereby violating the notion of opacity in Definition 1.

Nevertheless, one could construct an interface for a component using the periodic resource model and convert it to a bounded-delay interface when BROE is elected for global resource arbitration. An interface $\Gamma_s = (P_s, Q_s, \mathcal{X}_s)$, computed according to (27), represents a virtual task $\tau' = (P_s, Q_s, P_s, \mathcal{X}_s)$. By applying the bounded-delay abstraction using the $\texttt{lsbf}_{\Gamma'_s}(t)$ on the virtual task τ', one can derive a conservative budget Q'_s which $\forall t \geq 0$ upper bounds the periodic supply $\texttt{sbf}_{\Gamma_s}(t)$. According to the method by Lipari and Bini (2005), Q'_s is found by:

$$Q'_s = \frac{-(Y - 2P_s) + \sqrt{(Y - 2P_s)^2 + 8P_sQ_s}}{4}, \tag{35}$$

where $Y = 2P_s - Q_s$.

Reconsidering Example 3: applying the bounded-delay criteria onto the periodic resource model (PRM) $\Gamma_3^{(\texttt{PRM})} = (10, 1, \{0.5\})$ gives a conservative budget of $Q'_3 = 2.5$ time units. Although this method of converting interfaces allows a component to be be analysed with an arbitrary resource-supply model, the derived interface suffers abstraction overheads of two resource-supply models. It is therefore unattractive to convert a resource-supply model at the interface level.

A more processor-efficient solution is to delay the choice of a resource-supply model by deriving two interfaces for each component, $\Gamma_s^{(\texttt{PRM})}$ and $\Gamma_s^{(\texttt{BDM})}$, i.e., one interface for the periodic resource model (PRM) and one interface for its linearly approximated bounded-delay model (BDM). Upon component integration, we select an interface based on the global synchronization protocol.

To summarize, we foresee three solutions to make an opaque local analysis for a component independent of a resource-supply model:

1) **Bounded-delay interfaces:** by assuming an HSF where components are always composed based on bounded-delay interfaces, the periodic supply is by default approximated linearly. Although this is pessimistic for frameworks using ONP, OWP or SIRAP, the compositional model provides an open environment for components even in the presence of shared resources regardless of the synchronization protocol. Moreover, no additional complexity for converting component interfaces is required at the level of composition.
2) **Periodic-supply interfaces:** by assuming an HSF where components are always composed based on periodic-supply interfaces, one must convert the periodic interface into a bounded-delay-compatible interface when BROE is chosen for arbitrating access to global resources. During system composition, this approach may cause pessimism in the interface conversion.
3) **Interface candidates:** by providing an interface candidate for each resource-supply model, one can select the appropriate interface based on the HSF's global scheduling and resource-arbitration policies. During system composition, this approach requires that the resource-supply model is a parameter of the derived component interface, but it prevents that assumptions on the resource-supply model impact the performance of the HSF under certain scheduling policies.

Since we are interested in the relative performance of global synchronization protocols, in this paper we implicitly apply the latter approach.

8.2 Choosing local resource ceilings

Looking at the global system performance, the *global resource ceilings* are optimally configured according to the SRP, see (4). This is irrespective of whether the global scheduling policy is FPPS or EDF, because higher resource ceilings impose more blocking and lower resource ceilings violate mutual exclusive access to a shared resource. Within the hierarchy of an HSF, however, the *local resource ceilings* in (5) require the smallest budget, but may introduce large resource holding times $X_{s\ell}$ and, hence, large blocking terms for other components in the system. This raises the question: how to select local resource ceilings in the most resource-efficient way for the integrated system as a whole?

Each component exposes a number of m_s values of maximum resource holding times, $X_{s\ell} \in \mathcal{X}_s$, of an access to resource R_ℓ in its interface specification.

The local resource ceiling $rc_{s\ell}$ of resource R_ℓ can have at most n_s possible values, leading to different values for resource holding time $X_{si\ell}$ and its derivative $X_{s\ell}$. In general, each of the $n_s^{m_s}$ combinations yields a possible interface $(P_s, Q_s, \mathcal{X}_s)$ - called an *interface candidate*. It is therefore unattractive to explore every combination of interface candidates of composed components.

It is not just the number of iterations for exploring the set of interfaces that explodes exponentially. A component may need a total of 2^{m_s} sets of interface candidates and each of these sets may need to traverse an exponential number of candidates. The reason of this rapid design space explosion is that each resource accessed by a component may affect the size of budget Q_s if the corresponding local resource ceiling is increased artificially. To eliminate the impact of global sharing of a particular resource in case no other component accesses the same resource, a separate set of interface candidates with different local resource ceilings for the other global resources should be constructed. At the same time, also those remaining resources may or may not be shared by other components. Given that a component C_s accesses m_s global resources, this leads to 2^{m_s} sets of interface candidates.

Shin et al. (2008) sidestepped the problem of determining the sharing scope of resources by simply assuming that all resources accessed by a component are globally shared. Next, they proposed a method to explore the space of all possible ceilings of the accessed resources. To simplify this problem, Shin et al. (2008) assume that a component may access an arbitrary resource for a duration of at most X_s time units.

The main observation of Shin et al. (2008) is that a lower local resource ceiling $rc_{s\ell}$ leads to less local blocking, i.e., the lower $rc_{s\ell}$, the lower the contribution of b_{si} to the $\texttt{rbf}_s(t,i)$ in (27) or of $b(t)$ to the $\texttt{dbf}_s(t)$ in (24). Hence, the optimization criterion is represented by a Pareto trade off: the smaller the resource holding time is, the more blocking tasks experience locally and, thus, the more budget a component requires.

Furthermore, given a set of interface candidates for a component, initially an empty set, one only needs to consider settings that lead to an interface candidate which has both a smaller budget Q_s and a smaller value of X_s than any of the existing candidates have. In order to do so, the algorithm by Shin et al. (2008) tries to increase the local resource ceiling that causes the largest X_s and, in addition, the other local resource ceilings are increased if this does not require additional budget. Since all resources are globally shared anyway, it is sufficient to traverse the task set once, i.e., there are only n_s possible local resource ceilings to be considered. Hence the set of interface candidates is at most of size n_s and it can be generated in $\mathcal{O}(n_s)$ iterations.

We observed the above trade off between the amount of local blocking and the amount of budget can fail for non-opaque analyses. For example, SIRAP's analysis may allocate a smaller budget when the local resource ceiling is selected at a higher value than the necessary value in (5) determined by the SRP, so that the resource holding times are smaller. As a result, the added overhead for self-blocking, I_{si}, to the $\texttt{rbf}_s(t, i)$ in (30) is smaller. Even if a global resource is not shared with other components, the term I_{si} is present in the local analysis. Hence, none of the interface candidates in the produced set presents the view of local sharing of a resource. Moreover, since each resource holding time $X_{si\ell}$ may contribute to the size of the budget Q_s differently, each of the $n_s^{m_s}$ combinations of resource ceilings may enter the set of candidates.

Although Shin et al. (2008) implicitly use an opaque analysis, their method has two limitations: (*i*) it only considers ONP and (*ii*) if the local ceiling that determines X_s is increased (with the aim to decrease X_s) while that resource is not globally shared, then the budget of the component is unnecessarily increased. Behnam et al. (2010b) have applied the method by Shin et al. (2008) to BROE. As a consequence of optimizing just X_s, however, all resources are considered as globally shared; this is inconsistent with the local view of opacity.

We lift the assumption by Shin et al. (2008) that all global resources are globally shared. Our main idea is to repeat the step for generating interface candidates of a component for each resource separately. We derive a set of *partial interface candidates* for each resource accessed by the component, assuming this is the only resource that is globally shared with other components. Similar to Shin et al. (2008), the use of an opaque local analysis ensures that the lowest local resource ceiling yields a partial interface candidate that allows us to consider also this globally exposed resource as a local resource. We derive m_s sets of partial interface candidates and in case $m_s = 1$ our method specializes to the method proposed by Shin et al. (2008).

As a consequence of splitting interfaces into partial candidates per global resource, the selection criterion for the component interfaces (i.e., the actual selection of the local ceilings) of an entire system also becomes more complicated than the approach in Shin et al. (2008). Those partial candidates corresponding to the resources that are actually shared globally are combined into a timing interface of a component at integration time. In other words, the complete set of interface parameters for one component is derived from a selection of at most m_s of its partial candidates. To optimize the system load of a combination of resource-sharing components, we extend the interface selection method by Shin et al. (2008) by explicitly optimizing the blocking on the bottleneck resource.

In the remainder of this section, we first explain the measure of the system load, being used as our optimization criterion. Next, we present an algorithm to derive partial interface candidates. Finally, we show how partial interfaces can be combined into a true interface of a component.

8.2.1 System load

The system load is a quantitative measure to represent the minimum amount of processor resources necessary to guarantee the schedulability of the system. Its value therefore depends on the global scheduling policy.

From the global schedulability tests in (18) for FPPS of components, we define the processor request bound (α_s) as

$$\alpha_s \quad = \quad \min\left\{ \frac{B_s + \mathtt{RBF}(t,s)}{t} \;\middle|\; 0 < t \leq P_s \right\}. \tag{36}$$

Similarly, from the global schedulability tests in (17) for EDF scheduling of components[5], we define the processor demand bound (α_s), i.e.,

$$\alpha_s \quad = \quad \left(\frac{B(P_s)}{P_s} + \sum_{1 \leq u \leq s} \frac{Q_u + O_u(P_u)}{P_u} \right). \tag{37}$$

From the processor request bound in (36) and the processor demand bound in (37), we derive the system load, i.e.,

$$\mathrm{load}_{\mathtt{sys}} \quad = \quad \max\{\alpha_s \mid 1 \leq s \leq N\}. \tag{38}$$

The system load in (38) is determined by the component that stresses the system the most. If the $\mathrm{load}_{\mathtt{sys}} > 1$, then the system is unschedulable. This satisfies the schedulability conditions presented in Section 5.1, assuming a global resource-supply function of $\alpha_s t$. One can think of the system load as decreasing the speed of the processor by a factor $\mathrm{load}_{\mathtt{sys}}$. This increases a component's budget Q_s, resource holding times $X_{s\ell}$, and blocking times by a factor $1/\mathrm{load}_{\mathtt{sys}}$.

Example 4: Consider a system with a global FPPS scheduler comprising two components (C_1 and C_2) and one global resource R_1 arbitrated by ONP. Component C_1 has an interface $\Gamma_1 = (10,\ 1,\ \{0.5\})$ and C_2 has an interface $\Gamma_2 = (48,\ 1,\ \{1\})$. According to (36), we obtain $\alpha_1 = 0.25$ and $\alpha_2 = 0.198$. Using (38) we derive $\mathrm{load}_{\mathtt{sys}} = \alpha_1 = 0.25$, meaning both C_1 and C_2 can be scheduled together on a processor with a speed of factor 0.25.

Given a set of interface candidates for all the components in the system, the objective of our optimization is to find one interface for each component such that the system load is minimized.

5. Without loss of generality, we rewrote the utilization-based test in (17) instead of the demand-bound test in (11), so that the processor demand bound in (37) is also compatible with the bounded-delay model used by BROE.

8.2.2 Generating partial interface candidates

To accomplish our goal of optimizing the system load, we first need to generate the interface candidates of a component. The problem of generating interface candidates is as follows. Given a component C_s and a set of global resources, the problem is to generate a set of interface candidates such that there must exist an interface that contributes the least possible (optimal) system load. The partial interface candidates generated by the algorithms in this section are an intermediate step to the problem of selecting interface candidates.

As explained before, for a given component C_s with n_s tasks accessing m_s resources, each resource may have up to n_s different local resource ceilings and one interface candidate can be generated from each combination of the m_s local resource ceilings. Fortunately, not all the $n_s^{m_s}$ candidates have the potential to minimize the system load; those that require more processor demand and impose more blocking on other components can be considered as redundant.

Since a component is unaware of other components, it is also unknown which resources are shared. Instead of directly deriving the interface candidates, we therefore perform an intermediate step, i.e., we derive partial interface candidates. In line with the definition of opacity, these partial interface candidates can be combined into a true interface by selecting only the local ceilings of the resources that are globally shared upon integration of components.

We present the *generatePartialCandidates* algorithm that is computationally efficient and it produces also a bounded number of partial interface candidates. We first provide some notions and properties on which our algorithm is based. Given a component C_s, we assume that P_s is given by the system designer and is fixed during the whole process of generating a set of partial interface candidates.

A partial interface $\Gamma_{s\ell}$ considers one global resource R_ℓ in isolation, i.e., R_ℓ can be globally shared or it can be local to the component.

Definition 2: A *partial interface candidate* $\Gamma_{s\ell} = (P_s,\ Q_{s\ell},\ \{X_{s\ell}\})$ of a component C_s accessing resource R_ℓ is a valid interface Γ_s for component C_s - satisfying the opaque local schedulability conditions in (24) or (27) - under the assumption that only resource R_ℓ is globally shared with other components.

Since a partial interface $\Gamma_{s\ell}$ is a valid interface for the restrictive case where the resource R_ℓ is the only resource that needs to be shared globally, the additional bandwidth $O_s(t)$ of $\Gamma_{s\ell}$ - defined in Section 5.1 to prevent excessive blocking - takes into account R_ℓ in isolation as well, i.e., $X_s = X_{s\ell}$.

For the simple case of considering resource R_ℓ and ignoring all other resources $R_k \in \mathcal{R}_s \setminus \{R_\ell\}$, we derive a set of interfaces in a similar way as Shin et al. (2008). We are looking for a closed set of non-redundant partial interfaces.

Definition 3: A partial interface candidate $\Gamma_{s\ell} = (P_s,\ Q_s,\ \{X_{s\ell}\})$ is *redundant* if there exists $\Gamma'_{s\ell} = (P_s,\ Q'_s,\ \{X'_{s\ell}\})$ such that $X'_{s\ell} \leq X_{s\ell}$ and $Q'_s \leq Q_s$ (denoted as $\Gamma'_{s\ell} \leq \Gamma_{s\ell}$). Otherwise, $\Gamma'_{s\ell} = (P_s,\ Q_s,\ \{X_{s\ell}\})$ is non-redundant.

Intuitively, a redundant interface $\Gamma_{s\ell}$ as defined by Definition 3 can never cause a smaller system load than its non-redundant dominator $\Gamma'_{s\ell}$. Since we only consider a single resource in isolation, this intuition can be confirmed similarly to Lemma 1 by Shin et al. (2008).

Lemma 4: If $\Gamma'_{s\ell} \leq \Gamma^*_{s\ell}$, then $\Gamma'_{s\ell}$ cannot contribute more $\text{load}_{\texttt{sys}}$ than $\Gamma^*_{s\ell}$ does.

Proof: Assume $\Gamma^*_{s\ell}$ is redundant. Using Definition 3, there exists a $\Gamma'_{s\ell}$, such that $X'_{s\ell} \leq X_{s\ell}$ and $Q'_s \leq Q_s$. Looking at the definition of the system load in (38), it can be decreased by means of decreasing the blocking and/or the demand-part, e.g., $\texttt{DBF}(t)$ or $\texttt{RBF}(t, s)$. Since $X'_{s\ell} \leq X^*_{s\ell}$, $\Gamma^*_{s\ell}$ cannot reduce the blocking part of other components. Since $Q'_{s\ell} + O'_s(P_s) \leq Q^*_{s\ell} + O^*_s(P_s)$, $\Gamma^*_{s\ell}$ cannot reduce the demand part. Hence, $\Gamma^*_{s\ell}$ cannot decrease $\text{load}_{\texttt{sys}}$. □

Lemma 4 reduces the size of the set of partial interface candidates significantly. However, an exhaustive search for non-redundant candidates is still computationally intractable. Since we consider a single resource in isolation, we just need to traverse at most n_s possible ceilings for this resource.

From the above definitions and lemma, we can derive an efficient algorithm to compute the partial interface candidates of a component while considering one resource, R_ℓ, as a global resource. Algorithm 1 presents the details and, in case a component accesses only a single resource, the algorithm is exactly the same as the interface-candidate-generation algorithm by Shin et al. (2008).

Algorithm description. Initially, each resource ceiling rc_{sk} ($\forall k\ :\ 1 \leq k \leq m_s$) is set to the lowest value according to the SRP (line 1-2), i.e., see (5). Next, the loop in line 3-9 tries to increase the local resource ceiling of the resource under consideration, i.e., resource R_ℓ. For each of the possible ceilings of R_ℓ a number of steps is applied. Given the reconfigured local resource ceiling $rc_{s\ell}$, a new budget Q_s is computed for the component (line 4) and the resource holding times to resource R_ℓ are recomputed (line 5-6). These computed parameters Q_s and $X_{s\ell}$ define a partial interface candidate $\Gamma_{s\ell} = (P_s,\ Q_{s\ell},\ \{X_{s\ell}\})$ which is added to the set $\mathcal{IC}_s^{(\ell)}$ (line 7). Finally, the redundant partial interface candidates are removed from $\mathcal{IC}_s^{(\ell)}$ (line 8).

In algorithm 2, we apply Algorithm 1 to each of the m_s resources accessed by component C_s (line 1-4). This results in a matrix of partial interface candidates $\mathcal{IC}_s$, where each row represents the partial interface candidates $\mathcal{IC}_s^{(\ell)}$ for the case one particular resource R_ℓ is globally shared. In case a component does not access any resource, we can directly compute its interface (line 5-8).

Complexity. Algorithm 1 executes at most $\mathcal{O}(n_s)$ iterations, because resource R_ℓ has at most n_s possible ceilings. Hence, at most n_s candidates are stored in the set $\mathcal{IC}_s^{(\ell)}$. Also note that the removal of redundant candidates allows us to further decrease the set of candidates. Since algorithm 2 generates m_s sets of partial interface candidates, computing the entire matrix $\mathcal{IC}_s$ takes $\mathcal{O}(m_s n_s)$

Algorithm 1 generateSimplePartialCandidates($\mathcal{T}_s$, P_s, R_ℓ)

```
Require: - inputs - a task set $\mathcal{T}_s$, a component period $P_s$ and a single global
    resource $R_\ell$.
Require: calculateBudget($\mathcal{T}_s, P_s, \mathcal{RC}_s$);
            returns the smallest component budget that satisfies (24) or (27).
Require: increaseCeilingX$_{s\ell}$ ($\mathcal{RC}_s[\ell]$);
            returns whether or not the ceiling of the resource $X_{s\ell}$ can be
            increased by one. If so, it increases the ceiling of the selected
            resource.
Require: removeRedundant($\mathcal{IC}_s^{(\ell)}$);
            removes all redundant partial interfaces from the interface list,
            $\mathcal{IC}_s^{(\ell)}$.
Ensure: - output - a set of partial interfaces $\mathcal{IC}_s^{(\ell)}$ for the case $R_\ell$ is globally
    shared.
1: {Set the local resource ceilings $rc_{s1}, \dots, rc_{sm_s}$ according to the SRP, see (5):}
2: $\mathcal{RC}_s = \{rc_{s1}, \dots, rc_{sm_s}\}$ ;
3: repeat
4:     $Q_{s\ell} \leftarrow$ calculateBudget($\mathcal{T}_s, P_s, \mathcal{RC}_s$);
5:     for each $\tau_{si} \in \mathcal{T}_s$ do compute $X_{si\ell}$ using (6) end for
6:     $X_{s\ell} \leftarrow \max\{X_{si\ell} \mid 1 \le i \le n_s\}$ ;
7:     $\mathcal{IC}_s^{(\ell)} \leftarrow \mathcal{IC}_s^{(\ell)} \bigcup \{\Gamma_{s\ell} = (P_s, Q_{s\ell}, \{X_{s\ell}\})\}$;
8:     removeRedundant($\mathcal{IC}_s^{(\ell)}$);
9: until (increaseCeilingX$_{s\ell}$ ($\mathcal{RC}_s[\ell]$) = false);
10: return $\mathcal{IC}_s^{(\ell)}$;
```

iterations, resulting in at most $m_s n_s$ partial candidates.

8.2.3 Merging partial candidates into an interface

The limitation of a partial interface candidate $\Gamma_{s\ell}$ for a component C_s accessing resource R_ℓ is that it specifies the budget and the resource holding time to resource R_ℓ of the component for a particular local resource ceiling $rc_{s\ell}$, but other resources accessed by the same component are ignored. Algorithm 2 generates these partial interfaces for all resources accessed by the component. The remaining problem is to derive an interface for the case a component accesses more than one globally shared resource.

Assume we are given which of the resources accessed by a component must be shared globally. For each of those shared resources we select a local resource ceiling defined by a partial interface candidate. This can be done by selecting at most one partial interface candidate in each of the rows of $\mathcal{IC}_s$ (generated

Algorithm 2 generatePartialCandidates($\mathcal{T}_s$, P_s)

```
Require: - inputs - a task set $\mathcal{T}_s$ and a component period $P_s$.
Require: calculateBudget($\mathcal{T}_s, P_s, \mathcal{RC}_s$);
            returns the smallest component budget that satisfies (24) or (27).
Require: generateSimplePartialCandidates($\mathcal{T}_s$, $P_s$, $R_k$);
            returns the result from Algorithm 1.
Ensure: - output - a matrix $\mathcal{IC}_s$ containing an array $\mathcal{IC}_s^{(\ell)}$ of partial interface
    candidates in each row, repeated for all the globally accessed resources
    $R_\ell \in \mathcal{R}_s$.
1: for $k \leftarrow 1;\ k \leq m_s;\ k \leftarrow k+1$ do
2:     $\mathcal{IC}_s^{(k)} \leftarrow$ generateSimplePartialCandidates($\mathcal{T}_s$, $P_s$, $R_k$);
3:     $\mathcal{IC}_s \leftarrow \mathcal{IC}_s \bigcup \left\{\mathcal{IC}_s^{(k)}\right\}$
4: end for
5: if $m_s = 0$ then
6:     $Q_s \leftarrow$ calculateBudget($\mathcal{T}_s, P_s, \emptyset$);
7:     $\mathcal{IC}_s \leftarrow \{\{\Gamma_s = (P_s, Q_s, \emptyset)\}\}$
8: end if
9: return $\mathcal{IC}_s$;
```

by Algorithm 2). In other words, we must select one interface in $\mathcal{IC}_s^{(\ell)}$ of a globally shared resource R_ℓ. Lemma 5 shows how to combine the selected partial interface candidates into a true interface candidate.

Lemma 5: Assume a component C_s accesses m_s resources that all need to be shared globally. Let $\Gamma_{s1} = (P_s,\ Q_{s1},\ \{X_{s1}\}), \ldots, \Gamma_{sm_s} = (P_s,\ Q_{sm_s},\ \{X_{sm_s}\})$ be a selection of partial interfaces of C_s generated by Algorithm 2, i.e., one partial interface is selected from $\mathcal{IC}_s$ for each of the m_s shared resources. The schedulability of component C_s is satisfied by interface $\Gamma_s = (P_s,\ Q_s,\ \{X_{s\ell} \mid 1 \leq \ell \leq m_s\})$, where $Q_s = \max\{Q_{s\ell} \mid 1 \leq \ell \leq m_s\}$.

Proof: Let Q'_s be the smallest budget such that all tasks $\tau_{si} \in \mathcal{T}_s$ make their deadline while the local resource ceilings of all accessed resources are configured according to the SRP, see (5). By definition all partial candidates of component C_s generated by Algorithm 2 satisfy the following constraint:

$$(\forall \ell\ :\ 1 \leq \ell \leq m_s \wedge \Gamma_{s\ell} = (P_s,\ Q_{s\ell},\ \{X_{s\ell}\})\ :\ Q_{s\ell} \geq Q'_s)\,. \tag{39}$$

In Algorithm 1, an increase of only one local resource ceiling, i.e., $rc_{s\ell}$, is allowed and the additional amount of local blocking caused by this increase can result in an increase of $Q_{s\ell}$ compared to the smallest possible budget Q'_s. However, by virtue of the SRP a task τ_{si} can be blocked by just one (outermost) critical section of a lower priority task τ_{sj} (where $\pi_{si} \geq \pi_{sj}$) before τ_{si} can start its execution. Hence, if one or more ceilings $rc_{s1} \dots rc_{sm_s}$ are increased compared to the value required by the SRP, then it is sufficient to add the largest difference in budget to the value of Q'_s in order to accommodate for one local blocking occurrence. Combining this observation with the property in (39) of the partial interface candidates concludes our proof. □

Lemma 5 makes it possible to merge an arbitrary number of partial candidates into a single interface of a component. Prior to the integration of components, however, it is still unclear which of the global resources need to be shared with other components and which resources can be treated as local. The step of merging partial candidates into a true interface is therefore postponed until integration time (as will be described in Section 8.3), i.e, the interface candidates of a component that capture all resource dependencies into $\mathcal{R}_s$ are derived on the fly during global integration of components. Given such a set $\mathcal{R}_s$ and a selection of partial interface candidates, Algorithm 3 presents the steps derived from Lemma 5.

Algorithm 3 mergePartialCandidateIntoInterface(Γ_s, $\mathcal{R}_s$, ℓ, $ic_{s\ell}$)

```
Require: - inputs -
    • an initial interface $\Gamma_s = (P_s,\ Q_s,\ \mathcal{X}_s)$;
    • the set $\mathcal{R}_s$ of accessed global resources that are exposed globally;
    • a resource $R_\ell$;
    • a partial interface candidate in $\mathcal{IC}_s^{(\ell)}$ for resource $R_\ell$ at index $ic_{s\ell}$.
Ensure: - output - An updated interface $\widehat{\Gamma_s} = (P_s,\ Q'_s,\ \mathcal{X}'_s)$ such that all local
    ceilings are as high as the selected budget $Q_{s\ell}$ of partial candidate $\mathcal{IC}_s^{(\ell)}[ic_{s\ell}]$
    allows.
 1: if $1 \le ic_{s\ell} \le \left|\mathcal{IC}_s^{(\ell)}\right|$ then
 2:   Let $\mathcal{IC}_s^{(\ell)}[ic_{s\ell}] = (P_s,\ Q_{s\ell},\ \{X'_{s\ell}\})$ be the next partial candidate of $C_s$;
 3:   $Q_s \leftarrow \max(Q_s,\ Q_{s\ell})$;
 4: else
 5:   {no valid partial interface is given, so skip the update of $Q_s$.}
 6: end if
 7: for each $R_k \in \mathcal{R}_s$ do
 8:   {Try to increase the local resource ceilings $rc_{sk}$ without increasing $Q_s$:}
 9:   Let $\mathcal{IC}_s^{(k)}[ic_{sk}] = (P_s,\ Q_{sk},\ \{X'_{sk}\})$;
10:   for $\left(ic_{sk};\ ic_{sk} \le \left|\mathcal{IC}_s^{(k)}\right| \bigwedge Q_{sk} \le Q_s;\ ic_{sk} \leftarrow ic_{sk}+1\right)$ do
11:     $\mathcal{X}_s \leftarrow (\mathcal{X}_s \setminus \{X_{sk}\}) \bigcup \{X'_{sk}\}$;
12:   end for
13: end for
14: return $\widehat{\Gamma_s} \leftarrow (P_s,\ Q_s,\ \mathcal{X}_s)$;
```

Algorithm 3 takes as an input a given interface candidate Γ_s and an index $ic_{s\ell}$ to a partial interface candidate $\Gamma_{s\ell}$ in the array $\mathcal{IC}_s^{(\ell)}$. We merge the given interface and the partial interface into a new interface candidate $\widehat{\Gamma_s}$. First, we update the budget Q_s according to Lemma 5 (line 2-3). Next, we update all the local resource ceilings of the globally shared resources (line 7-13), i.e., we increase the indices ic_{sk} of all globally shared resources $R_k \in \mathcal{R}_s$ as long as this does not increase the size of budget Q_s. This efficiently exploits the increase in budget (line 3) being imposed by partial candidate $\mathcal{IC}_s^{(\ell)}[ic_{s\ell}]$. That is, we keep any of the resource holding times in $\mathcal{X}_s$ as small as possible (see line 11). Finally, the new interface candidate $\widehat{\Gamma_s}$ is returned (line 14).

In the next section, we present an algorithm to select interface candidates efficiently. To conclude this section, we have shown that a non-opaque analysis requires 2^{m_s} sets of interface candidates and the entire exponential $n_s^{m_s}$ design space of possible local resource ceilings must be traversed in order to derive each of these sets. Contrary, a total of m_s sets of partial interface candidates is sufficient with an opaque analysis and the size of each set is at most n_s (representing the Pareto-optimal local resource ceilings). Each of the m_s sets can be obtained within $\mathcal{O}(n_s)$ iterations. As soon as it is known which resources are globally shared, the partial interfaces can be combined into a true interface candidate. In the next section, we present a method to select an interface candidate with the objective to optimize the system load.

8.3 Optimized composition of components by interface selection

This section considers the problem of selecting the optimal system configuration, i.e., one interface is derived from its partial candidates per component and all interfaces of the components together minimize the system load. The algorithm presented in this section (Algorithm 4) finds an optimal solution to this problem through a polynomial number of iterative steps.

Algorithm 4, called selectInterfaceCandidates($\mathcal{IC}_1, \ldots, \mathcal{IC}_N$), assumes that the partial candidates in each set of the given sets $\mathcal{IC}_s^{(\ell)} \in \mathcal{IC}_s$ (at least those for R_ℓ that are globally shared) are sorted in an increasing order of the total bandwidth allocated to component C_s. Then, the first partial candidate of $\mathcal{IC}_s^{(\ell)}$ has the largest resource holding time $X_{s\ell}$, but the smallest budget $Q_{s\ell}$. The key strategy behind our algorithm is that in each iteration the total amount of bandwidth allocation, i.e., $Q_s + O_s(P_s)$, to at least one component is increased.

To make this strategy work, Lemma 6 removes redundant partial interface candidates from $\mathcal{IC}_s^{(\ell)}$, given R_ℓ is globally shared. Lemma 6 generalizes Lemma 5 by Shin et al. (2008) for other synchronization protocols than ONP.

Lemma 6: Consider two partial candidates $\Gamma'_{s\ell} = (P_s, Q'_{s\ell}, X'_{s\ell})$ and $\Gamma^*_{s\ell} = (P_s,\ Q^*_{s\ell},\ X^*_{s\ell})$ such that $Q'_{s\ell} + O'_s(P_s) \leq Q^*_{s\ell} + O^*_s(P_s)$ and $X'_{s\ell} \leq X^*_{s\ell}$. Then, $\Gamma'_{s\ell}$ will never contribute more to load_{sys} than $\Gamma^*_{s\ell}$ does.

Proof: See the proof of Lemma 4. □

We use Lemma 6 to preprocess the input sets $\mathcal{IC}_s$. Next, our selection algorithm generates a polynomial number of system configurations and each configuration is represented by a set of N interfaces $\{\Gamma_s \mid 1 \leq s \leq N\}$.

8.3.1 Initialization of the interface selection

In the beginning of Algorithm 4 (line 1-20), our algorithm computes an initial configuration such that it consists of the first interface candidates of all components, i.e, each of the components has the smallest possible budget. To construct such an interface for a component C_s, we identify the partial interfaces of the global resources that need to be shared by multiple components (see the loop in line 3-11). The variable $ic_{s\ell}$ is a counter that iterates through the array $\mathcal{IC}_s^{(\ell)}$ (initialized in line 5). Line 6 preprocesses the array $\mathcal{IC}_s^{(\ell)}$. First, it applies Lemma 6 to $\mathcal{IC}_s^{(\ell)}$ to remove redundant candidates. Moreover, it removes all infeasible partial candidates, i.e., it is required that the total amount of allocated bandwidth satisfies the constraint $Q_{s\ell} + O_s(P_s) \leq P_s$, because this guarantees that a granted access to a global resource can be completed in the same component period. The preprocessing of $\mathcal{IC}_s^{(\ell)}$ is completed by sorting the remaining candidates in increasing order of $Q_{s\ell} + O_s(P_s)$. If no partial candidate is feasible, then global sharing of resource R_ℓ makes the system infeasible (line 7). Line 9 constructs a set $\mathcal{R}'_s \subseteq \mathcal{R}_s$ of the resources that need to be shared globally. For each of those resources in $\mathcal{R}'_s$, the interface Γ_s is updated by repeatedly applying Algorithm 3 (line 10).

If a component only accesses resources that no other component wishes to access, then we can treat all accessed resources as local, i.e., the component is independent of other components (line 12-17).

Finally, after initializing the interface Γ_s of a component, we keep track of the successor interface $\widehat{\Gamma_s}$ (line 19). The latter interface $\widehat{\Gamma_s}$ denotes the interface where some of the partial interfaces $\mathcal{IC}_s^{(\ell)}[ic_{s\ell}]$ have been replaced by their successor $\mathcal{IC}_s^{(\ell)}[ic_{s\ell}+1]$. However, both Γ_s and $\widehat{\Gamma_s}$ are the same initially. For this initial configuration $\{\widehat{\Gamma_s} \mid 1 \leq s \leq N\}$, $\text{load}_{\textsf{sys}}$ is computed (line 29).

8.3.2 Iterations of interface selections

In each iteration of Algorithm 4 (line 23-37), we try to improve the system load. Let s^* denote the component C_{s^*} with the largest processor request bound among all components, i.e, $\text{load}_{\textsf{sys}} = \alpha_{s^*}$ according to (38). Component C_{s^*} is found by inspecting α_{s^*} of all components (line 35). By definition of s^*, we can further reduce the value of $\text{load}_{\textsf{sys}}$ by reducing the left-hand sides of the terms in the schedulability tests in (17) or (18). There are two ways to reduce these terms. One option is to reduce its maximum blocking $B(t)$ or B_{s^*} and the other option is to reduce the component's processor demands, i.e., $Q_s + O_s(P_s)$.

Similar to Shin et al. (2008) our algorithm always reduces the blocking part, but it does not reduce the request-bound or demand-bound part. The reason is that the algorithm starts from the interface candidates that have the smallest demands and the largest resource holding times. Hence, for each interface candidate, there is no room to further reduce its demand. However, there is a chance to reduce the maximum blocking to component C_{s^*}. It can be reduced by decreasing the $X_{u\ell}$ of a component C_u that imposes the largest blocking to component C_{s^*} by accessing the same resource R_ℓ as C_{s^*} wishes to access. Note that multiple components can cause the same amount of maximum blocking or a component can cause the maximum amount of blocking to C_{s^*} via more than one resource. We collect all those pairs $(u,\ \ell)$ in a set $\mathcal{B}$.

Each pair $(u,\ \ell) \in \mathcal{B}$ represents a component C_u that wishes to access resource R_ℓ, thereby causing the maximum blocking to C_{s^*} (line 36). In each iteration of the main loop (line 23-37), we select the next partial interface in $\mathcal{IC}_u^{(\ell)}$ (see line 24-28). The interface $\widehat{\Gamma_u}$ of component C_u is updated by merging its previous interface $\widehat{\Gamma_u}$ with the partial interface candidate $\mathcal{IC}_u^{(\ell)}[ic_{u\ell}+1]$; this is done by using Algorithm 3 (line 26-27). We repeat these steps for all the pairs $(u,\ \ell) \in \mathcal{B}$ that cause the same amount of maximum blocking to C_{s^*}.

After changing the configurations of the components C_u captured in $\mathcal{B}$, we recompute $\text{load}_{\mathsf{sys}}$ for the set of interfaces $\{\widehat{\Gamma_s} \mid 1 \le s \le N\}$ (line 29). Dependent on whether or not the changed system configuration has the potential to reduce the system load, the loop in line 23-37 is repeated. In other words, all elements in the recomputed set $\mathcal{B}$ (line 36) must make progress, otherwise the system load cannot further decrease. This condition is tested in the *until*-clause (line 37).

Algorithm 4 selectInterfaceCandidates($\mathcal{IC}_1, \ldots, \mathcal{IC}_N$)

Require: - inputs - the matrices $\mathcal{IC}_1, \ldots, \mathcal{IC}_N$ with partial interface candidates.

Require: preprocessCandidates($\mathcal{IC}_s^{(\ell)}$);

first, apply Lemma 6 to $\mathcal{IC}_s^{(\ell)}$; next, remove the infeasible partial candidates with $Q_{s\ell} + O_s(P_s) > P_s$ and sort the remaining candidates in increasing order of $Q_{s\ell} + O_s(P_s)$.

Require: componentWithMaxLoad($\{\widehat{\Gamma_s} \mid 1 \le s \le N\}$);

returns the component C_{s^*} with the largest processor load, i.e., $\text{load}_{\mathsf{sys}} = \alpha_{s^*}$.

Require: maxBlockingComponentResourcePairsToSystemload(s^*, $\{\widehat{\Gamma_s} \mid 1 \le s \le N\}$);

returns a set of pairs of a component C_u and a corresponding resource R_ℓ that produce the largest blocking to component C_{s^*} (where C_{s^*} determines $\text{load}_{\mathsf{sys}}$).

Ensure: - output - if the system is feasible, we return a set $\{\Gamma_s \mid 1 \le s \le N\}$ that contains one interface for each of the N components. This set imposes the smallest system load.

```
for $s \leftarrow 1;\ s \leq N;\ s \leftarrow s+1$ do
    $\Gamma_s \leftarrow (P_s,\ 0,\ \emptyset)$; $\mathcal{R}'_s \leftarrow \emptyset$;
    for each $R_\ell \in \mathcal{R}_s \bigcap \left(\bigcup_{(1 \leq u \leq N \wedge u \neq s)} \mathcal{R}_u\right)$ do
        {Initialize index $ic_{s\ell}$ in array $\mathcal{IC}_s^{(\ell)}$ and check whether sharing of $R_\ell$ can be feasible:}
        $ic_{s\ell} \leftarrow 1$;
        preprocessCandidates($\mathcal{IC}_s^{(\ell)}$);
        if $\mathcal{IC}_s^{(\ell)} = \emptyset$ then return infeasible end if
        {Select the smallest $Q_s$ and the largest $X_{s\ell}$ satisfying Lemma 5:}
        $\mathcal{R}'_s \leftarrow \mathcal{R}'_s \bigcup \{R_\ell\}$;
        $\Gamma_s \leftarrow$ mergePartialCandidateIntoInterface($\Gamma_s,\ \mathcal{R}'_s,\ \ell,\ ic_{s\ell}$);
    end for
    if $\mathcal{R}'_s = \emptyset$ then
        {$C_s$ is independent of the other components; throw away all its candidates:}
        $Q_s \leftarrow \min\left\{Q_{s\ell} \mid \mathcal{IC}_s^{(\ell)}[1] = (P_s,\ Q_{s\ell},\ \{X_{s\ell}\}) \bigwedge 1 \leq \ell \leq m_s\right\}$;
        $\Gamma_s \leftarrow (P_s,\ Q_s,\ \emptyset)$;
        $\mathcal{IC}_s \leftarrow \emptyset$;
    end if
    {Initialize the successor $\widehat{\Gamma_s}$ of candidate $\Gamma_s$:}
    $\widehat{\Gamma_s} \leftarrow \Gamma_s$;
end for
load$^*_{\text{sys}} \leftarrow \infty$;
$\mathcal{B} \leftarrow \emptyset$;
repeat
    for each $(u,\ell) \in \mathcal{B}$ do
        {Decrease the $X_{u\ell}$ of component $C_u$ to $R_\ell$:}
        $ic_{u\ell} \leftarrow ic_{u\ell} + 1$;
        $\widehat{\Gamma_u} \leftarrow$ mergePartialCandidateIntoInterface($\widehat{\Gamma_u},\ \mathcal{R}'_u,\ \ell,\ ic_{u\ell}$);
    end for
    compute load$_{\text{sys}}$ for $\left\{\widehat{\Gamma_s} \mid 1 \leq s \leq N\right\}$ according to (38);
    if load$_{\text{sys}}$ < load$^*_{\text{sys}}$ then
        {Select the interfaces $\widehat{\Gamma_u}$ that have resulted into the better system load:}
        for $u \leftarrow 1;\ u \leq N; u \leftarrow u+1$ do $\Gamma_u \leftarrow \widehat{\Gamma_u}$; end for
        load$^*_{\text{sys}} \leftarrow$ load$_{\text{sys}}$;
    end if
    $s^* \leftarrow$ componentWithMaxLoad( $\{\widehat{\Gamma_s} \mid 1 \leq s \leq N\}$ );
    $\mathcal{B} \leftarrow$ maxBlockingComponentResourcePairsToSystemload( $s^*$, $\{\widehat{\Gamma_s} \mid 1 \leq s \leq N\}$ );
until $\left(\exists (u,\ell) \in \mathcal{B}\ :\ \mathcal{B} \neq \emptyset\ :\ ic_{u\ell} \geq \left|\mathcal{IC}_u^{(\ell)}\right|\right)$
if load$^*_{\text{sys}} > 1$ then return infeasible end if
return $\{\Gamma_s \mid 1 \leq s \leq N\}$ determining load$^*_{\text{sys}}$;
```

8.3.3 Termination of the interface selection

Algorithm 4 terminates (line 38-39), if there exists a component C_u that causes the maximum blocking to component C_{s^*} via resource R_ℓ and C_u cannot decrease its resource holding time $X_{u\ell}$. Then, a set of interface candidates is returned (line 39), i.e., if the set is deemed feasible (line 38). The returned set minimizes the system load.

Theorem 4: When Algorithm 4 terminates, it returns a set of interfaces $\{\Gamma_s \mid 1 \leq$

$s \leq N\}$ imposing the smallest possible system load, i.e., it is impossible to find another interface $\widehat{\Gamma_s}$ that further decreases the system load.

Proof: Assume it is possible to further decrease the system load. Let s^* denote the component C_{s^*} that determines the system load, i.e., $\text{load}_{\mathtt{sys}} = \alpha_{s^*}$. Since all the partial interface candidates are sorted in increasing order of $Q_{s\ell} + O_s(P_s)$, it is impossible to reduce the demand-part of C_{s^*}. The only other way to reduce the system load is to reduce the blocking to C_{s^*}. Our algorithm terminates when one of the pairs $(u, \ell) \in \mathcal{B}$ that cause the most blocking to C_{s^*} cannot select a new partial interface candidate in $\mathcal{IC}_u^{(\ell)}$ (line 37). In this case the blocking to C_{s^*} cannot be reduced and the value of α_{s^*} therefore remains the same. In other words, whenever we move from one step to another one, our selection is the only that can decrease the system load. Hence, when Algorithm 4 terminates, the system load cannot be decreased any further. □

From Theorem 4 we conclude that Algorithm 4 returns an interface for each component in the system (if the system is feasible). Also note that Algorithm 4 always terminates, because the number of interface-selection steps is finite. The returned interfaces together minimize the system load.

8.3.4 Complexity of the interface selection algorithm

Given a set of N components sharing M global resources, the initialization of Algorithm 4 (line 1-22) takes $\mathcal{O}(NM)$ iterations. The dominating part in terms of the computational complexity is the while-loop in line 23-37. In each iteration (except in the first iteration), at least one of the iterators $ic_{s\ell}$ is incremented (line 26); they are never decremented. The progress of at least one of the iterators $ic_{s\ell}$ is ensured by the guard of the main loop (line 37). Algorithm 4 therefore traverses at most $(n_s m_s + 1)$ combinations of partial candidates per component C_s. Hence, the set of interfaces of N components (optimal with respect to the system load) is computed by Algorithm 4 within $\mathcal{O}(NMn_s^{\mathtt{max}})$ iterations, where $n_s^{\mathtt{max}} = \max\{n_s \mid 1 \leq s \leq N\}$.

Algorithm 4 efficiently selects an interface that minimizes the processor requirements of a system by applying an opaque analysis to the individual components. The system load of the selected interfaces may be non-optimal beyond the scope of opaque analyses. Although a non-opaque local analysis may further tighten the system's analysis, the presented algorithms may be unable to find the optimal configurations of the local resource ceilings. An opaque analysis is therefore instrumental for a tractable exploration of the design space of resource-sharing components in HSFs.

8.4 A case study

We now present a case study to demonstrate the algorithms presented in this section. First, we consider the two steps at the component level from Section 8.1

and Section 8.2. Next, we show the integration of components using the step from Section 8.3, where we compose a system from different components.

Example 5: We consider a component C_5 with $P_5 = 125$ and a task set comprising 6 fixed-priority-scheduled tasks. The component C_5 requires access to 2 global resources, R_1 and R_2. The characteristics of the task set are given in Table 3. The same task set is used as an example by Shin et al. (2008).

First, we have to choose a resource-supply model. Without loss of generality, we will use the periodic resource model by Shin and Lee (2008).

Secondly, we generate the partial interface candidates IC_s using Algorithm 2. This requires 2 iterations: (*i*) for resource R_1 Algorithm 1 returns a set $\mathcal{IC}_s^{(1)}$ containing 3 non-redundant partial interface candidates within 3 iterations and (*ii*) for resource R_2 Algorithm 1 returns a set $\mathcal{IC}_s^{(2)}$ containing 2 non-redundant partial interface candidates within 6 iterations. Table 4 presents the iterations of Algorithm 1 and the resulting partial interface candidates.

For resource R_1 we can choose 3 different local resource ceilings and each of the choices leads to a non-redundant partial interface candidate. For resource R_2 we can choose 6 different local resource ceilings. Since budget Q_{52} is the same for $rc_{52} = \pi_{52} \ldots \pi_{56}$ and the resource holding times X_{5i2} reduce in each iteration, the partial interface candidates corresponding to $rc_{52} = \pi_{53} \ldots \pi_{56}$ are redundant.

Table 3
Example task set of component C_5.

Task	$T_{5i} = D_{5i}$	E_{5i}	$h_{5i\ell}$	R_ℓ	Task	$T_{5i} = D_{5i}$	E_{5i}	$h_{5i\ell}$	R_ℓ
τ_{51}	150	2	-	-	τ_{54}	600	10	-	-
τ_{52}	160	1	-	-	τ_{55}	650	50	5	R_1
τ_{53}	500	35	10	R_1	τ_{56}	750	8	4	R_2

Table 4
Partial interface candidates of component C_5 in $\mathcal{IC}_5^{(1)}$ (left) and in $\mathcal{IC}_5^{(2)}$ (right), where the non-redundant candidates are highlighted by $*$.

	iteration	Q_{51}	X_{51}	rc_{51}		**iteration**	Q_{52}	X_{52}	rc_{52}
$*$	1	51	13	π_{53}		1	51	102	π_{56}
$*$	2	52.5	12	π_{52}		2	51	52	π_{55}
$*$	3	56	10	π_{51}		3	51	42	π_{54}
						4	51	7	π_{53}
					$*$	5	51	6	π_{52}
					$*$	6	53	4	π_{51}

Example 6: Consider two components C_2 and C_5. The partial interface candidates in Example 5 are computed in Table 4. Component C_2 (taken from Example 2) has a period $P_2 = 10$ and it has just a single task $\tau_{21} = (27, 5, 27, \{0.5\})$

accessing resource R_2. The corresponding interface, satisfying the periodic resource model, is $\Gamma_2 = (10, \frac{8}{3}, \{0.5\})$, i.e., component C_2 has just one partial interface candidate.

Resource R_1 is accessed only by component C_5 and resource R_2 is shared by component C_2 and C_5. Note that the smallest resource holding time of component C_5 to resource R_1 is 10 time units, so that sharing of resource R_1 would immediately make the system infeasible. Since both Shin et al. (2008) and Behnam et al. (2010b) assume that all the resources accessed by a component are globally shared, their interface selection algorithms deem the system unschedulable. Contrary, our selection algorithm (Algorithm 4) will return a feasible set of interface candidates.

Let us assume a global EDF scheduler and ONP for the global arbitration of resource R_2. During the initialization phase of Algorithm 4, Lemma 6 labels the partial interface candidate $\Gamma_{52} = (125, 51, \{6\})$ redundant, so that C_5 has one partial interface candidate remaining, i.e., the 6-th row in Table 4. Each of the components has therefore just a single partial interface candidate. For each of the components, their partial candidate directly transforms into an interface and it determines the local resource ceilings. That is, Algorithm 4 returns the set $\{\Gamma_2 = (10,\ \frac{8}{3},\ \{0.5\}),\ \Gamma_5 = (125,\ 53,\ \{4\})\}$ and the corresponding system load is 0.773.

Example 7: Consider a system with 3 components: C_2, C_5 and C_7. The components C_2 and C_5 are the same as in Example 6 and they share resource R_2. Component C_7 has a period $P_7 = 80$ and it shares resource R_1 with component C_5. Moreover, it contains two fixed-priority-scheduled tasks: $\tau_{71} = (1000, 2, 1000, \{0.5\})$ and $\tau_{72} = (1000, 1, 1000, \emptyset)$. Task τ_{71} accesses a global resource R_1 for a duration of $h_{711} = X_{71} = 0.5$ time units; task τ_{72} is independent. Using the local schedulability condition in (27) and using the periodic resource model, we derive a single partial interface candidate, i.e., directly resulting into an interface $\Gamma_7 = (80,\ \frac{3}{11},\ \{0.5\})$.

Let us assume a global EDF scheduler and ONP for the global arbitration of the resources R_1 and R_2. The initialization of Algorithm 4 selects the following interface candidates:

$$\left\{\Gamma_2 = (10, \frac{8}{3}, \{0.5\}),\ \Gamma_5 = (125, 53, \{12, 4\}),\ \Gamma_7 = (80, \frac{3}{11}, \{0.5\})\right\}. \tag{40}$$

Note that the resource holding time $X_{51} = 12$ (rather than $X_{51} = 13$). This is the consequence of labeling the partial interface candidate $\Gamma_{52} = (125, 51, \{6\})$ redundant, so that the other local resource ceiling rc_{51} can be increased to π_{52} without further increasing the budget (see Table 4).

By computing the system load, we obtain $\text{load}_{\texttt{sys}} = \alpha_5 \approx 0.85$. Since C_5 experiences no blocking – i.e., $\forall t \geq 125 \ :: \ B(t) = 0$ –, we obtain $\mathcal{B} = \emptyset$, so that there is no chance to further decrease the system load. Hence, (40) contains the optimal set of interfaces.

9 Evaluation

This section evaluates analysis methods for global resource sharing. From the results, we derive which method matches the best with given system characteristics.

In our experiments, we choose a system utilization U and we generate individual component utilizations $U(\mathcal{T})$ using the UUnifast algorithm by Bini and Buttazzo (2004). The period of a component is uniformly drawn from the interval $[40, 70]$. We assume global EDF scheduling of components and a single non-preemptively shared global resource by all components.

Given a cumulative component utilization $U(\mathcal{T})$, we generate $n_s = 8$ tasks for each component. The task periods T_{si} are uniformly drawn from the interval $[140, 1000]$. We initially assume deadlines equal to periods, i.e., $T_{si} = D_{si}$ and we assign deadline-monotonic fixed priorities to tasks. The individual task utilizations u_{si} are generated using the UUnifast algorithm by Bini and Buttazzo (2004). Using the task's utilization u_{si} and the randomly generated period T_{si}, we can derive the worst-case execution time E_{si} of a task τ_{si}, i.e., $E_{si} = u_{si} \times T_{si}$. All tasks access a single global resource for a random duration between $0.1 \times E_{si}$ and $0.25 \times E_{si}$. In each experimental setting a new set of 10,000 systems is generated.

9.1 Feasibility of task sets in the presence of global resources

We first investigate for which task-characteristics a particular analysis method is better at the component level. We look at the percentage of schedulable task sets, generated according to the description above, while requiring $Q_s + O_s(P_s) \leq P_s$.

In each simulation study a new set of 10,000 systems is generated and the following settings are changed:

1) *Component utilization:* The utilization of a component $U(\mathcal{T})$ is varied within a range of $[0.05, 1.0]$ using incremental steps of 0.05, see Figure 8.
2) *Component periods:* The period of the periodic resource P_s is varied within a range of $[5, 70]$ with incremental steps of 5, see Figure 9.

For comparison purposes we included the results for the improved local analysis of ONP by Behnam et al. (2011), i.e., IONP. Both experiments show that the different overrun methods have little impact on the local schedulability of a task set on a periodic resource. The main reason for this is the constraint that the calculated budget Q_s and the overrun budget X_s have to fit within period P_s, i.e., we applied the constraint $Q_s + X_s \leq P_s$. For SIRAP and BROE, we require $X_s \leq Q_s$. Due to this constraint, both SIRAP's and BROE's performance

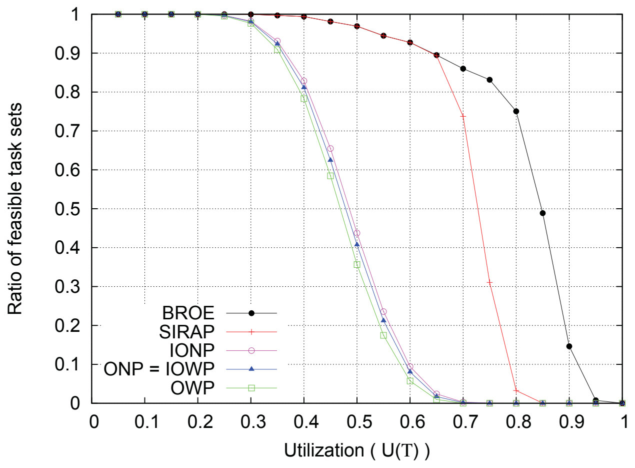

Figure 8. Ratio of schedulable task sets versus the utilization, where the component period is $P_s = 40$ and the number of tasks is $n_s = 8$.

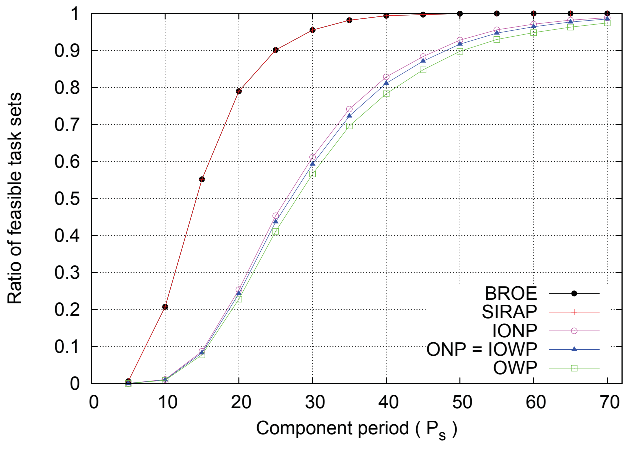

Figure 9. Ratio of feasible task sets as a function of the component period, where the number of tasks is $n_s = 8$ and the utilization $U(\mathcal{T}) = 0.4$.

are suppressed and overlapping for small resource periods (see Figure 9). BROE may require a larger budget for a component, because it must use the bounded-delay model. In terms of the schedulability ratio, however, BROE clearly outperforms the other protocols (see Figure 8). In addition, both figures show the cost of an opaque analysis in the context of two-level FPPS-based HSFs, in which BROE is inapplicable.

The constraint, $Q_s + X_s \leq P_s$, is the main weakness for all overrun variants, determined by the ratio $\frac{X_s}{P_s}$. This ratio can be increased by increasing the utilization (Figure 8), choosing smaller resource periods (Figure 9), decreasing the number of tasks (n_s) or by increasing the range of the task periods. When keeping the utilization $U(\mathcal{T})$ constant, the last two alternatives result in larger WCETs and resource access times. Since X_s is computed from a fixed fraction of the tasks' execution times, this increases the $\frac{X_s}{P_s}$ ratio.

Our improved OWP (IOWP) performs equally well as ONP at the local level and its global schedulability is superior compared to ONP. OWP is therefore preferred above ONP. Note that the non-opaque IONP analysis in Behnam et al. (2011) may slightly improve on IOWP and ONP. However, the global analysis for OWP is always better than or equal to the global analysis of ONP. This gives an advantage to ONP when both integration tests instantiated from (19) yield the same result, i.e., when all *component periods* are chosen approximately the same, so that also OWP accounts for an overrun in each component period.

Finally, we note that the choice for local FPPS versus local EDF does not influence the relative strength of the global synchronization protocols. The reason is that the ratio of $\frac{X_s}{P_s}$ - determining the performance for overrun-based protocols - is only dependent on task WCETs and the component period. The same holds for SIRAP's and BROE's requirement $X_s \leq Q_s$. In general, the priority assignment to tasks may change the value of the computed resource holding times and, thus, it may influence the value of X_s. Since in our system model P_s is smaller than any of the local task periods, however, the way of computing resource holding times is the same for EDF and for deadline-monotonic fixed priorities. As a nice property of opacity, we have obtained independence of the overheads of global synchronization and the local scheduling policy.

9.2 Global scheduling penalties for global synchronization

In this section, we compare the analyses at the compositional level, because at the local level - especially with opaque analyses - the resource-supply model may hide scheduling penalties.

We observed that BROE is superior in terms of the number of task sets that can be accommodated, because BROE does not need additional overrun budget and it does not insert idle time. However, these results from Section 9.1 ignore the required processor bandwidth by a single component. The bounded-delay model, exclusively applied to BROE, performs relatively poorly compared to the periodic resource model when the utilization of a component $U(\mathcal{T})$

is small. Components having such characteristics contain tasks with small computation times relative to task periods, so that the stair-cased curve of the periodic resource supply gives a relative high provisioning with respect to the WCET of tasks. A solution would be to reduce the period P_s of a component. Although this makes the linear approximation of the periodic supply tighter (being advantageous for the bounded-delay model and BROE), the period size cannot be decreased arbitrarily, because an entire critical section must fit within one period. Moreover, the shorter the component period is, the higher context switching overhead will be. The implementation overhead of the synchronization protocols is ignored in this evaluation, however, and it is different for each protocol, for example, see van den Heuvel et al. (2012).

In the first experiment, we investigate how composing multiple resource-sharing components affects the number of schedulable systems. Figure 10 shows the results for $N = 2$ components and Figure 11 for $N = 5$ components. When the utilization of a single component is relatively large, i.e., $U(\mathcal{T}) \gtrsim 0.1$, BROE clearly outperforms all other protocols, see Figure 10. For smaller utilizations, SIRAP becomes more advantageous than BROE, see Figure 11. The different overrun methods have little impact on the global schedulability of components. The main reason for this is the local constraint that the calculated budget Q_s and the overrun budget X_s already have to fit within period P_s.

In the second experiment, we repeated the same experiment for $N = 5$ components and we randomly generated tasks with deadlines $D_{si} \leq T_{si}$, uniformly drawn from the range $[E_{si} + 0.5(T_{si} - E_{si});\ T_{si}]$. Figure 12 reports the results. Compared to the first experiment, the bounded-delay model further reduces the performance of BROE. Intuitively, postponing budget supply to a task set, being subject to tight deadline constraints, deflates BROE's performance compared to the non-opaque analysis of SIRAP. However, BROE's performance is considerably better than any overrun variant.

In the third experiment, the system utilization $U = 0.5$ and the range of task deadlines $D_{si} \leq T_{si}$ are fixed. The number of components, N, is varied within a range of $[1, 14]$, see Figure 13. Composing a system of many resource-sharing components, e.g., the operating system itself can be a single point of synchronization, may significantly decrease the number of schedulable systems. It is interesting to see that BROE covers the entire performance spectrum compared to SIRAP, ONP and OWP: from a superior performance for components with large individual utilizations, to an inferior performance for small component utilizations.

In the fourth experiment, we keep a system utilization of $U = 0.5$ for $N = 5$ components and we vary the range of the task deadlines using parameter δ. We generated task deadlines uniformly drawn from the range $[E_{si}+\delta(T_{si}-E_{si});\ T_{si}]$. A low value of δ allows tasks to have short deadlines relative to their execution

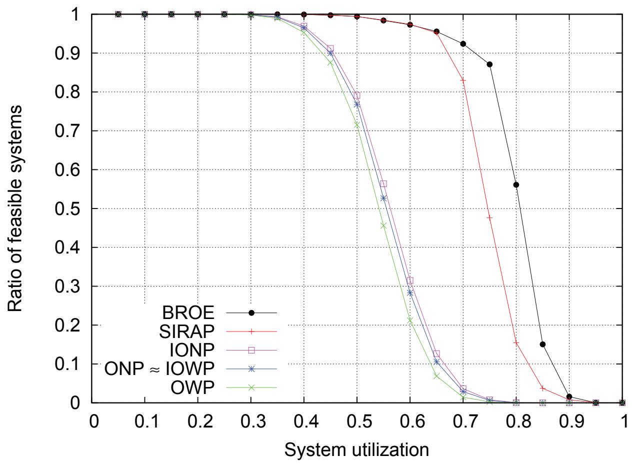

Figure 10. Ratio of schedulable systems versus the system utilization, where the number of components is $N = 2$ and all tasks have $D_{si} = T_{si}$.

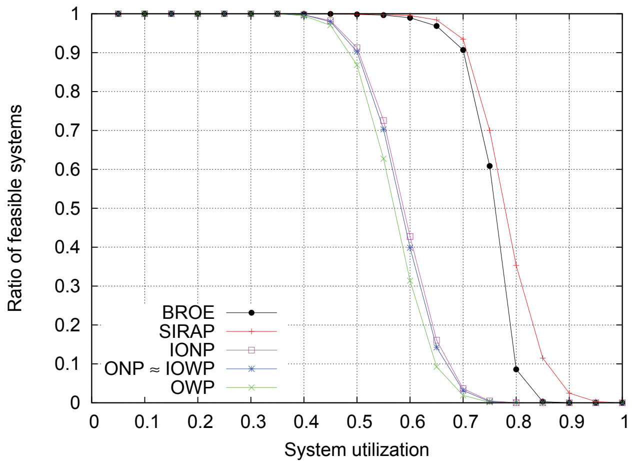

Figure 11. Ratio of schedulable systems versus the system utilization, where the number of components is $N = 5$ and all tasks have $D_{si} = T_{si}$.

time and $\delta = 1$ means that deadlines are equal to periods. Figure 14 shows the results. This experiment confirms the previous experiments: SIRAP's non-opaque analysis is beneficial for deadline-constrained tasks, while HSRP's overrun and BROE perform equally poor under tight deadline constraints.

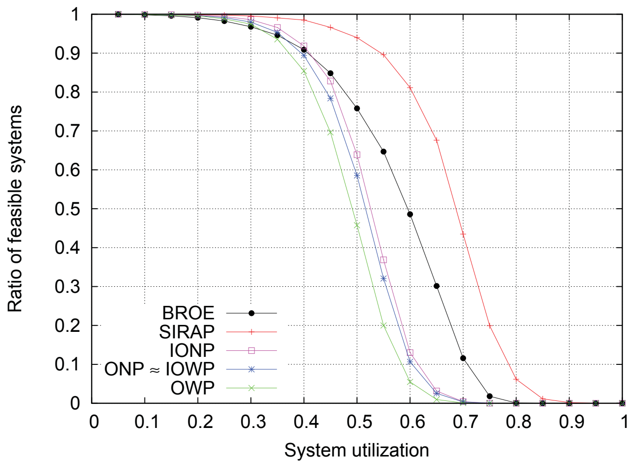

Figure 12. Ratio of schedulable systems versus the system utilization, where the number of components is $N = 5$ and tasks may have $D_{si} \leq T_{si}$.

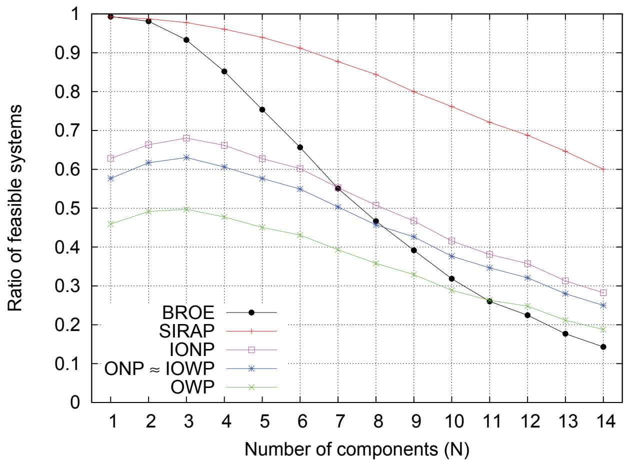

Figure 13. Ratio of schedulable systems versus the number of components, where the system utilization is $U = 0.5$ and tasks may have $D_{si} \leq T_{si}$.

Finally, both SIRAP and BROE have shown to be more resilient than HSRP's overrun variants for relatively large critical section lengths compared to a component's budget. For SIRAP, the analysis for a single shared resource by

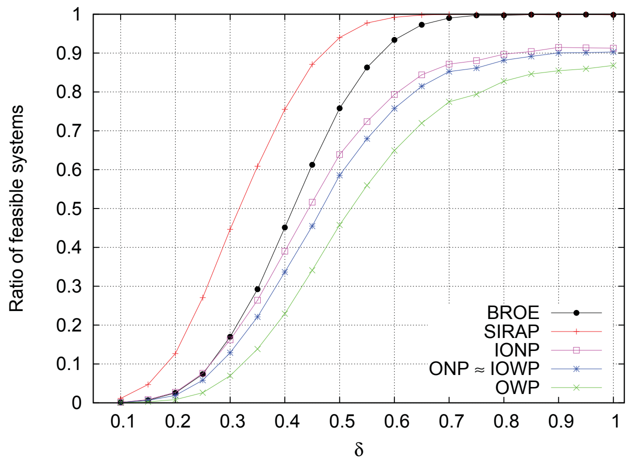

Figure 14. Ratio of schedulable systems versus the deadline distribution of tasks, where the number of components is $N = 5$ and the system utilization is $U = 0.5$.

each task performs relatively poorly, because the more individual resource accesses are considered, the better its analysis. BROE and overrun have opaque analysis, i.e., only based on the local SRP. Sharing more global resources would therefore not affect much the performance of an opaque analysis, because the budget parameters are by definition of opacity independently derived of the values of $X_{s\ell}$. A non-opaque local analysis, however, might be able to further reduce the estimated protocol-specific scheduling penalties.

9.3 Recommendations

In HSFs where the (global) scheduling policy and the (global) resource-arbitration policy are part of the implementation of a black-box component, an opaque analysis is the only way to integrate components that share global resources. Based on our evaluation of the existing SRP-based global synchronization protocols, we showed that

- BROE's analysis is opaque and, in many situations, it performs superior or very competitive compared to any other protocol (ONP, OWP and SIRAP).
- BROE's performance significantly degrades, if a system is composed of components with tight internal task deadlines or if it is composed of many components having small utilizations.
- Our improved OWP (IOWP) analysis shows the most consistent performance of the existing opaque analyses, i.e., its performance is less sensitive

for local task deadlines than BROE and it allocates less or an equal amount of overrun budget to components than ONP.
- If the periods of integrated components are chosen almost the same, then the amount of allocated overrun budgets to components is the same for ONP and IOWP.

Suppose that during the local analysis of a component we are given which of the global resources are actually shared with other components in the system. In these situations a non-opaque analysis may better estimate the cost of global resource sharing at the local level than a non-opaque analysis. Considering both opaque and non-opaque analysis:

- Only when all component periods are almost the same, a non-opaque ONP may take advantage over OWP in terms of schedulability of a system.
- We have proven the enhanced overrun by Behnam et al. (2010c) superfluous, i.e., it is outperformed by our improved OWP;
- SIRAP's analysis consistently outperforms ONP and OWP.
- In many cases SIRAP is performing equally well as, or better than, BROE.
- If local deadline constraints are loose, then BROE provides a considerably easier analysis than SIRAP without (significant) performance loss.

Although a non-opaque analysis might be able to improve the schedulability of a system, the problem of a non-opaque analysis is that it might be computationally intractable to find the optimal settings of the local resource ceilings with respect to the smallest possible system load. We have presented algorithms to optimize the system load by means of an opaque analysis within a polynomial number of iterations in the size of the system. Only with an opaque analysis these algorithms are guaranteed to find a solution.

We therefore believe an opaque analysis enables an incremental way of synthesizing a system. Firstly, an opaque analysis abstracts from the actual dependencies between components until component integration time, so that during integration the system load can be minimized by considering just the actual dependencies. Secondly, given that a component has known (*i*) timing details of its local tasks and (*ii*) pre-computed resource ceilings from the first step, then the system load might be further improved by re-analyzing the individual components in a composed system with a non-opaque analysis. This incremental way of system analysis makes it possible to share resources between tasks in arbitrary components, while being able to reduce the scheduling penalties of global resource arbitration by means of computationally tractable algorithms.

10 Conclusion

This paper introduced the notion of *opaque* analysis for resource-sharing components that need to be integrated on a uni-processor platform. An opaque local analysis of a component abstracts from global resource sharing until component integration. Sufficient conditions for opacity are:

- component periods are smaller than the local tasks' periods, so that resource holding times of a component are defined independently of the global synchronization protocol;
- resource holding times must disappear from the local schedulability test, so that the budget parameter of a component can be solely computed in order to meet deadline constraints of tasks (and independently of the global synchronization protocol).

As a result of both conditions, when the SRP arbitrates access to shared resources between periodic components, the necessary condition of opacity is satisfied: all interface parameters of a component are computed independently of a global synchronization protocol. By strictly separating local and global scheduling, including resource arbitration, an opaque analysis enables a computationally tractable design-space exploration - which cannot be guaranteed with a non-opaque analysis - for optimizing the required processor resources of a system as a whole (i.e., for optimizing the system load).

We applied opacity to four existing global synchronization protocols: SIRAP, ONP, OWP and BROE. Although SIRAP's original analysis is non-opaque, we can use the analysis of overrun without payback (ONP) as a conservative and opaque alternative. We also presented an opaque analysis for overrun with payback (OWP), which dominates the opaque ONP.

In a system with global FPPS of components, opaque analyses (ONP and our new OWP) have shown a significant drop in the number of schedulable systems. The performance of opaque analyses is much better under global EDF of components. The reason is that BROE's analysis is opaque and, in many situations, it is competitive with SIRAP's non-opaque analysis. Although BROE does not require any allocated processor time for idle time (like SIRAP does) or overruns (like HSRP does), we identified that BROE's analysis may require more processor bandwidth than any of the other protocols if a system is composed of components with tight deadlines or if it is composed of many components having small utilizations. For those systems, a non-opaque analysis may therefore significantly improve schedulability (demonstrated by SIRAP), regardless of the global and local scheduling policies.

To explore the impact of different settings of resource-sharing components on the system load, an opaque analysis is extremely useful. A non-opaque analysis may further tighten the results of an opaque analysis, thereby enabling an incremental system analysis. Since an opaque analysis of a component only looks at local resource sharing, we conjecture an opaque analysis is also important in multi-level HSFs (with more than two scheduling levels), so that a component can be reused at an arbitrary position in the scheduling tree. We leave the integration of resource-sharing components in multi-level HSFs as future work.

References

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

Almeida L, Peidreiras P (2004) Scheduling with temporal partitions: response-time analysis and server design. In: Conference on Embedded Software (EMSOFT), pp 95–103

Baker T (1991) Stack-based scheduling of realtime processes. Real-Time Systems 3(1):67–99

Balbastre P, Ripoll I, Crespo A (2009) Exact response time analysis of hierarchical fixed-priority scheduling. In: Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 315–320

Baruah SK (2006) Resource sharing in EDF-scheduled systems: A closer look. In: Real-Time Systems Symposium (RTSS), pp 379–387

Behnam M, Shin I, Nolte T, Nolin M (2007) SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In: Conference on Embedded Software (EMSOFT), pp 279–288

Behnam M, Nolte T, Åsberg M, Bril RJ (2009a) Overrun and skipping in hierarchically scheduled real-time systems. In: Conference on Embedded Real-Time Computing Systems and Applications (RTCSA), pp 519–526

Behnam M, Nolte T, Bril RJ (2009b) Refining SIRAP with a dedicated resource ceiling for self-blocking. In: Conference on Embedded Software (EMSOFT), pp 157–166

Behnam M, Nolte T, Bril RJ (2010a) Bounding the number of self-blocking occurrences of SIRAP. In: Real-Time Systems Symposium (RTSS), pp 61–72

Behnam M, Nolte T, Fisher N (2010b) On optimal real-time subsystem-interface generation in the presence of shared resources. In: Conference on Emerging Technologies and Factory Automation (ETFA)

Behnam M, Nolte T, Sjodin M, Shin I (2010c) Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems. IEEE Transactions on Industrial Informatics (TII) 6(1):93–104

Behnam M, Nolte T, Bril RJ (2011) Tighter schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks. In: International Conference on Engineering of Complex Computer Systems (ICECCS), pp 35–44

Bertogna M, Fisher N, Baruah S (2007) Static-priority scheduling and resource hold times. In: Parallel and Distributed Processing Symposium (IPDPS)

Bertogna M, Fisher N, Baruah S (2009a) Resource holding times: computation and optimization. Real-Time Systems 41(2):87–117

Bertogna M, Fisher N, Baruah S (2009b) Resource-sharing servers for open environments. IEEE Transactions on Industrial Informatics (TII) 5(3):202–219

Bini E, Buttazzo G (2004) Biasing effects in schedulability measures. In: Euromicro Conference on Real-Time Systems (ECRTS), pp 196–203

Bril RJ, Verhaegh WFJ, Wüst CC (2006) A cognac-glass algorithm for conditionally guaranteed budgets. In: Real-Time Systems Symposium (RTSS), pp 388–397

Buttazzo G (2005) Hard real-time computing systems - predictable scheduling algorithms and applications (2^{nd} edition). Springer

Caccamo M, Sha L (2001) Aperiodic servers with resource constraints. In: Real-Time Systems Symposium (RTSS), pp 161–170

Davis R, Burns A (2005) Hierarchical fixed priority pre-emptive scheduling. In: Real-Time Systems Symposium (RTSS), pp 389–398

Davis R, Burns A (2006) Resource sharing in hierarchical fixed priority pre-emptive systems. In: Real-Time Systems Symposium (RTSS), pp 257–267

Deng Z, Liu JS (1997) Scheduling real-time applications in open environment. In: Real-Time Systems Symposium (RTSS), pp 308–319

Easwaran A, Anand M, Lee I (2007) Compositional analysis framework using EDP resource models. In: Real-Time Systems Symposium (RTSS), pp 129–138

Feng X, Mok A (2002) A model of hierarchical real-time virtual resources. In: Real-Time Systems Symposium (RTSS), pp 26–35

Fisher N, Dewan F (2012) A bandwidth allocation scheme for compositional real-time systems with periodic resources. Real-Time Systems 48(3):223–263

Ghazalie TM, Baker TP (1995) Aperiodic servers in a deadline scheduling environment. Real-time Systems 9(1):31–67

van den Heuvel MMHP, Bril RJ, Lukkien JJ (2011) Dependable resource sharing for compositional real-time systems. In: Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 153–163

van den Heuvel MMHP, Bril RJ, Lukkien JJ (2012) Transparent synchronization protocols for compositional real-time systems. IEEE Transactions on Industrial Informatics (TII) 8(2):322–336

Holman P, Anderson J (2002) Locking in pfair-scheduled multiprocessor systems. In: Real-Time Systems Symposium (RTSS), pp 149–158

Kumar P, Chen JJ, Thiele L, Schranzhofer A, Buttazzo G (2011) Real-time analysis of servers for general job arrivals. In: Conference on Embedded Real-Time Computing Systems and Applications (RTCSA), pp 251–258

Kuo TW, Li CH (1999) A fixed-priority-driven open environment for real-time applications. In: Real-Time Systems Symposium (RTSS), pp 256–267

Lehoczky JP, Sha L, Ding Y (1989) The rate monotonic scheduling algorithm:

Exact characterization and average case behavior. In: Real-Time Systems Symposium (RTSS), pp 166–171

Lipari G, Baruah S (2000) Efficient scheduling of real-time multi-task applications in dynamic systems. In: Real-Time Technology and Applications Symposium (RTAS), pp 166–175

Lipari G, Bini E (2005) A methodology for designing hierarchical scheduling systems. Journal of Embedded Computing (JEC) 1(2):257–269

Lipari G, Lamastra G, Abeni L (2004) Task synchronization in reservation-based real-time systems. IEEE Transactions on Computers (TC) 53(12):1591–1601

Liu C, Layland J (1973) Scheduling algorithms for multiprogramming in a real-time environment. Journal of the ACM 20(1):46–61

López Martinez P, Barros L, Drake J (2010) Scheduling configuration of real-time component-based applications. In: Reliable Software Technology - Ada-Europe, Lecture Notes in Computer Science (LNCS), vol 6106, Springer, pp 181–195

Mercer C, Savage S, Tokuda H (1994) Processor capability reserves: Operating system support for multimedia applications. In: International Conf. on Multimedia Computing and Systems (ICMCS), pp 90–99

de Niz D, Abeni L, Saewong S, Rajkumar R (2001) Resource sharing in reservation-based systems. In: Real-Time Systems Symposium (RTSS), pp 171–180

Rajkumar R, Juvva K, Molano A, Oikawa S (1998) Resource kernels: A resource-centric approach to real-time and multimedia systems. In: SPIE/ACM Conference on Multimedia Computing and Networking (CMCN), pp 150–164

Sha L, Rajkumar R, Lehoczky J (1990) Priority inheritance protocols: an approach to real-time synchronisation. IEEE Transactions on Computers (TC) 39(9):1175–1185

Shin I, Lee I (2004) Compositional real-time scheduling framework. In: Real-Time Systems Symposium (RTSS), pp 57–67

Shin I, Lee I (2008) Compositional real-time scheduling framework with periodic model. ACM Transactions on Embedded Computing Systems (TECS) 7(3):1–39

Shin I, Behnam M, Nolte T, Nolin M (2008) Synthesis of optimal interfaces for hierarchical scheduling with resources. In: Real-Time Systems Symposium (RTSS), pp 209–220

Steinberg U, Wolter J, Härtig H (2005) Fast component interaction for real-time systems. In: Euromicro Conference on Real-Time Systems (ECRTS), pp 89–97

Wandeler E, Thiele L (2005) Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In: Conference on Embedded Software (EMSOFT), pp 80–89

PAPER D:

Transparent synchronization protocols for compositional real-time systems

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien

In IEEE Transactions on Industrial Informatics (TII),
pp. 322–336, vol. 8, issue 2, May 2012

Abstract

Hierarchical scheduling frameworks (HSFs) provide means for composing complex real-time systems from well-defined, independently analyzed components. To support resource sharing in two-level HSFs, three synchronization protocols based on the stack resource policy (SRP) have recently been presented for single-processor execution platforms, i.e., HSRP, SIRAP and BROE. This paper presents a transparent implementation of these three protocols side-by-side in an HSF-enabled real-time operating system. Transparent synchronization interfaces make it possible to select a protocol during integration time based on its relative strengths.

A timing interface describes the required budget to execute a component on a shared platform and an accessor's maximum critical-section execution time to global shared resources. These resources are arbitrated based on the available budget of the accessing task. We enable this explicit synchronization of virtual time with global time by means of a novel virtual-timer mechanism. Moreover, we investigate system overheads caused by each synchronization protocol, so that these can be included in the system analysis. Based on the analytical and implementation overheads of each protocol, we present guidelines for the selection of a synchronization protocol during system integration.

Finally, we show that unknown task-arrival times considerably complicate an efficient implementation of SIRAP's self-suspension mechanism. We briefly discuss the implementation complexity caused by these arrivals for bandwidth-preserving servers, e.g., deferrable servers and BROE.

1 Introduction

Many real-time embedded software applications are becoming increasingly complex and diverse, while their time to market and cost is continuously under pressure. By using dedicated component-based architectures, manufacturers aim to reuse hardware and software components and support product families. The automotive industry, for example, initiated several standardized development models, e.g., the AUtomotive Open System ARchitecture (AUTOSAR) [4]. This industrial standard specifies that an underlying OSEK-based [5] operating system should prevent timing faults in any component to propagate to different components on the same processor.

Hierarchical scheduling frameworks (HSFs) have been investigated to provide such temporal isolation [6] by allocating a *budget* to each component, where budgets are implemented by means of servers. Moreover, HSFs provide a paradigm for facilitating a decoupling [7] of (*i*) development and analysis of individual components and (*ii*) integration of components on a shared processor, including analysis at the system level. In such *compositional systems*, a component that is validated to meet its timing constraints when executing in isolation will continue meeting its timing constraints after integration or admission on a shared uni-processor platform.

To accommodate further resource sharing between components, three synchronization protocols have been proposed based on the *stack resource policy* (SRP) [8], i.e., hierarchical stack resource policy (HSRP) [1], subsystem integration and resource allocation policy (SIRAP) [2] and bounded-delay resource open environment (BROE) [3]. An HSF extended with such a protocol makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary components, in a mutually exclusive manner. A resource that is used in more than one component is denoted as a *global shared resource*. A resource that is only shared within a single component is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of the corresponding budget, excessive blocking periods can occur which may hamper the correct timeliness of other components [9].

HSRP, SIRAP and BROE each provide a different run-time mechanism to prevent the depletion of a component's budget during global resource access. Each protocol has its relative strengths depending on system characteristics [10], [11]. We would therefore like to enable these three protocols side-by-side within the same HSF to minimize the calculated resource needs of a system. This requires *protocol transparency*, i.e., an application programmer (*i*) can *ignore* which synchronization protocol is selected by the system, and (*ii*) cannot *exploit* the knowledge of the selected protocol. Our implementations of these SRP-based synchronization protocols are based on the μC/OS-II operating system [12], which we have extended with proprietary support for two-level hierarchical scheduling. The choice of operating system is driven by its (former) OSEK compatibility[1].

Although components are designed with the intention to be independent, it may practically be necessary to synchronize on logical resources such as operating-system primitives, shared communication devices and other memory-mapped peripherals [1], [6]. Global resource sharing protocols within HSFs are extensively investigated for ideal system models, e.g., [1], [2], [6], [3]. Unfortunately, most off-the-shelf real-time operating systems and middleware, including μC/OS-II, by default do not support hierarchical scheduling nor SRP-based synchronization. Moreover, the run-time overhead of these protocols hardly received any attention. These overheads become relevant during deployment of an HSF with resource-sharing components.

Contributions

The contribution of this paper is six fold.

- We present a *transparent* implementation of HSRP [1], SIRAP [2] and BROE [3] to support global resource sharing in two-level HSFs. We restrict this implementation to single unit resources and single processor platforms. This updates our first implementation in [13].
- We extend our time-management module for HSFs with a novel virtual-timer mechanism to arbitrate global shared resources based on the accessing task's available budget.
- We extend [13] with a programming model that makes it possible to transparently choose a synchronization protocol at different abstraction levels, i.e., per component, per task or per resource. We accordingly show how a wide range of existing analysis for different server implementations complemented with HSRP, SIRAP and BROE can be integrated in a HSF.
- Contrary to [13], we also investigate the overheads of HSRP, SIRAP and BROE for tasks with unknown arrival times, e.g., sporadic tasks, serviced by bandwidth-preserving servers. We show that these servers combined with SIRAP lead to high implementation costs. BROE includes its own bandwidth-preserving server and does not suffer these implementation penalties.
- We compare overheads and interrupt latencies caused by each resource-sharing protocol, because in many microkernels, including μC/OS-II, the only way for tasks to share data structures with interrupt service routines (ISRs) is by means of disabling interrupts. Furthermore, we evaluate our modifications to μC/OS-II.

1. Unfortunately, the supplier of μC/OS-II, Micrium, has discontinued the support for the OSEK-compatibility layer.

- Based on the implementation and analytical complexity of each synchronization protocol, we extract guidelines to select a protocol during system integration.

The paper is organized as follows. Section 2 describes related works. Section 3 presents our system model. Section 4 presents our programming model to support transparent resource sharing. Section 5 summarizes our HSF extensions for μC/OS-II. Section 6 presents our implementations for SRP-based resource arbitration, including HSRP, SIRAP and BROE. In Section 7 we experimentally compare these protocol implementations. Section 8 evaluates our implementations. Finally, Section 9 concludes this paper.

2 Related work

In hierarchically scheduled systems a group of tasks, forming a component, is mapped on a reservation [7]. These tasks may share resources with other tasks, either within the same component or located in other components. This means that resource sharing expands across reservations which calls for specialized resource access protocols. We first give an overview of existing resource-sharing protocols. Next, we present interface abstraction techniques for components.

2.1 Task Synchronization in Reservation-based Systems

In literature several alternatives are presented to accommodate resource sharing between tasks in reservation-based systems. De Niz et al. [14] support this in their fixed-priority preemptively scheduled (FPPS) Linux/RK resource kernel based on the immediate priority ceiling protocol (IPCP) [15] and propose a mechanism for temporal protection against misbehaving critical sections. Steinberg et al. [16] implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach is described in [17] for earliest-deadline-first (EDF)-based systems and termed bandwidth-inheritance (BWI). BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol [15], i.e., when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. BWI does not require a-priori knowledge of tasks, i.e., no ceilings need to be precalculated. Buttazzo and Gai [18] implemented a reservation-based EDF scheduler for the real-time ERIKA Enterprise kernel, including SRP-based synchronization support. All these approaches assume a one-to-one mapping from tasks to budgets, and inherently only have a single scheduling level.

A prerequisite to enable independent analysis of interacting components and their integration is the knowledge of which resources a task will access [7].

When a task accesses a global shared resource, one needs to consider the priority inversion between components as well as local priority inversion between tasks within the component. To accommodate such resource sharing, three synchronization protocols have been proposed based on SRP [8], i.e., HSRP [1], SIRAP [2] and BROE [3]. Although HSRP [1] originally does not integrate in HSFs due to the lacking support for independent analysis of components, Behnam et al. [19] lifted this limitation. BROE [3] only considers resource sharing under global EDF scheduling. Recently, HSRP and SIRAP have been implemented in the two-level FPPS-based HSF on top of VxWorks [20]. We presented similar implementations in [21] for HSRP and SIRAP within the real-time microkernel μC/OS-II, and included BROE and its required EDF support [13].

2.2 Component Abstraction

In [21], [22] we have extended μC/OS-II with support for two-level FPPS with idling periodic servers [23], polling servers [24] and bandwidth-preserving, deferrable servers [25]. Most works implement an idling periodic server [23] as a budget provider, e.g., see [20], [21], because it behaves as an ideal periodic task and it is easy to implement. Davis and Burns [23] claim that there are no components comprising a task set that can be scheduled using a deferrable server and cannot be scheduled using an equivalent idling periodic server with the same period and capacity. However, [26] shows that their claim relies on a specific task-set construction. We therefore investigate the efficiency of HSRP and SIRAP combined with deferrable servers to enhance average response times to external events.

Lipari and Bini [27] presented a method to calculate budget parameters for independent components. In [28], [29] techniques are proposed to obtain similar parameters in the presence of shared resources with heuristics to optimize the component's calculated resource needs. Recently, HSRP and SIRAP were compared analytically with respect to their impact on the total system load for various component parameters [11]. A preliminary comparison including BROE has been presented in [10]. These protocols have relative strengths depending on chosen parameters. Supporting these three synchronization protocols in the same HSF puts demands on the implementation and the schedulability analysis. Although we provide a clean *timing interface* to separate component internals from their global timing parameters, we consider tools and methods to obtain these parameters outside the scope of this paper. We focus on analytical and programming aspects of HSRP, SIRAP and BROE and compare the efficiency of corresponding primitives for different server models.

3 Real-time scheduling model

We consider a two-level HSF using the periodic resource model [7] to specify guaranteed processor allocations to components. An SRP-based synchronization protocol is used for mutually exclusive access to global resources.

3.1 System model

A system contains a set $\mathcal{R}$ of M global logical resources R_1, R_2, …, R_M, a set $\mathcal{C}$ of N components C_1, C_2, …, C_N, a set $\mathcal{B}$ of N budgets for which we assume a periodic resource model [7], and a single processor. Each component C_s has a dedicated budget which specifies its periodically guaranteed fraction of the processor. In the remainder of this paper, we leave budgets implicit, i.e., the timing characteristics of budgets are taken care of in the description of components. Components are scheduled by means of FPPS or EDF.

3.2 Component and Task model

Each component C_s contains a set $\mathcal{T}_s$ of n_s sporadic tasks τ_{s1}, τ_{s2}, …, τ_{sn_s}. The set $\mathcal{R}_s$ denotes the subset of M_s global resources accessed by component C_s. The maximum time that a component C_s executes while accessing resource $R_l \in \mathcal{R}_s$ is denoted by X_{sl}, where $X_{sl} \in \mathbb{R}^+ \cup \{0\}$ and $X_{sl} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$. The timing characteristics of C_s are specified by means of a triple $< P_s, Q_s, \mathcal{X}_s >$, where $P_s \in \mathbb{R}^+$ denotes its period, $Q_s \in \mathbb{R}^+$ its budget, and $\mathcal{X}_s$ the set of maximum access times to global resources. The maximum value in $\mathcal{X}_s$ is denoted by X_s, where $0 < X_s \leq Q_s \leq P_s$.

Timing characteristics of a task $\tau_{si} \in \mathcal{T}_s$ are specified by means of a triple $< T_{si}, E_{si}, D_{si} >$, where $T_{si} \in \mathbb{R}^+$ denotes its minimum inter-arrival time, $E_{si} \in \mathbb{R}^+$ its worst-case computation time, $D_{si} \in \mathbb{R}^+$ its (relative) deadline, where $0 < E_{si} \leq D_{si} \leq T_{si}$. For notational convenience we assume that tasks (and components) are given in deadline-monotonic order, i.e., τ_{sn_s} has the largest and τ_{s1} the smallest deadline.

3.3 Resource model

The *processor supply* refers to the amount of processor allocation that a component C_s can provide to its task set $\mathcal{T}_s$. The supply bound function $\mathtt{sbf}_{\Gamma_s}(t)$ of the periodic resource model $\Gamma(P_s, Q_s)$, that computes the minimum possible supply for any interval of length t, is given by [7]:

$$\mathtt{sbf}_{\Gamma_s}(t) = \begin{cases} t-(k+1)(P_s-Q_s) & \text{if } t \in V^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{1}$$

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

3.4 SRP-based Protocols

We assume that resource access is arbitrated using SRP [8]. SRP has non-blocking lock and unlock operations and makes it possible to share a single execution stack between all tasks within a component. These properties reduce a component's memory utilization [30]. To be able to use SRP in an HSF for synchronizing global resources, its associated ceiling terms need to be extended and excessive blocking must be prevented.

3.4.1 Preemption level

SRP introduces a *preemption level*, which is a static value assigned to each task and component. Each task τ_{si} has a preemption level equal to $\pi_{si} = 1/D_{si}$. Similarly, a component has a preemption level equal to $\Pi_s = 1/P_s$, where P_s serves as a component's periodic deadline. For FPPS this preemption level reduces to a deadline-monotonic priority assignment. By inverse ordering of tasks or components with respect to their relative deadlines [8], their priority field can be used to indicate the preemption level for EDF.

3.4.2 Resource ceiling

Every global resource has two types of resource ceilings: a *global* resource ceiling for global scheduling and a *local* resource ceiling for local scheduling. These ceilings are static, off-line calculated values, which are defined according to SRP as:

$$\begin{aligned} RC_l &= \max(\Pi_N, \max\{\Pi_s \mid R_l \in \mathcal{R}_s\}), && (2) \\ rc_{sl} &= \max(\pi_{sn_s}, \max\{\pi_{si} \mid \tau_{si} \text{ uses } R_l \in \mathcal{R}_s\}). && (3) \end{aligned}$$

These values represent the maximum preemption level of any component/task that shares resource R_l. We use the outermost $\max$ in (2) and (3) to define RC_l and rc_{sl} also in those situations where no component or task uses R_l. In [31], [32] techniques are presented to trade-off preemptiveness against resource holding times, here simply denoted as $X_{sl} \in \mathcal{X}_s$.

3.4.3 System and component ceilings

During run time we need to keep track of *component and system ceilings* and the scheduler needs to be extended with the notion of a these ceilings. These ceilings are dynamic parameters that change during execution. The system/component ceiling is equal to the highest global/local resource ceiling of a currently locked resource in the system/component. Under SRP, a task can only preempt the currently executing task if its preemption level is higher than its component ceiling. A similar condition for preemption holds for components.

3.4.4 Prevent excessive blocking

SIRAP [2] and BROE [3] use a self-blocking approach to prevent budget depletion inside a critical section. If a task wants to enter a critical section, it enters the critical section at the earliest time instant so that it can complete the critical section before the component's budget expires. If the remaining budget is not sufficient to lock and release a resource before expiration, (*i*) the task is suspended until new budget becomes available and (*ii*) the component ceiling is raised to limit other tasks in the component to execute until the resource is released. Contrary to SIRAP, BROE implements its own server model [3].

HSRP [1] uses an overrun mechanism to prevent budget depletion inside a critical section. When the budget of a component expires and the component has a task τ_i that is still locking a global shared resource, this task τ_i continues its execution until it releases the locked resource. When a task accesses a global resource the component ceiling is raised. HSRP assigns a static amount of overrun budget, X_s, to each component. Two alternatives of the overrun mechanism are presented: (*i*) overrun with payback, and (*ii*) overrun without payback. The payback mechanism requires that when an overrun happens in a component C_s, the budget of this component is decreased with the consumed amount of overrun in its next execution instant. Without payback no further actions are taken after an overrun has occurred. The relative strengths of both alternatives have been investigated in [11].

4 Transparent SRP-based Synchronization

Enabling the integration of HSRP, SIRAP and BROE into the same HSF requires that the primitive interfaces allow the connection and use of an arbitrary synchronization protocol without modification of operational procedures on either side of the interface, i.e., the interface is *transparent*. SRP's lock operation is a *non-blocking* primitive. This property is preserved by HSRP, but not by SIRAP and BROE. The latter two require to explicitly check the remaining budget before granting resource access. HSRP provides a static overrun budget, so that a task can finish its critical section even though its budget has depleted. Hence, all these protocols require knowledge of resource holding times (RHT) [31], either for comparison purposes or to determine the overrun budget. Filling in RHTs by hand in the lock interface may lead to high memory costs to store all these values and provides an error-prone way of programming. Moreover, a programmer typically does not know RHTs, because these values are platform dependent. Development tools therefore need to bridge the gap between system analysis and application engineering.

4.1 Component Integration and Analysis Recapitulated

FPPS is the de-facto standard in industry for task scheduling [5]. Having such support will simplify migration to and integration of existing legacy applications into the HSF, avoiding technology revolutions for engineers. Meanwhile, the distinction between local scheduling of tasks within a component and global scheduling of components makes it possible to deploy an EDF scheduler at the global level. This optimizes the system utilization due to the optimality of EDF with SRP-based resource arbitration [33]. We therefore allow an arbitrary global scheduler, i.e., FPPS or EDF, and assume FPPS to be used locally, in each component.

4.1.1 EDF-based Component Integration

The following sufficient schedulability condition holds for global EDF-based systems [33]:

$$\forall t > 0 \quad : \quad B(t) + \mathtt{dbf}_{\mathtt{EDF}}(t) \leq t \tag{4}$$

The demand bound function $\mathtt{dbf}_{\mathtt{EDF}}(t)$ computes the total processor demand of all components in the system for every time interval of length t, i.e.,

$$\mathtt{dbf}_{\mathtt{EDF}}(t) \quad = \quad \sum_{C_s \in \mathcal{C}} \left\lfloor \frac{t}{P_s} \right\rfloor Q_s \tag{5}$$

Equation (5) holds for idling periodic, polling and sporadic servers with implicit deadlines and for BROE. A deferrable server C_{ds} can be modeled as a periodic server by including a release-jitter term of $P_{ds} - Q_{ds}$ in the demand bound function [34]. The blocking term, $B(t)$, is defined as [33]:

$$B(t) \quad = \quad \max(0, \max\{X_u \mid P_u > t\}). \tag{6}$$

We use the outermost $\max$ in (6) to define $B(t)$ in those situations where no shared resources are accessed.

A component using HSRP demands more resources in its worst-case scenario [19]. We therefore replace Q_s in Equation (5) with $Q_s + X_s$. If overrun with payback is chosen, we only need to accommodate a single overrun budget within an interval of length t, i.e., we replace Q_s in Equation (5) with $Q_s + O_s(t)$, where

$$O_s(t) = \begin{cases} X_s & \text{if } t \geq P_s \\ 0 & \text{otherwise} \end{cases}$$

4.1.2 Deadline-Monotonic (DM)-based Component Integration

For global FPPS the following sufficient condition holds:

$$\forall 1 \leq s \leq N : \quad \exists t \in [0, P_s] : \quad \mathtt{dbf}_{\mathtt{DM}}(t, s) \leq t \tag{7}$$

where $\mathtt{dbf}_{\mathtt{DM}}(t, s)$ denotes the worst-case cumulative processor request of C_s for a time interval of length t, i.e.,

$$\mathtt{dbf}_{\mathtt{DM}}(t, s) \quad = \quad B_s + Q_s + \sum_{1 \leq r < s} \left\lceil \frac{t}{P_r} \right\rceil Q_r \tag{8}$$

Equation (8) holds for idling periodic, polling and sporadic servers [23]. A deferrable server C_{ds} can be modeled as a periodic server by including a release-jitter term of $P_{ds} - Q_{ds}$ in $\mathtt{dbf}_{\mathtt{DM}}$ [35]. The blocking term, B_s, is defined as [8]:

$$\begin{aligned} B_s \quad = \quad & \max(0, \max\{X_{ul} \mid \Pi_u < \Pi_s \wedge \\ & R_l \in \mathcal{R}_u \wedge RC_l \geq \Pi_s\}). \end{aligned} \tag{9}$$

We use the outermost $\max$ in (9) to define B_s in those situations where no shared resources are used within a component.

A component using HSRP requests more resources in its worst-case scenario, i.e., similar to the EDF case: Q_r in Equation (8) is replaced with $Q_r + X_r$. If overrun with payback is chosen, we only need to accommodate a single overrun budget within an interval of length t, i.e., replace the summation in Equation (8) by $\sum\limits_{1 \leq r < s} \left\lceil \frac{t}{P_r} \right\rceil (Q_r) + X_r$.

4.1.3 Component Analysis

By filling in task characteristics in the demand bound $\mathtt{dbf}$ of (4) or (7) and replacing their right-hand sides by (1), i.e., replace t for $\mathtt{sbf}_{\Gamma_s}(t)$, the same schedulability analysis holds for tasks executing within a component as for components at the global level[2]. Dependent on the chosen synchronization protocol, we may need to compensate for additional blocking effects. HSRP, with or without payback, has the same local blocking term as SRP [8], because it has a non-blocking lock operation. HSRP with payback, however, influences the processor supply to serviced tasks, see [19] for the modified $\mathtt{sbf}_{\Gamma_s}(t)$.

SIRAP's and BROE's self-suspension mechanisms lead to higher processor demands of serviced tasks. SIRAP imposes at most one self-blocking occurrence in each component period P_s [36], i.e., for an interval of length t the additional local self-blocking term has an upper bound of $\left\lceil \frac{t}{P_s} \right\rceil X_s$. BROE can

2. Polling servers [24] do not comply to the $\mathtt{sbf}_{\Gamma_s}(t)$ in (1), because serviced tasks may suffer a supply delay of $2P_s - Q_s$, i.e., larger than $2(P_s - Q_s)$.

self-block for at most $P_s - Q_s$, where self-blocking delays the resource supply for a virtual task arrival, being the critical section [3]. This delay influences the RHT, but does not directly influence the schedulability of $\mathcal{T}_s$ [27, Theorem 3], because the worst-case delay that any task serviced by BROE may suffer is $2(P_s - Q_s)$.

4.2 Global Transparency: Abstraction versus Efficiency

HSRP, SIRAP and BROE are SRP-based protocols, each of which complies to SRP-based global blocking rules. These protocols may be chosen with different server models according to the system integrator's needs. This transparent use of synchronization protocols can be provided

1) at the component level, so that each component that uses a shared resource is arbitrated by a predefined protocol;
2) at the task level, so that each task that uses a shared resource is arbitrated by a predefined protocol;
3) at the resource level, so that each resource is arbitrated by a predefined protocol;
4) per resource access.

The latter three options result in more complicated analysis where each component potentially suffers overrun and self-blocking in its budget dimensioning. By allowing transparent use of these protocols at a lower abstraction level, i.e., the highest level is per component and the lowest level is per resource access, a component interface should be supplied with more timing information. This makes it possible to perform a tighter, but more complicated system analysis and requires more timing information to be available during run time.

Firstly, BROE includes its own server model and therefore all its encompassed tasks are obliged to use the corresponding synchronization primitives. Secondly, these protocols have only been compared with relative strengths depending on component parameters [10], [11]. In line with these state-of-the-art results, we implement transparent synchronization interfaces for all three protocols at the component level.

4.3 A Framework-Specific Component Model

Given our choice for transparent interfaces at the component level, each component C_s may have a dedicated interface description $< P_s, Q_s, \mathcal{X}_s >$ for each of the three synchronization protocols. The system integrator can choose to instantiate a component based on the chosen server model and synchronization protocol. However, during system integration this choice may not be entirely free. Although the choice of a synchronization protocol is transparent from a programmer's perspective, it can be appurtenant to a particular framework.

For example, when a component uses a service from a shared library by means of calling *a reentrant function*, the access to this service is guarded by semaphores at the system level. Hence, when one of the protocols, HSRP, SIRAP or BROE, has been linked to these shared library services, the use of this protocol is also enforced to its calling components. A protocol can be linked to semaphore calls statically, during compile time, or dynamically, during run-time. In our implementation, we provide an initialization function that links a particular protocol to a component.

4.4 Transparent Interface Design

A weak precondition for transparency is that the *maximum* RHT, X_s, within component C_s is known from the analysis. We can store this information within the server-structure. This relaxes the amount of run-time information and allows to remove the RHT parameter from lock operations, although potentially at the cost of larger self-blocking times. Replacing the RHT with X_s requires to keep track of nested critical sections. Only the outermost critical section may self block (SIRAP and BROE) or give up overrun budget (HSRP).

We define an SRP interface to access global resources, and to maintain its corresponding data structure, as follows:

1) `void SRPMutexLock(Resource* r);`
2) `void SRPMutexUnlock(Resource* r);`

For notational convenience we will use different names for the lock and unlock primitives of each protocol. We include the RHT in each lock interface, which is accounted in processor cycles and allocated to the calling task's budget. This allows a programmer to overrule the choice for a protocol and reduce pessimistic budget assignments. If the RHT parameter is absent, then X_s is used.

4.5 Transparent Resource Scopes

A component designer may use local and global shared resources in the component's code. Local shared resources are internally hidden, while global shared resources are externally visible. In order to determine the local blocking of tasks, based on Equation (6) or (9), a programmer needs to determine the worst-case execution time (WCET) of each critical section, e.g., by using appropriate WCET tooling. These WCETs serve as an input for further analysis to determine RHTs based on techniques described in [31], [32]. Since accessing global resources may block tasks in other components, a programmer must specify the use of global resources. The system integrator cannot perform the global schedulability analysis without a valid upperbound on RHTs to global resources within each component. For each resource-component pair (R_l, C_s), at least the maximum RHT, denoted X_{sl}, is therefore recorded in the timing interface $< P_s, Q_s, \mathcal{X}_s >$.

Since filling in RHTs by hand is error-prone, software development environments or compiler tools should be extended with support to fill in these values in appropriate places in the (binary) code. Moreover, from an *opacity* perspective [37], i.e., neither the environment nor other components can modify a component's code, unified primitives may be desirable to access local and global resources, e.g., as proposed in [20]. This allows to decouple the specification and use of global resources in the actual implementation of a component with the help of analysis tooling and the development environment.

If during admission time the system detects that only a single component uses a global shared resource, then this resource may be considered as a local resource. During system integration, however, one needs to account for the worst-case global-blocking effects as specified by Equation (6) and (9). Moreover, in open environments where components may enter and leave the system during run time, global resource ceilings may be recalculated during admission time. Extensions of the admission procedure to dynamically fill in RHTs are also conceivable for such environments.

5 μC/OS-II AND HIERARCHICAL SCHEDULING

μC/OS-II is a microkernel which is maintained and supported by Micrium [38] and is applied in many application domains, e.g., avionics, automotive, medical and consumer electronics. The kernel is open source and is extensively documented [12]. The μC/OS-II kernel features preemptive multitasking for up to 256 tasks, and its size is configurable at compile time, e.g., services like mailboxes and semaphores can be disabled.

This section outlines our realizations of hierarchical time management and server implementations for μC/OS-II. These are required basic blocks to enable implementations of SRP-based protocols. In [21], [22], we presented a basic implementation of a two-level FPPS-based HSF with periodic idling, polling and deferrable servers in μC/OS-II for FPPS-based HSFs and we extended this framework with a global EDF scheduler and constant bandwidth servers (CBS) in [13].

5.1 Timed Event Management

Intrinsic to our reservation-based component scheduler is timed-event management. This comprises timers to accommodate (*i*) periodic timers at the global level for budget replenishment of periodic servers and at the component level to enforce minimal inter-arrivals of sporadic task activations; (*ii*) virtual timers to track a component's budget; (*iii*) timers to wake-up a server after a period of being suspended; and (*iv*) deadline timers to implement EDF.

In [22] a dedicated module is presented for managing *relative timed event queues* (RELTEQs). Its basic idea is to store timed events - called *timers* - relative to each other, by expressing the expiration time of the timer (i.e., the arrival time of the event) relative to the expiration time of the previous timer. The arrival time of the head event is relative to the current time. These timers as well as tasks and servers are stored in queues. The timer value of the head of each RELTEQ is decremented at each tick and the expiration of a timer triggers an event handler which manipulates these timer queues, a server ready queue or a task ready queue.

In a system we employ several timer queues to control tasks and servers, as illustrated in Figure 1. In case of single level scheduling, we just have a single system queue that represents the timer events associated with the arrival of tasks.

To support hierarchical scheduling we use this system queue for the scheduling of servers. The timers in this queue represent replenishment events corresponding to the start of a new period. In addition there is a local queue for each server which keeps track of the timers needed to manage the tasks inside a server such as task deadlines or the arrival of periodic tasks. At any time at most one server can be running on the processor; all other servers are inactive. When a server is suspended, its local queue is deactivated. In this configuration the hardware timer drives two timer queues, i.e., the local queue of the active (running) server and a system queue.

When the running server is preempted, its local queue is deactivated and the queue belonging to the newly scheduled server is activated. In order to ensure correct execution, the time that passed since the previous deactivation needs to be accounted for upon activation. To keep track of this time we introduce an additional queue: the *stopwatch queue*. Upon deactivation of a server, a timer is added at the head of this queue. The stopwatch queue is organized in the same relative fashion and contains one event for each non-active server. However, contrary to the other queues, the head is counted upwards. The accumulated time between the head of the stopwatch queue and a stopwatch timer represents the time since the corresponding server was deactivated. Whenever a server is activated, its local queue is synchronized with the stopwatch, i.e., all timers in its local queue which would have expired if the server was running are handled. As a result, all local timers with a smaller accumulated value than the stopwatch timer are popped from the local queue. After a simultaneous traversal of the stopwatch and local queues, its stopwatch event is subsequently deleted from the stopwatch queue. The time spent to synchronize the local queue of the newly activated server with global time is accounted to this server and subtracted from its budget.

Finally, a fourth queue represents timers that expire relative to the server budget. These events trigger the depletion of (a fraction of) the server's budget. We call these *virtual timers* as their notion of time is limited to the server budget. Rather than putting these in the system queue we have a separate queue for them, since otherwise we would need to insert them into the system queue upon activation and remove them again upon deactivation. This is therefore more efficient than the organization reported in [20]. In this new configuration, at every tick the heads of at most four queues are updated: a system queue, an active server queue, a stopwatch queue, and an active server virtual queue. The last queue does not need to get synchronized when a server is resumed, because a deactivated server does not consume its budget. Figure 1 shows an example of our timer-management. Based on this support for timer-management, we successfully applied an HSF to *independent video applications* in consumer electronics [22].

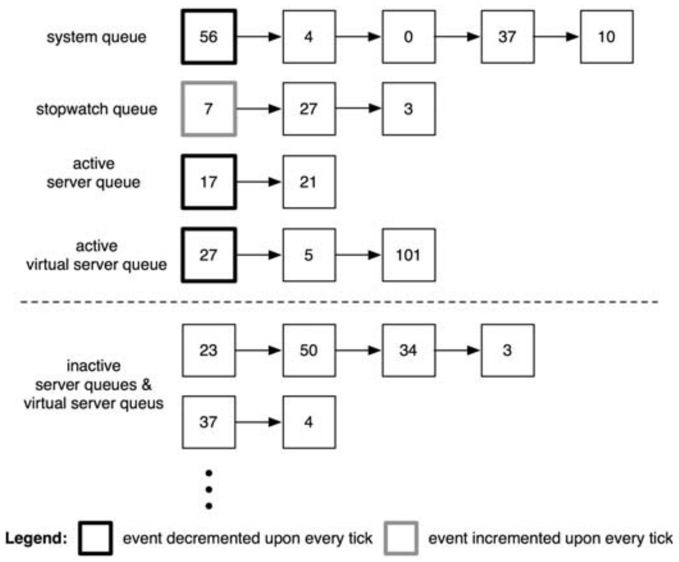
system queue
56
4
0
37
10
stopwatch queue
7
27
3
active server queue
17
21
active virtual server queue
27
5
101
inactive server queues & virtual server queus
23
50
34
3
37
4
Legend:
event decremented upon every tick
event incremented upon every tick

Figure 1. RELTEQ-based timer management for two-level HSFs. At each tick the head of the active queues are updated. When an event timer expires it is popped from the queue. Upon a server context switch the server queue of the activated server is synchronized with the stopwatch queue.

The current paper focuses on sporadic task scheduling and only considers the virtual timer of budget depletion; the queue therefore has length 1. Hence, SIRAP and BROE can inspect a component's virtual-timer in constant time to (implicitly) synchronize its remaining budget with global time before granting access to a global shared resource.

In contrast to an ordered list of absolute times, the relative time representation does not lead straightforwardly to a $O(log(N))$ implementation for inserting and deleting events, where N is the number of events in the queue. Although it is possible using *red-black* trees, this requires to keep track of absolute values in each node in the tree. We consider such an alternative implementation out of the scope of this paper.

5.2 Server Scheduling

We choose to limit HSRP and SIRAP to idling periodic, polling and deferrable servers. The sporadic server achieves a similar performance as a deferrable server, but at a higher implementation cost [35]. However, our implementation considerations of SRP-based synchronization protocols for a deferrable server also apply to a sporadic server. BROE comes with its own server design. Its implementation is presented in Section 6.4. Extending μC/OS-II with basic HSF support requires a realization of the following concepts:

5.2.1 Global Scheduling

Since the server period serves as a relative deadline for that server, the relative representation automatically sorts the corresponding timers in the system queue by absolute deadline. Similar to the μC/OS-II task scheduling approach, we introduce a bit-mask to represent whether a server has capacity left. When the scheduler is called, it traverses the deadline queue and activates the *ready* server with the earliest deadline in the queue. Subsequently, the μC/OS-II fixed-priority scheduler determines the highest priority ready task within the server.

5.2.2 Periodic Servers

Since μC/OS-II tasks are bundled in groups of sixteen to accommodate efficient fixed-priority scheduling, a server can naturally be represented by such a group. The implementation of periodic servers is very similar to implementing periodic tasks using our RELTEQ extensions [22]. Whereas a polling server immediately discards its budget when there is no ready task to be serviced, an idling server contains an idling task at the lowest, local priority, which is always ready to execute. A deferrable server suspends when no task is ready, until a task arrives. A server is only elected for execution when it contains a ready task.

A bandwidth preserving server, e.g., a deferrable server, may need to provide its preserved budget to tasks which arrive while it is suspended. Hence, we cannot simply deactivate its local event queue when its workload is exhausted. One option is to keep its local queue active, but this would increase the interference from suspended servers. We therefore proposed an alternative solution based on *wake-up timers* [22]. When a server is suspended, we insert a wake-up timer into the system queue with the expiration time of the first timer in the server's local queue, i.e., the earliest possible time of a sporadic arrival. When a wake-up timer expires, we change the server state to ready, allowing it to be scheduled by the global scheduler so that the corresponding local event gets handled.

5.2.3 Greedy Idle Server

In our HSF, we reserve the lowest priority levels for a deferrable idle server, which contains μC/OS-II's idle task at the lowest local priority. This server has the largest possible deadline, so that it always has the lowest priority. Only if no other server is eligible to execute, then the idle server is switched in.

6 Implementing SRP-based Synchronization

This section describes the common implementation parts of HSRP, SIRAP and BROE. The implementation of SRP as a local synchronization protocol can trivially be extracted. Thereafter, we present the three protocol implementations.

Our SRP-based HSF extensions entail changes to μC/OS-II. The protocol implementations are using (*i*) variable assignments, (*ii*) SRP-based operations, (*iii*) timed event management [22], (*iv*) a method enable and disable interrupts and (*v*) scheduler extensions. The first three building blocks are not specifically bound to μC/OS-II. The latter two are μC/OS-II specific, e.g., the extension of the scheduler by SRP's preemption rule [8] is eased by μC/OS-II's open-source character. This is accomplished by extending the μC/OS-II scheduler with two if-statements.

6.1 Scheduling and Tracking Ceilings

The μC/OS-II scheduler calls a function which returns the highest priority ready task. We extend this function with two rules according to SRP. When the scheduler selects the next server to be activated, its associated component priority must exceed the current system ceiling. Similarly, the priority of the selected task must exceed the component ceiling. Otherwise, the priority on top of the global/local stack is returned.

Each global resource accessed using an SRP-based mutex is represented by a `Resource` structure, defined as follows:

```
typedef struct resource{
    INT8U  ceiling;
    INT8U  lockingTask;
    void*  previous;
} Resource;
```

We use the `Resource` data-structure to implement a *system ceiling stack*. `ceiling` stores the resource ceiling and `lockingTask` stores the identifier of the task currently holding the resource. The `previous` pointer is used to maintain the stack structure, i.e., it points to the previous `Resource` structure on the stack that maintains the system ceiling. The `ceiling` field of the `Resource` on top of the stack represents the current system ceiling. The primitives *SRPMutexLock* and *SRPMutexUnlock* maintain this global-ceiling stack structure.

Each global resource is also represented by a `localResource` structure defined as follows:

```
typedef struct {
  resource* globalResource;
  INT8U     localCeiling;
  INT8U     localLockingTask;
  void*     previous;
} localResource;
```

The `localResource` data-structure maintains a separate *component ceiling stack* for each component. The `globalResource` points to the corresponding resource block at the global level, i.e., all global resources are mapped to a corresponding local resource. `localCeiling` stores the local resource ceiling and `localLockingTask` stores the identifier of the task currently holding the resource. The `previous` pointer is used to maintain the stack structure, i.e., it points to the previous `localResource` structure on the stack. The `localCeiling` field of the `localResource` on top of the stack represents the current component ceiling. The primitive *updateComponentCeiling* maintains this local ceiling stack structure.

6.2 SIRAP Implementation

This section presents our SIRAP implementation. The kernel primitives presented throughout this paper are assumed to execute non-preemptively, unless denoted differently.

6.2.1 Resource Locking

The lock operation first updates the component's local ceiling according to SRP to prevent other tasks within the component from interfering during the execution of the critical section. In order to successfully lock a resource there must be sufficient remaining budget within the server's current period. The remaining budget $Q_{remaining}$ is returned by a function that depends on the virtual timers mechanism, see Section 5.1. If $Q_{remaining}$ is not sufficient to complete the critical section, the task will spinlock until the next replenishment event expires. To avoid a race-condition between a resource unlock and budget depletion, we require that $Q_{remaining}$ is strictly larger than RHT before granting access to a resource. The lock operation is presented in more detail in Pseudo-code 1.

Pseudo-code 1 void SIRAP_lock(`Resource*` r, `INT16U` RHT);

```
1: $updateComponentCeiling()$;
2: while $RHT >= Q_{remaining}$ do
3:     $enableInterrups$;
4:     $disableInterrups$;
5: end while
6: $SRPMutexLock(r)$;
```

When the server's budget is replenished, all tasks spinlocking on a resource are unlocked as soon as they are rescheduled. Although after budget replenishment a repeated test on the remaining budget is superfluous [2], spinlocking may efficiently implements the self-blocking mechanism. A disadvantage of this implementation is that it relies on the assumption of an idling periodic server. For any budget-preserving server, e.g., the deferrable server [25], the self-blocking mechanism by means of a spinlock is unacceptable, because a task consumes server budget during spinlocking.

An alternative implementation is to suspend a task when the budget is insufficient and resume a task when the budget is replenished. However, μC/OS-II requires at any time a schedulable ready task, which is optionally a special idle task at the lowest priority. The system and component ceilings prevent the idle task to be switched in. Making an exception for the idle task potentially breaks the property of SRP allowing to share stack space among tasks within a component [8], [30]. The idle task should always be eligible to execute and therefore must have its own execution stack. Idling a processor is typically more energy efficient than spinlocking, e.g., [39] presents techniques to bundle idling intervals in reservation-based systems. Since μC/OS-II provides a context-preserving task model, i.e., threads, it does not support stack sharing between tasks anyway. We therefore opt for the latter implementation, which self-suspends a task, rather than spinlocking.

6.2.2 Resource Unlocking

Unlocking a resource simply means that the system and component ceiling must be updated and the global SRP mutex must be released. Note that the latter command will also call the scheduler.

6.3 HSRP Implementation

This section presents our HSRP implementation. Additionally to the implementation of HSRP's locking and unlocking operations, we need to adapt the budget-depletion event handler to cope with overrun. This requires to keep track of the number of resources locked ($lockedResourceCounter$) within component C_s. HSRP assigns a static amount of overrun budget, X_s, to each server within the system [1]. The server data-structure is extended with five additional fields for bookkeeping purposes, i.e., *lockedResourceCounter*, *inOverrun*, *unusedOverrun*, *paybackEnabled* and X_s. Optionally, we implement a payback mechanism in the budget replenishment event. These event handlers are managed by our timed-event module as presented in Section 5.1.

6.3.1 Resource Locking

The lock operation first updates the component's local ceiling to the highest local priority to prevent other tasks within the component from interfering during the execution of the critical section. Next, the system ceiling is updated by locking the SRP mutex with *SRPMutexLock*.

6.3.2 Resource Unlocking

Unlocking a resource means that the system and component ceilings must be updated and the SRP mutex must be released. In case that any overrun budget is consumed and no other global resource is locked within the same component, we need to inform the scheduler that overrun has ended. Optionally, the amount of unused overrun budget is stored to support payback upon the next replenishment. The unlock operation in pseudo-code is:

Pseudo-code 2 void HSRP_unlock(`Resource* r`);

```
1: $updateComponentCeiling()$;
2: $C_s.lockedResourceCounter--$;
3: if $C_s.lockedResourceCounter == 0$ and $C_s.inOverrun$ then
4:     if $C_s.paybackEnabled$ then
5:         $C_s.unusedOverrun = X_s - Q_{remaining}$;
6:     end if
7:     $C_s.inOverrun$ = false;
8:     $setComponentBudget(0)$;
9: end if
10: SRPMutexUnlock($r$);
```

The command $setComponentBudget(0)$ performs two actions: (*i*) the server is blocked to prevent the scheduler from rescheduling the server, and (*ii*) the budget-depletion timer is canceled by updating RELTEQ's virtual event queue.

6.3.3 Budget Depletion

We extend the event handler corresponding to a budget depletion with the following rule: if any task within the component holds a resource, then the budget is replenished with an amount X_s and server inactivation is postponed. This requires to inserts a new event in RELTEQ's virtual event queue with the value X_s.

6.3.4 Budget Replenishment

For each periodic server an event handler is periodically executed to recharge its budget. To support the optionally enabled payback mechanism, the replenished budget is decreased with the unused overrun after consuming overrun budget in the previous period.

When a server is still consuming overrun budget while its normal budget is replenished, the overrun state of this server is reset [40]. The original HSRP analysis [1] implicitly assumes that periodic budgets $Q_s + X_s$ complete before their deadline D_s. However, this is a pessimistic but sustainable [41] assumption. Figure 2 shows an example where a component replenishes its budget during a critical section. Although exact analysis is not provided, [40] shows that the schedulability analysis can be considerably improved compared to the analysis in [1] when a component's deadline only holds for budget Q_s rather than $Q_s + X_s$. The implementation of this observation is compliant with the analysis in [1].

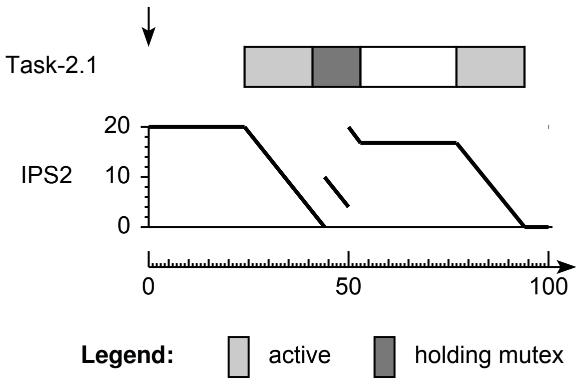

Figure 2. The overrun budget of a server (IPS2) depletes when its normal budget replenishes (without payback). This example is generated from instrumented code [42].

6.4 BROE Implementation

BROE assumes global EDF scheduling and implements its own server model. First, this section presents our realization of the BROE server in our HSF, followed by BROE's synchronization primitives.

6.4.1 Server Implementation

The BROE server is related to a CBS [43]: a bandwidth preserving server which immediately replenishes its budget upon depletion, however, with a postponed deadline. Unfortunately, a CBS is not suitable for HSFs, because it suffers from the *deadline aging* problem [44]. This happens when the scheduling deadline of a server is postponed many times and takes a large value compared to the scheduling deadlines of the other servers.

To resolve this problem, a BROE server for a component C_s modifies a plain CBS by two additional rules [3]:

1) when the server's budget is exhausted, the server is suspended until its deadline. When the deadline expires, the budget is recharged with a relative deadline P_s. Hence, the absolute deadline of a BROE server is represented in the system queue and its event handler replenishes the server's budget.
2) when a task arrives after a time interval that server C_s was idle and has remaining budget, then the server C_s is suspended when its virtual time is running ahead relative to its absolute deadline t_{d_k}. This means that the BROE server resumes with a recharged budget and a relative deadline P_s (i.e., equal to its period) when the absolute time has passed the virtual time.

These rules guarantee a server utilization of $U_s = \frac{Q_s}{P_s}$, but introduce the notion of a suspended server state, which is absent in a plain CBS. The first rule is similar to a deferrable server with the difference that the deadline of a BROE server is no longer strictly periodic due to the second rule. The second rule introduces the notion of virtual time, which can be derived from the server's deadline and the virtual-timer mechanism as presented in Section 5.1. However, notice the difference with the value of the event in the virtual server queue, which represents the server's currently remaining budget, $Q_{remaining}$.

Within component C_s the virtual time advances with a rate $\frac{Q_s}{P_s}$. When the component consumed too much of its budget relative to its absolute deadline t_{d_k}, it has to calculate the time t_a until the next replenishment, where $t_a = t_{d_k} - \frac{P_s}{Q_s} Q_{remaining}$ with a corresponding deadline $t_{d_{k+1}} = t_a + P_s$. In other words, we perform the following actions to suspend a server: (*i*) replenish its budget, (*ii*) postpone its deadline to $t_a + P_s$ and (*iii*) insert a wake-up event in the system queue which expires at time t_a. If time t_a has already passed, then a replenishment can even happen without self-blocking, because the component was lagging behind. Similarly, when a server has exhausted its workload, its local queues can be deactivated at least until time t_a, because any earlier local event expiration will be postponed anyway until t_a.

6.4.2 Resource Locking

The critical section of a task serviced by a BROE server is considered as a task arrival on itself and the server will correspondingly suspend itself, if necessary, according to the second rule presented above.

The lock operation first updates the component's local ceiling to prevent other tasks within the component from interfering during the execution of the critical section. In order to successfully lock a resource there must be sufficient remaining budget. If $Q_{remaining}$ is not sufficient to complete the critical section, the server may get suspended until the absolute time passes the virtual time. This requires to set a timer in the system queue to wake up the server, because the virtual timer does not advance when the server is suspended. Moreover, BROE fetches its current deadline, postpones its deadline and replenishes its budget, which requires three more timer operations.

This budget becomes available after resuming the server and from that time instant this budget is serviced before a relative deadline of P_s. Finally, the scheduler is called to determine which other component has the earliest deadline. When the server is woken up and switched in again, sufficient budget is guaranteed to complete the critical section. Pseudo-code 3 denotes the lock operation in more detail[3].

Pseudo-code 3 void BROE_lock(`Resource*` r, `INT16U` RHT);

```
1:  $updateComponentCeiling()$;
2:  **while** $RHT >= Q_{remaining}$ **do**
3:      $t_a \leftarrow getComponentDeadline() - \frac{P_s}{Q_s} Q_{remaining}$;
4:      **if** $t_a > 0$ **then**
5:          $ComponentSuspendUntil(t_a)$;
6:      **end if**
7:      $setComponentDeadline(t_a + P_s)$;
8:      $setComponentBudget(Q_s)$;
9:      $enableInterrups$;
10:     $Schedule()$;
11:     $disableInterrups$;
12: **end while**
13: $SRPMutexLock(r)$;
```

To avoid a race-condition between a resource unlock and budget depletion, we require that $Q_{remaining}$ is strictly larger than RHT before granting access

3. Strictly speaking, an if-statement is a sufficient budget test, because the component ceiling prevents interference of other tasks within the component. The while loop is a form of defensive programming.

to a resource.

6.4.3 Resource Unlocking

Unlocking a resource simply means that the system and component ceilings must be updated, so that the global SRP mutex is released.

6.5 Overview of Implementation Complexity

The implementation complexity for each of the three synchronization protocols as discussed in this Section is summarized in Table 1. These results show that BROE's self-blocking mechanism is especially expensive due to expensive timer operations. HSRP requires to reset budget timers and SIRAP does not need to change any timer. However, these results only hold for periodic-idling servers.

Table 1
Overview of the synchronization primitive's implementation complexity for HSRP, SIRAP and BROE.

Event	HSRP	SIRAP	BROE
Lock resource	-	suspend (or spinlock)	suspend server; renew deadline
Unlock resource	end overrun	-	-
Budget depletion	start overrun	-	-
Budget replenishment	payback (optional)	resume (end spinlock)	resume

6.5.1 Self-blocking on Bandwidth-Preserving Servers

SIRAP may suspend a deferrable server when a task experiences insufficient budget to complete its critical section. Meanwhile, its component ceiling is raised. If a task arrives that is blocked by the component ceiling, the server should remain suspended until the earliest time instant that either (*i*) its budget replenishes, so that a self-blocking task can access its desired resource, or (*ii*) a next sporadic event arrives which repeats this rule. It is therefore required to suspend all tasks with a lower priority than the component ceiling upon locking a global resource and resume these tasks upon unlocking the resource[4]. In the special case where resource accesses are chosen to be locally non-preemptive, e.g., as in [1], a server can discard its remaining budget and suspend itself until its budget replenishes. Alternatively, we can re-suspend the server upon detection of an arrived event. A corresponding implementation changes the scheduler by extending the SRP-rules presented in Section 6.1.

4. A similar implementation is chosen in [20], because they manipulate scheduling queues from outside the kernel.

Although the overhead for suspending a task is only there when a task actually arrives, this overhead can take place at arbitrary moments in time. These overheads make SIRAP expensive and unattractive for bandwidth-preserving servers.

BROE does not suffer this additional self-blocking complexity, because a task arrival does not change the server's state and any task arrival during suspension will be served according to its priority after the server resumes. When BROE resumes its execution, a fully recharged budget with a newly calculated deadline becomes available to all its ready tasks. Any task arrival during self suspension can therefore be safely postponed until the server resumes.

6.5.2 Which Protocol to Choose

Given the three alternative protocols, HSRP, SIRAP and BROE supported by our framework, the question is: which protocol to use? Although both BROE and SIRAP assume that resource-holding times fit within a component's budget, i.e., $X_{sl} \leq Q_s$, this assumption is unnecessary for HSRP. HSRP still requires that a budget of $Q_s + X_s$ can be guaranteed within a single period P_s, however, so that there is essentially no difference with allocating a larger budget such that critical sections fit, i.e., satisfying the condition $X_{sl} \leq Q_s$.

Hence, from an analytical perspective, BROE is superior compared to SIRAP and HSRP, because it does not allocate dedicated overrun budgets in addition to a component's normal budget, like HSRP does, and it also does not account for self-blocking durations in the analysis, like SIRAP does. This makes BROE an attractive mechanism to limit global preemptions in regions where arbitrary preemptions may cause significant run-time overhead [45], in particular due to architectural related preemption costs, such as cache misses and pipeline flushes.

Both BROE and HSRP require expensive timer operations, whereas SIRAP can be efficiently implemented on idling periodic servers. Independent of the global scheduling policy, this makes SIRAP the preferred protocol for applications with many short critical sections. In those situations, SIRAP's analytical overhead is relatively small and the timer manipulations of BROE and HSRP can consume more time than the critical section itself.

Finally, BROE only supports HSFs based on global EDF scheduling. For HSFs based on global FPPS, we can choose between SIRAP and HSRP. In [11] it has been concluded, based on simulation studies which investigate their effect on the system load, that both alternatives are incomparable and their relative strengths depend on the component parameters, $< P_s, Q_s, \mathcal{X}_s >$. Their simulation results show that, in most cases, HSRP is preferred for components where task periods are relatively close to the server period and SIRAP works better for task sets with a relatively small server period compared to task periods.

7 Experiments and Results

In this section we compare the implementations for HSRP, SIRAP and BROE. First, we present a brief overview of our test platform. Next, we compare the implementations of these protocols and investigate the system overhead of their primitives. Finally, we demonstrate each protocol's behaviour by means of an example system.

7.1 Experimental Setup

We recently created a port for μC/OS-II to the OpenRISC platform to experiment with the accompanying cycle-accurate simulator. This open-source hardware platform is developed by the OpenCores project [46] and comprises a scalar processor and basic peripherals. The OpenRISC simulator allows software-performance evaluation via a cycle-count register. This profiling method may result in either longer or shorter measurements between two matching calls due to the pipelined OpenRISC architecture. Some instructions in the profiling method interleave better with the profiled code than others. The measurement accuracy is approximately 5 instructions.

Since it is important to know whether an real-time operating system behaves in a timewise predictable manner [47], we investigate the disabled interrupt regions caused by the execution of our synchronization primitives. Disabling interrupts directly affects the response of the system to external events, but is the only way for tasks in μC/OS-II to share data structures between primitives executed in the task's context and ISRs. In [47] the worst-case execution times of almost all μC/OS-II primitives, including its synchronization services, are investigated. These execution times mainly depend on a task's state. In our hierarchical setup, however, some primitives also include timer operations. Our primitives therefore have partially static and dynamic execution times.

Our HSF supports at most 16 servers, where the highest and lowest priority levels are reserved by the operating system. For each multiple of 2 servers within our supportive range, we generated 100 random task sets with a total utilization of 45%. We repeated this for all three synchronization protocols, i.e., we tested in total 2100 setups. Each component comprises 6 tasks with implicit deadlines, i.e., $D_{si} = T_{si}$. We choose idling periodic servers in combination with HSRP and SIRAP. Each server period, P_s, is the largest integer value which is lower than half of the smallest task period serviced by C_s, i.e., $2P_s \leq T_{si}^{min}$. All tasks τ_{si} in the system access a single global resource for a duration varying between $0.1C_{si}$ and $0.25C_{si}$. These critical sections are locally and globally non-preemptive for other tasks. Only interrupt handlers may therefore interfere during global resource access.

7.2 Synchronization Overheads

We now present the time and space complexity of the implemented synchronization primitives. A single-level SRP implementation [21] is used as a reference.

7.2.1 Time Complexity

All primitives are independent of the number of servers and tasks in a system, except BROE's lock operation, see Table 2. BROE may change the deadline ordering of components which makes its lock operation linear in the number of components in the system.

Both SIRAP implementations are presented in Table 2, i.e., first the suspend-resume and subsequently the spinlocking version. If we implement SIRAP by means of spinlocking, then its overheads stay locally in a component, i.e., the spinlock adds to the budget consumption of the particular task that locks the resource. SIRAP's overhead consists of at least a single test for sufficient budget in case the self-blocking test is passed. The overhead is at most two of such tests in case the initial test fails, i.e., one extra self-blocking test is done after budget replenishment and before resource access is granted. All remaining tests during spinlocking are already included as self-blocking terms in the local analysis [36]. The processor instructions executed for a single test are 10 instructions. Our default implementation based on suspend-resume, which includes a complementary scheduler rule for deferrable servers, blocks a task on a semaphore which is released upon budget replenishment. This implementation may cause longer interrupt latencies compared to spinlocking, but the number of interrupt-disabled regions is decreased.

HSRP introduces overheads that interfere at the system level, i.e., the overrun mechanism requires the manipulation of budget timers. These timer manipulations cause deviations in the unlock operation's execution time. The best-

Table 2
Overview of best-case (BC) and worst-case (WC) execution-times for SRP-based protocols measured in number of processor instructions.

Event	**SRP**		**HSRP**		**SIRAP**		**BROE**	
	BC	WC	BC	WC	BC	WC	BC	WC
Lock resource	124	124	196	196	214	320 / 224	214	> 1200 $\mathcal{O}(N)$
Unlock resource	106	106	196	725 $\pm$ 2	192	192	192	192
Budget depletion	-	-	0	383	-	-	-	-
Budget replenish	-	-	0	15	0 / -	68 / -	-	-

case HSRP overhead is null in addition to the normal number of processor instructions that are spent to increase and decrease the component and system ceilings. The worst-case HSRP overhead occurs at overrun. When the budget depletes, it is replenished with an overrun budget of X_s, which takes 383 instructions. Overrun completion can occur when a task unlocks a resource while consuming overrun budget. The system overhead for overrun-completion is on average 725 instructions with 2 instruction defining its 95% confidence interval. When the payback mechanism is enabled, one additional computation is done to calculate the number that needs to be paid back at the next server replenishment, i.e., an overhead of 5 instructions.

BROE's self-blocking mechanism is relatively expensive, because it manipulates deadline orderings of components, its component's budget timer and may suspend itself. BROE's overhead consists of at least a single test for sufficient budget, i.e., 10 instructions, and at most two such tests (like SIRAP). The cost for budget replenishment is similar to overrun, i.e., 383 instructions. Deadline postponement and server suspension each cost at least the same, but these costs increase linearly in the number of servers in the system. Although these timer manipulations before accessing a resource replaces those due to budget replenishment in periodic servers, these expensive timer operations may happen more often in the presence of shared resources and should be accounted to BROE.

7.2.2 Memory Complexity

The code sizes in bytes of all three protocols are presented in Table 3. For comparison purposes μC/OS-II's synchronization primitives of counting semaphores and the priority-inheritance-based protocol are included in this table, as well as our task-level SRP implementation. The sizes of the three global protocols are approximately two times the size of plain SRP. Only BROE's lock operation is considerably more expensive due to its ready-queue manipulations. The last column presents the code sizes of the transparently implemented primitives. The lock operation is slightly increased in size due to glue code that enables protocol transparency. The size of the unlock operation is determined by HSRP.

In Section 6.1 we showed that each SRP resource has a data structure in each component that shares this resource and at the global level. At the global level, the system-ceiling stack contains at most M shared resources, i.e., linear in the size of $\mathcal{R}$. Each component C_s has a similarly sized stack structure, i.e., $\mathcal{O}(M_s)$. Furthermore, counting semaphores and priority-inheritance-based protocols do not support resource sharing across budgets and require waiting queues to track tasks that pend on a locked resource. For these *event-based* protocols, μC/OS-II reserves two bytes per task in the system for each resource

Table 3
Code sizes (in bytes) of SRP primitives compared to μC/OS-II's counting semapphore (CS) and priority-inheritance protocol (PIP).

Function	**CS**	**PIP**	**SRP**	**SIRAP**	**HSRP**	**BROE**	**SIRAP+HSRP+BROE**
Lock resource	416	924	196	492	436	792	820
Unlock resource	208	396	192	384	436	384	436
Create resource	196	304	228	168	168	168	168
Delete resource	460	560	116	40	40	40	40
Set component protocol	-	-	-	-	-	-	192
Set component ceiling	-	-	-	264	264	264	264
Set component's RHT	-	-	-	148	148	148	148

control block. SRP-based protocols do not need this expensive infrastructure, because waiting queues are contained in the ready queue. Instead, the server control blocks are extended with five integers for timer management.

7.3 Transparent Interfaces: An Example

We recently extended our development environment with a visualization tool, which makes it possible to plot a HSF's behaviour [42]. To demonstrate the behavioural differences between SIRAP, HSRP and BROE, consider an example system consisting of three components, i.e., two deferrable servers (DS 1 and DS 2) and a BROE server (see Table 4). The system comprises five tasks[5] divided over the servers, which share a single global resource R_1 (see Table 5). The component/task with the lowest index has the highest preemption level/priority. The local resource ceilings, rc_{sl}, of R_1 are chosen to be equal to the highest local priority.

Figure 3 shows the behaviour of the EDF-scheduled example system, where the period of a server represents its relative deadline. DS 1 selects SIRAP and DS 2 selects HSRP with payback. SIRAP suspends DS 1 (time 5) and postpones resource access due to insufficient budget. Although DS 1 preserves its capacity during self-suspension, task 1 cannot start at time 10 due to the raised component ceiling. HSRP immediately grants access to R_l and allows task 4 to overrun its server's budget for the duration of the critical section. DS 2

5. The computation time C_{si} denotes the task's consumed time units before/after locking/unlocking a resource, e.g., the scenario $C_{s1,1}$; $Lock(R_1)$; $C_{s1,2}$; $Unlock(R_1)$; $C_{s1,3}$ is denoted as $C_{s1,1} + C_{s1,2} + C_{s1,3}$.

Table 4
Example System: component parameters

Component	Period (P_s)	Budget (Q_s)	Max. blocking (X_s)
DS 1	50	20	15
DS 2	60	20	15
BROE	75	20	10

Table 5
Example System: task parameters

Server	Task	Period	Computation time[4]	Phasing
DS 1	Task 1	125	10	10
DS 1	Task 2	150	5+15+5	0
DS 2	Task 3	125	10	10
DS 2	Task 4	200	5+15+5	0
BROE	Task 5	220	15+10+5	0

replenishes its budget with X_2 at time 25. At time 36 task 4 unlocks R_1 and the remainder of X_2 is discarded. The normal budget of DS 2 is reduced with its consumed overrun at the next replenishment (time 60). Task 5 encounters insufficient budget to complete its critical section at time 52. The BROE server has a deadline of 75 and a remaining budget of 5 time units, so that it suspends its execution until time $75 - \frac{75}{20}5 \approx 56$. In addition, it replenishes its budget with a renewed deadline of 131. After the suspension expires at time 56, task 5 is serviced by its server so that it gains access to resource R_1 at time 75.

8 Evaluation

Short critical sections that share resources with the kernel, e.g., system calls, are typically executed non-preemptively. Disabling and enabling interrupts itself is very cheap, i.e., approximately 5 instructions. Long non-preemptive critical-section durations may hamper the system's responsiveness, however, so that it becomes advantageous to trade preemptivenes against resource holding times. This is where our protocol implementations come into play.

We showed that each of the protocol implementation has bounded computation times and jitter, which makes the use of these protocols timewise predictable. For protocols that need to manipulate timers, i.e., HSRP and BROE, these jitters may become larger. We refer to [22] for a more extensive study on timer management in our HSF. The execution times of the primitives must be included in the system analysis by adding these to the RHTs in a component's interface, i.e., $X_{sl} \in \mathcal{X}_s$.

By default, current analysis techniques do not account for overheads of the operating system and the corresponding synchronization primitives. Us-

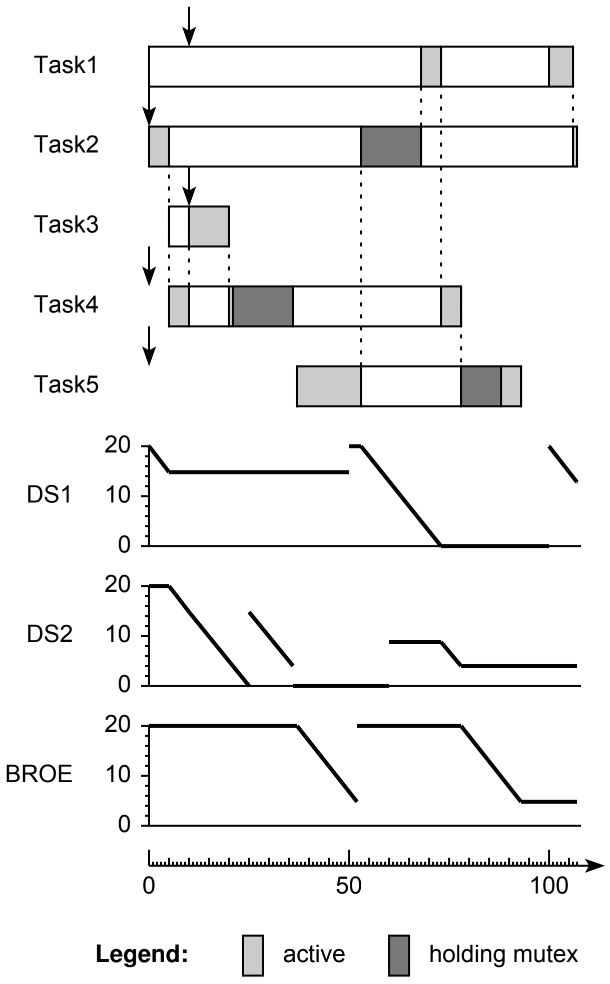

Figure 3. Example combining SIRAP (DS1), HSRP with payback (DS2) and BROE to access one shared resource, generated from instrumented code [42].

ing more advanced analysis methods, for example as proposed in [48], our measures can be included in the existing system analysis. Our experiments do not incorporate scheduling costs, because these are the same for all protocols. At the cost of implementing a data structure that maintains a waiting queue for suspended tasks in the HSF, one can reduce the number of scheduler calls, see [20]. We changed the scheduler, so that the ready queue of μC/OS-II contains SRP's waiting queues. Hence, we always call the scheduler at unlocking a resource.

9 Conclusion

HSFs have been developed to provide temporal isolation between components and decouple independent development of components from their integration on a shared platform. We described the implementation of three SRP-based synchronization protocols, i.e., HSRP, SIRAP and BROE, in a two-level preemptively scheduled HSF to support inter-component synchronization. We transparently offer these protocols side-by-side within the same HSF, so that their primitives can be selected based on relative strengths of the protocol.

Our implementations are based on an HSF-enabled real-time operating system, μC/OS-II. Our HSF builds on an hierarchical setup of relative timed-events queues (RELTEQs). The decoupling of system- and server-related events limits the interference of handling expiring timers corresponding to inactive servers from the perspective of the executing server. To support SIRAP's and BROE's self-blocking mechanisms, we implemented a novel virtual-timer mechanism. HSRP uses a run-time overrun mechanism, which is implemented such that overrun consumption ends when a component's normal budget replenishes. BROE provides a CBS-based server and uses a self-blocking mechanism in conjunction with deadline-postponement. Because BROE and HSRP require expensive timer operations in their primitives, SIRAP is the preferred protocol for integrating legacy components into the HSF with many short critical sections. BROE is especially advantageous for global EDF-scheduled systems with relatively large non-preemptive code fragments. Finally, in global FPPS-based systems, HSRP is beneficial compared to SIRAP when task periods are relatively close to their server period.

For each of the synchronization protocols we discussed the system overhead. The memory requirements of these protocols are lower than priority-inheritance-based protocols where tasks may pend in a waiting queue. Furthermore, our primitives have bounded computation times and jitter. BROE's lock operation is especially expensive, in terms of time and space complexity, due to its deadline manipulation and self-suspension. The implementation complexity of HSRP is independent of the underlying server model. SIRAP's overhead is low for idling periodic servers, which are therefore preferred above deferrable servers. As a future work we leave software-development-environment extensions that support HSFs and global resource sharing.

Acknowledgment

The authors are grateful to the anonymous reviewers for their useful feedback and their help to improve the quality of this paper.

REFERENCES

[1] R. Davis and A. Burns, “Resource sharing in hierarchical fixed priority pre-emptive systems,” in *Real-Time Systems Symp.*, Dec. 2006, pp. 257–267.
[2] M. Behnam, I. Shin, T. Nolte, and M. Nolin, “SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems,” in *Conf. on Embedded Software*, Oct. 2007, pp. 279–288.
[3] M. Bertogna, N. Fisher, and S. Baruah, “Resource-sharing servers for open environments,” *IEEE Trans. on Industrial Informatics*, vol. 5, no. 3, pp. 202–219, Aug. 2009.
[4] AUTOSAR GbR, “Technical overview,” 2008. [Online]. Available: http://www.autosar.org/
[5] OSEK Group, “OSEK VDX operating system specificiation 2.2.3.” [Online]. Available: http://www.osek-vdx.org/
[6] T. Nolte, I. Shin, M. Behnam, and M. Sjodin, “A synchronization protocol for temporal isolation of software components in vehicular systems,” *IEEE Trans. on Industrial Informatics*, vol. 5, no. 4, pp. 375–387, Nov. 2009.
[7] I. Shin and I. Lee, “Compositional real-time scheduling framework with periodic model,” *ACM Trans. on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–39, 2008.
[8] T. Baker, “Stack-based scheduling of realtime processes,” *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, March 1991.
[9] T. M. Ghazalie and T. P. Baker, “Aperiodic servers in a deadline scheduling environment,” *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.
[10] M. Behnam, T. Nolte, M. Åsberg, and I. Shin, “Synchronization protocols for hierarchical real-time scheduling frameworks,” in *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Nov. 2008, pp. 53–60.
[11] M. Behnam, T. Nolte, M. Åsberg, and R. J. Bril, “Overrun and skipping in hierarchically scheduled real-time systems,” in *Conf. on Embedded and Real-Time Computing Systems and Applications*, Aug. 2009, pp. 519–526.
[12] J. J. Labrosse, *MicroC/OS-II: the real-time kernel (2nd edition)*. CMP Books, 2002.
[13] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, “Protocol-transparent resource sharing in hierarchically scheduled real-time systems,” in *Conf. on Emerging Technologies and Factory Automation*, Sept. 2010.
[14] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, “Resource sharing in reservation-based systems,” in *Real-Time Systems Symp.*, 2001, pp. 171–180.
[15] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronisation,” *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.
[16] U. Steinberg, J. Wolter, and H. Härtig, “Fast component interaction for real-time systems,” in *Euromicro Conf. on Real-Time Systems*, July 2005, pp. 89–97.
[17] G. Lipari, G. Lamastra, and L. Abeni, “Task synchronization in reservation-based real-time systems,” *IEEE Trans. on Computers*, vol. 53, no. 12, pp. 1591–1601, Dec. 2004.
[18] G. Buttazzo and P. Gai, “Efficient implementation of an EDF scheduler for small embedded systems,” in *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.
[19] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, “Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems,” *IEEE Trans. on Industrial Informatics*, vol. 6, no. 1, pp. 93–104, Feb. 2010.
[20] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, “Implementation of overrun and skipping in VxWorks,” in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010, pp. 45–52.
[21] M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam, “Extending a HSF-enabled open-source real-time operating system with resource sharing,” in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010, pp. 71–81.
[22] M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, and J. J. Lukkien, “Constant-bandwidth supply for priority processing,” *IEEE Trans. on Consumer Electronics*, vol. 57, no. 2, pp. 873–881, May 2011.

[23] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Real-Time Systems Symp.*, Dec. 2005, pp. 389–398.
[24] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Syst.*, vol. 1, no. 1, pp. 27–60, June 1989.
[25] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. on Computers*, vol. 44, no. 1, pp. 73–91, Jan. 1995.
[26] P. J. L. Cuijpers and R. J. Bril, "Towards budgeting in real-time calculus: deferrable servers," in *Conf. on Formal modeling and analysis of timed systems*. Springer-Verlag, 2007, pp. 98–113.
[27] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257–269, 2005.
[28] I. Shin, M. Behnam, T. Nolte, and M. Nolin, "Synthesis of optimal interfaces for hierarchical scheduling with resources," in *Real-Time Systems Symp.*, Dec. 2008, pp. 209–220.
[29] M. Behnam, T. Nolte, and N. Fisher, "On optimal real-time subsystem-interface generation in the presence of shared resources," in *Conf. on Emerging Technologies and Factory Automation*, Sept. 2010.
[30] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Real-Time Systems Symp.*, Dec. 2001, pp. 73–83.
[31] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *Parallel and Distributed Processing Symp.*, March 2007, pp. 1–8.
[32] N. Fisher, M. Bertogna, and S. Baruah, "Resource-locking durations in EDF-scheduled systems," in *Real-Time and Embedded Technology and Applications Symp.*, April 2007, pp. 91–100.
[33] S. K. Baruah, "Resource sharing in EDF-scheduled systems: A closer look," in *Real-Time Systems Symp.*, 2006, pp. 379–387.
[34] F. Zhang and A. Burns, "Analysis of hierarchical EDF pre-emptive scheduling," in *Real-Time Systems Symp.*, Dec. 2007, pp. 423–434.
[35] G. Bernat and A. Burns, "New results on fixed priority aperiodic servers," in *Real-Time Systems Symp.*, Dec. 1999, pp. 68–78.
[36] M. Behnam, T. Nolte, and R. J. Bril, "Bounding the number of self-blocking occurrences of sirap," in *Real-Time Systems Symp.*, Dec. 2010.
[37] P. López Martinez, L. Barros, and J. Drake, "Scheduling configuration of real-time component-based applications," in *Reliable Software Technologiey - Ada-Europe*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6106, pp. 181–195.
[38] Micrium, "RTOS and tools," March 2010. [Online]. Available: http://micrium.com/
[39] A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar, "Rate-harmonized scheduling and its applicability to energy management," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 3, pp. 265–275, Aug. 2010.
[40] R. J. Bril, U. Keskin, T. Nolte, and M. Behnam, "Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited," Eindhoven University of Technology, Tech. Rep. CS 10-05, June 2010, http://alexandria.tue.nl/repository/books/685859.pdf.
[41] A. Burns and S. Baruah, "Sustainability in real-time scheduling," *Journal of Computing Science and Engineering*, vol. 2, no. 1, pp. 72–94, 2008.
[42] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2010, pp. 37–42.
[43] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Real-Time Systems Symp.*, Dec. 1998, pp. 4–13.
[44] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari, "Resource reservations for general purpose applications," *IEEE Trans. on Industrial Informatics*, vol. 5, no. 1, pp. 12–21, Feb. 2009.
[45] M. Bertogna and S. Baruah, "Limited preemption edf scheduling of sporadic task systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 4, pp. 579–591, Nov. 2010.

[46] OpenCores. (2009) OpenRISC overview. [Online]. Available: http://www.opencores.org/project,or1k

[47] M. Lv, N. Guan, Q. Deng, G. Yu, and Y. Wang, "Static worst-case execution time analysis of the μC/OS-II real-time kernel," *Frontiers of Computer Science in China*, vol. 4, no. 1, pp. 17–27, 2010.

[48] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Real-Time Systems Symp.*, Dec. 2003, pp. 25–36.

PAPER E:

Dependable resource sharing for compositional real-time systems

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien

In proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),
pp. 153–163, August 2011

ABSTRACT

Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating temporal isolation between components that need to be integrated on a single shared processor. In the presence of shared resources, however, temporal isolation may break when one of the accessing components executes longer than specified during global resource access. The ability to confine such temporal faults makes the HSF more dependable. As a solution we propose a stack-resource-policy (SRP)-based synchronization protocol for HSFs, named Hierarchical Synchronization protocol with Temporal Protection (HSTP).

When a component exceeds its specified critical-section length, HSTP enforces a component to self-donate its own budget to accelerate the resource release. In addition, a component that blocks on a locked resource may donate budget. The schedulability of those components that are independent of the locked resource is unaffected. HSTP efficiently limits the propagation of temporal faults to resource-sharing components by disabling local preemptions in a component during resource access. We finally show that HSTP is SRP-compliant and applies to existing synchronization protocols for HSFs.

1 Introduction

Many real-time embedded systems implement increasingly complex and safety-critical functionality while their time to market and cost is continuously under pressure. This has resulted in standardized component-based software architectures, e.g. the AUTomotive Open System ARchitecture (AUTOSAR), where each component can be analysed and certified independently of its performance in an integrated system. Hierarchical scheduling frameworks (HSFs) have been investigated as a paradigm for facilitating such a decoupling [1] of development of individual components from their integration. HSFs provide temporal isolation between components by allocating a *budget* to each component. A component that is validated to meet its timing constraints when executing in isolation will therefore continue meeting its timing constraints after integration or admission on a shared uniprocessor platform.

An HSF without further resource sharing is unrealistic, however, since components may for example use operating system services, memory mapped devices and shared communication devices which require mutually exclusive access. Extending an HSF with such support makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary components, in a mutually exclusive manner. A resource that is used in more than one component is denoted as a *global shared resource*. A resource that is only shared by tasks within a single component is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of its budget, excessive blocking periods can occur which may hamper the correct timeliness of other components [2].

To accommodate resource sharing between components, three synchronization protocols [3], [4], [5] have been proposed based on the *stack resource policy* (SRP) [6]. Each of these protocols describes a run-time mechanism to handle the depletion of a component's budget during global resource access. In short, two general approaches are proposed: (*i*) *self-blocking* before accessing a shared resource when the remaining budget is insufficient to complete a critical section [4], [5] or (*ii*) *overrun* the budget until the critical section ends [3]. However, when a task exceeds its specified worst-case critical-section length, i.e. it *misbehaves* during global resource access, temporal isolation between components is no longer guaranteed. The protocols in [3], [4], [5] therefore break the temporal encapsulation and fault-containment properties of an HSF without the presence of complementary protection.

A common practice to temporal protection is based on watchdog timers, which (*i*) trigger the termination of a misbehaving task, (*ii*) release all its locked resources and (*iii*) call an error handler to execute a *roll-back strategy* [7]. For some shared resources, e.g. network devices, it may be advantageous to continue a critical section, because interrupting and resetting a busy device can be time consuming. In addition, an eventual error handler needs to be constrained, so that it does not interfere with independent components.

A solution to confine temporal faults to those components that share global resources is considered in [8]. Each task is assigned a dedicated budget per global resource access and this budget is synchronous with the period of that task. After depletion of this budget, a critical section is discontinued until its budget replenishes and the task will therefore miss its deadline. To improve reliability, they propose a donation mechanism for tasks that encounter a locked resource, so that a critical section can continue and both the blocking as well as the resource-locking task may still meet their deadlines. However, in [8] they allow only a single task per component.

The first problem is to limit the propagation of temporal faults in HSFs, where multiple concurrent tasks are allocated a shared budget, to those components that share a global resource. However, when a critical-section exceeds its specified length, a component may nevertheless have remaining budget apart from the budget assigned to that critical section. The second problem is to allocate these unused budgets to accelerate a resource release before a resource-sharing component blocks on the locked resource. We aim to increase the reliability of resource-sharing components, while preserving the schedulability of independent components, by self-donations of the remaining budget of a resource-locking component itself and, next, by budget donations from others to critical sections.

Contributions

To achieve temporal isolation between components, even when resource-sharing components misbehave, we propose a modified SRP-based synchronization protocol, named *Hierarchical Synchronization protocol with Temporal Protection* (HSTP). It supports fixed-priority as well as earliest-deadline-first (EDF) scheduled systems and it complements existing synchronization protocols [3], [4], [5] for HSFs. We efficiently achieve fault-containment by disabling preemptions of other tasks within the same component during global resource access. Secondly, we allow a cascaded continuation of a critical section via self-donations as long as a component has remaining budget, or via budget donations from dependent components. Thirdly, we present HSTP's analysis and show that sufficiently short critical sections can execute with local preemptions disabled and preserve system schedulability. Finally, we evaluate HSTP's implementation in a real-time operating system.

Organization

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 presents our system model. Section 4 presents our dependable resource-sharing protocol, HSTP, including a donation mechanism that enhances the reliability of inter-dependent components. Section 5 presents HSTP's corresponding schedulability analysis. Section 6 discusses HSTP's implementation and investigates its corresponding system overhead. Finally, Section 7 concludes this paper.

2 Related work

Our basic idea is to use two-level SRP to arbitrate access to global resources, similar as [3], [4], [5]. In literature several alternatives are presented to accommodate task communication in reservation-based systems. De Niz et al. [8] support resource sharing between reservations based on the immediate priority ceiling protocol (IPCP) [9] in their fixed-priority preemptively scheduled (FPPS) Linux/RK resource kernel and use a run-time mechanism based on resource containers [10] for temporal protection against misbehaving tasks. Steinberg et al. [11] showed that these resource containers are expensive and efficiently implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach is described in [12] for EDF-based systems and termed bandwidth-inheritance (BWI). BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol [9], i.e. when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. BWI does not require a-priori knowledge of tasks, i.e. precalculated ceilings are unnecessary. BWI has been extended with the *Clearing Fund Protocol* (CFP) [13], which makes a task pay back its inherited bandwidth, if necessary. All these approaches assume a one-to-one mapping from tasks to budgets, and inherently only have a single scheduling level.

In HSFs a group of concurrent tasks, forming a component, are allocated a budget [14]. A prerequisite to enable independent analysis of interacting components and their integration is the knowledge of which resources a task will access [4], [5], [15]. When a task accesses a global shared resource, one needs to consider the priority inversion between components as well as local priority inversion between tasks within the component. To *prevent budget depletion* during global resource access, three synchronization protocols have been proposed based on SRP [6], i.e. HSRP [3], SIRAP [4] and BROE [5]. Although HSRP [3] originally does not integrate into HSFs due to the lacking support for independent analysis of components, Behnam et al. [15] lifted this limitation. However, these three protocols, including their implementations in [16], [17], assume that components respect their timing contract with respect to global resource sharing. In this paper we smooth and limit the unpredictable interferences caused by contract violations to the components that share the global resource.

3 Real-time scheduling model

We consider a two-level HSF using the periodic resource model [1] to specify guaranteed processor allocations to components. The global scheduler and each individual component may apply a different scheduling algorithm. As scheduling algorithms we consider EDF, an optimal dynamic uniprocessor scheduling algorithm, and the deadline-monotonic (DM) algorithm, an optimal fixed-priority uniprocessor scheduling algorithm. We use an SRP-based synchronization protocol to arbitrate mutually exclusive access to global shared resources.

3.1 Compositional model

A system contains a set $\mathcal{R}$ of M global logical resources R_1, R_2, ..., R_M, a set $\mathcal{C}$ of N components C_1, C_2, ..., C_N, a set $\mathcal{B}$ of N budgets for which we assume a periodic resource model [1], and a single shared processor. Each component C_s has a dedicated budget which specifies its periodically guaranteed fraction of the processor. The remainder of this paper leaves budgets implicit, i.e. the timing characteristics of budgets are taken care of in the description of components.

The timing characteristics of a component C_s are specified by means of a triple $< P_s, Q_s, \mathcal{X}_s >$, where $P_s \in \mathbb{R}^+$ denotes its period, $Q_s \in \mathbb{R}^+$ its budget, and $\mathcal{X}_s$ the set of maximum access times to global resources. The maximum value in $\mathcal{X}_s$ is denoted by X_s, where $0 < Q_s + X_s \leq P_s$. The set $\mathcal{R}_s$ denotes the subset of M_s global resources accessed by component C_s. The maximum time that a component C_s executes while accessing resource $R_l \in \mathcal{R}_s$ is denoted by X_{sl}, where $X_{sl} \in \mathbb{R}^+ \cup \{0\}$ and $X_{sl} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$.

3.1.1 Processor supply

The *processor supply* refers to the amount of processor allocation that a component C_s can provide to its workload. The supply bound function $\mathtt{sbf}_{\Gamma_s}(t)$ of the periodic resource model $\Gamma_s(P_s, Q_s)$, that computes the minimum supply for any interval of length t, is given by [1]:

$$\mathtt{sbf}_{\Gamma_s}(t) = \begin{cases} t - (k+1)(P_s - Q_s) & \text{if } t \in V^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{1}$$

where $k = \max\left(\lceil (t - (P_s - Q_s))/P_s \rceil, 1\right)$ and $V^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$. The longest interval a component may receive no processor supply is named the *blackout duration*, BD_s, i.e. $BD_s = 2(P_s - Q_s)$.

3.1.2 Task model

Each component C_s contains a set $\mathcal{T}_s$ of n_s sporadic tasks τ_{s1}, τ_{s2}, …, τ_{sn_s}. Timing characteristics of a task $\tau_{si} \in \mathcal{T}_s$ are specified by means of a triple $< T_{si}, E_{si}, D_{si} >$, where $T_{si} \in \mathbb{R}^+$ denotes its minimum inter-arrival time, $E_{si} \in \mathbb{R}^+$ its worst-case computation time, $D_{si} \in \mathbb{R}^+$ its (relative) deadline, where $0 < E_{si} \leq D_{si} \leq T_{si}$. We assume that period P_s of component C_s is selected such that $2P_s \leq T_{si}(\forall \tau_{si} \in \mathcal{T}_s)$, because this efficiently assigns a budget to component C_s [1]. The worst-case execution time of task τ_{si} within a critical section accessing R_l is denoted c_{sil}, where $c_{sil} \in \mathbb{R}^+ \cup \{0\}$, $E_{si} \geq c_{sil}$ and $c_{sil} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$. For notational convenience we assume that tasks (and components) are given in deadline-monotonic order, i.e. τ_{s1} has the smallest deadline and τ_{sn_s} the largest.

3.2 Synchronization protocol

Traditional synchronization protocols such as PCP [9] and SRP [6] can be used for *local* resource sharing in HSFs [18]. This paper focuses on arbitrating *global* shared resources using SRP. To be able to use SRP in an HSF for synchronizing global resources, its associated ceiling terms need to be extended and excessive blocking must be prevented.

3.2.1 Preemption levels

Each task τ_{si} has a static preemption level equal to $\pi_{si} = 1/D_{si}$. Similarly, a component has a preemption level equal to $\Pi_s = 1/P_s$, where period P_s serves as a relative deadline. If components (or tasks) have the same calculated preemption level, then the smallest index determines the highest preemption level.

3.2.2 Resource ceilings

With every global resource R_l two types of resource ceilings are associated; a *global* resource ceiling RC_l for global scheduling and a *local* resource ceiling rc_{sl} for local scheduling. These ceilings are statically calculated values, which are defined as the highest preemption level of any component or task that shares the resource. According to SRP, these ceilings are defined as:

$$\begin{aligned} RC_l &= \max(\Pi_N, \max\{\Pi_s \mid R_l \in \mathcal{R}_s\}), & (2) \\ rc_{sl} &= \max(\pi_{sn_s}, \max\{\pi_{si} \mid c_{sil} > 0\}). & (3) \end{aligned}$$

We use the outermost max in (2) and (3) to define RC_l and rc_{sl} in those situations where no component or task uses R_l.

3.2.3 System and component ceilings

The system and component ceilings are dynamic parameters that change during execution. The system ceiling is equal to the highest global resource ceiling of a currently locked resource in the system. Similarly, the component ceiling is equal to the highest local resource ceiling of a currently locked resource within a component. Under SRP a task can only preempt the currently executing task if its preemption level is higher than its component ceiling. A similar condition for preemption holds for components.

3.2.4 Prevent excessive blocking

HSRP [3] uses an overrun mechanism [15] when a budget depletes during a critical section. If a task $\tau_{si} \in \mathcal{T}_s$ has locked a global resource when its component's budget Q_s depletes, then component C_s can continue its execution until task τ_{si} releases the resource. To distinguish this additional amount of required budget from the *normal budget* Q_s, we refer to X_s as an *overrun budget*. These budget overruns cannot take place across replenishment boundaries, i.e. the analysis guarantees $Q_s + X_s$ processor time before the relative deadline P_s of component C_s [3], [15].

SIRAP [4] uses a self-blocking approach to prevent budget depletion inside a critical section. If a task τ_{si} wants to enter a critical section, it enters the critical section at the earliest time instant so that it can complete the critical section before the component's budget depletes. If the remaining budget is insufficient to lock and release a resource R_l before depletion, then (*i*) the task blocks itself until budget replenishment and (*ii*) the component ceiling is raised to prevent tasks $\tau_{sj} \in \mathcal{T}_s$ with a preemption level lower than the local ceiling rc_{sl} to execute until the requested critical section has been finished.

BROE [5] uses an other self-blocking variant than SIRAP uses. Contrary to SIRAP and HSRP, BROE only works with a global EDF scheduler. Its major advantage is that when the remaining budget of a component is insufficient to complete a critical section, it discards this remainder without violating the periodic resource supply [1]. BROE does therefore not waste processor resources during self-blocking and it is also unnecessary to allocate overrun budgets to components.

In this paper we ignore the relative strengths of the mechanisms presented by the protocols in [3], [4], [5], [15]. We focus on mechanisms to extend these existing protocols with dependability attributes and merely investigate their relative complexity with respect to our mechanisms.

4 Dependable resource sharing

Dependability encompasses many quality attributes [7], i.e. amongst others: safety, integrity, reliability, availability and robustness. Temporal faults can have catastrophic consequences, making the system *unsafe*. These temporal faults may cause improper system alterations, e.g. due to unexpectedly long blocking or an inconsistent state of a resource. Hence, it affects system *integrity*. Without any protection a self-blocking approach [4], [5] may miss its purpose under erroneous circumstances, i.e. a task still overruns its budget to complete its critical section. Even an overrun approach [3], [15] must guarantee a maximum duration of the overrun situation. Otherwise, overruns can hamper temporal isolation and resource *availability* to other components due to unpredictable blocking effects. A straightforward implementation of the overrun mechanism, e.g. as implemented in the ERIKA kernel [19], where a task is allowed to indefinitely overrun its budget as long as it locks a resource, is therefore *unreliable*. The extent to which a system tolerates such unforeseen interferences defines its *robustness*.

4.1 Resource monitoring and enforcement

A common approach to ensure temporal isolation and prevent propagation of temporal faults within the system is to group tasks that share resources into a single component [18]. However, this might be too restrictive and leading to large, incoherent component designs, which violates the principle of HSFs to independently develop components. Since a component defines a coherent piece of functionality, a task that accesses a global shared resource is critical with respect to all other tasks in the same component.

To guarantee temporal isolation between components, the system must *monitor* and *enforce* the length of a global critical section to prevent a malicious task to execute longer in a critical section than assumed during system analysis [8]. Otherwise such a misbehaving task may increase blocking to components with a higher preemption level, so that even independent components may suffer, as shown in Figure 1.

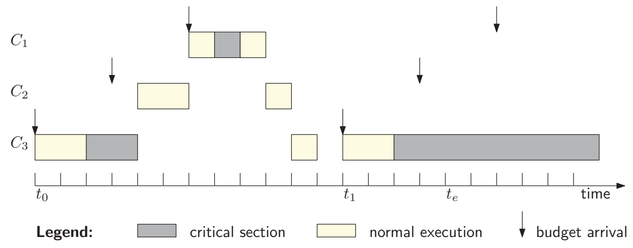

Fig. 1. Temporal isolation is unassured when a component, C_3, exceeds its specified critical-section length, i.e. at time instant t_e. The system ceiling, defined by resource ceiling $RC_l = \Pi_1$, blocks all other components.

To prevent this effect we introduce a *resource-access budget* q_s in addition to a component's budget Q_s, where budget q_s is used to enforce critical-section lengths. When a resource R_l gets locked, q_s replenishes to its full capacity, i.e. $q_s \leftarrow X_{sl}$. To monitor the available budget at any moment in time, we assume the availability of a function $Q_s^{\texttt{rem}}(t)$ that returns the remaining budget of Q_s. Similarly, $q_s^{\texttt{rem}}(t)$ returns the remainder of q_s at time t. If a component C_s executes in a critical section, then it consumes budget from Q_s and q_s in parallel, i.e. depletion of either Q_s or q_s forbids component C_s to continue its execution. We maintain the following invariant to prevent budget depletion during resource access:

Invariant 1: $Q_s^{\texttt{rem}}(t) \geq q_s^{\texttt{rem}}(t)$.

The way of maintaining this invariant depends on the chosen policy to prevent budget depletion during global resource access, e.g. by means of SIRAP [4], HSRP [3] or BROE [5]. For ease of presentation we will first complement HSRP's overrun mechanism with a mechanism for temporal protection. In Section 5.4 we show how our resulting protocol, HSTP, applies to both SIRAP's and BROE's self-blocking mechanisms.

4.1.1 Fault containment of critical sections

Existing SRP-based synchronization protocols in [4], [5], [15] make it possible to choose the local resource ceilings, rc_{sl}, according to SRP [6], see (3). In [20], [21] techniques are presented to trade-off preemptiveness against resource holding times. Given their common definition for local resource ceilings, a resource holding time, X_{sl}, may also include the interference of tasks with a preemption level higher than the resource ceiling. Task τ_{si} can therefore lock resource R_l longer than specified, because an interfering task τ_{sj} (where $\pi_{sj} > rc_{sl}$) exceeds its computation time, E_{sj}.

To prevent this effect we choose to disable preemptions for other tasks within the same component during critical sections, i.e. similar as HSRP [3] we choose local resource ceilings by $\forall R_l \in \mathcal{R}_s : rc_{sl} = \pi_{s_1}$. As a result X_{sl} only comprises task execution times within a critical section, i.e.

$$X_{sl} = \max_{1 \leq i \leq n_s} c_{sil}. \tag{4}$$

Since X_{sl} is enforced by budget q_s, temporal faults are contained within a subset of resource-sharing components.

4.1.2 Maintaining SRP ceilings

To enforce that a task τ_{si} resides no longer in a critical section than specified by X_{sl}, a resource $R_l \in \mathcal{R}$ maintains a state *locked* or *free*. We introduce an extra state *busy* to signify that R_l is locked by a misbehaving task. When a task τ_{si} tries to exceed its maximum critical-section length X_{sl}, we update SRP's *system* *ceiling* by mimicking a resource unlock and mark the resource *busy* until it is released. Since we decrease the system ceiling after τ_{si} executes for a duration of X_{sl} in a critical section to resource R_l, we can no longer guarantee absence of deadlocks. Nested critical sections to global shared resources are therefore unsupported[1]. One may alternatively aggregate global resource accesses into a simultaneous lock and unlock of a single artificial resource [22]. Many protocols, or their implementations, lack deadlock avoidance [8], [11], [12], [17].

Although it seems attractive from a schedulability point of view to release the *component ceiling* when the critical-section length is exceeded, i.e. similar to the system ceiling, this would break SRP compliance, because a task may block on a busy resource instead of being prevented from starting its execution. Our approach therefore preserves the SRP property to share a single, consistent execution stack per component [6]. At the global level tasks can be blocked by a depleted budget, so that components cannot share an execution stack anyway.

4.1.3 Repetitive self-donation to resource-access budgets

A component that accesses a resource R_l maintains a dedicated resource-access budget q_s synchronous to its normal budget Q_s, i.e. similar to [8]. Since critical sections are often shorter than budget Q_s is, a component may still have remaining budget $Q_s^{\mathtt{rem}}(t) > 0$ when q_s depletes. Contrary to [8], when q_s depletes we repeatedly replenish it by a self-donation of X_{sl} until budget Q_s is exhausted. Although we need to *decrease the system ceiling before replenishing* q_s to avoid excessive blocking durations to other components, we may again increase the system ceiling for a duration of X_{sl} as soon as component C_s is selected by the global scheduler to continue its execution. This increases the likelihood that a resource-using component meets its deadline despite a misbehaving critical section and it reduces the likelihood that a resource-sharing component encounters a busy resource. Our resource-access budgets are therefore more reliable and robust than those in [8] are.

4.2 HSTP with self-donation

We specify four rules to manipulate a component's budget q_s when a resource is locked or unlocked and change the way budgets Q_s and q_s are replenished and depleted. This scheme defines HSTP, a protocol that maintains temporal protection between components during global resource access.

4.2.1 Lock resource

Upon an attempt of task $\tau_{si} \in \mathcal{T}_s$ to lock resource $R_l \in \mathcal{R}_s$ at time t_l, we raise the component ceiling independent of whether or not R_l is free.

1. We allow nesting of a (sequence of) global resource access(es) inside local critical sections.

If resource R_l is *free*, then τ_{si} locks the resource and q_s replenishes, i.e. $q_s \leftarrow X_{sl}$. Component C_s now runs in parallel on budget Q_s and q_s. We need to guarantee that C_s' normal budget Q_s does not deplete during global resource access. We therefore first save C_s' remaining budget as $Q_s^{\nabla} \leftarrow Q_s^{\texttt{rem}}(t_l)$ and then provide an overrun budget $Q_s^{\texttt{rem}}(t_l) \leftarrow X_{sl}$.

By virtue of SRP, a resource R_l can never be *locked* when a task τ_{si} attempts to lock it. If there exists a task $\tau_{uj} \in \mathcal{T}_u$ which currently holds R_l *busy*, then we immediately deplete the remaining budget of C_s, i.e. $Q_s^{\texttt{rem}}(t_l) \leftarrow 0$. After the budget Q_s of component C_s has replenished, the blocked task τ_{si} again tests whether or not resource R_l is free.

4.2.2 Unlock resource

If task $\tau_{si} \in \mathcal{T}_s$ unlocks a resource $R_l \in \mathcal{R}_s$ at time t_f, then the component and system ceilings are decreased according to the rules of SRP and resource R_l is marked *free*. Moreover, component C_s no longer consumes budget q_s and we need to restore C_s' budget. When a budget overrun has occurred, i.e. $Q_s^{\nabla} < X_{sl} - q_s^{\texttt{rem}}(t_f)$, then component C_s is suspended on its depleted budget Q_s. Otherwise, component C_s continues within its remaining budget, i.e. we restore Q_s with $\max(0, Q_s^{\nabla} - (X_{sl} - q_s^{\texttt{rem}}(t_f)))$.

4.2.3 Budget depletion

If component C_s' budget Q_s or q_s depletes and all resources $R_l \in \mathcal{R}_s$ are free, then nothing changes compared to default budget-depletion policies. However, if a task τ_{si} holds a resource $R_l \in \mathcal{R}_s$ so that q_s has been depleted, then (*i*) resource R_l is marked *busy*; (*ii*) the *system ceiling* is decreased according to the rules of unlocking an SRP resource and (*iii*) the budget Q_s of component C_s is restored with $\max(0, Q_s^{\nabla} - X_{sl})$.

These actions guarantee that τ_{si} can overrun at most an amount X_{sl} and a component C_s can therefore at most request $Q_s + X_s$ processor time during each period P_s. If no other component C_t can preempt based on its preemption level $\Pi_t > \Pi_s$ after decreasing the system ceiling, the system ceiling can be raised again for a duration of X_{sl} without increasing the blocking times that component C_t may experience. Since the component ceiling is persistently raised during global resource access, other tasks in $\mathcal{T}_s$ than the resource-accessing one cannot execute. Hence, as long as a resource R_l is kept locked or busy by component C_s, task τ_{si} may entirely consume the remaining budget $Q_s^{\texttt{rem}}(t)$ with a raised component and system ceiling via self-donations, *if* it decreases the system ceiling after every X_{sl} and performs a global preemption test.

4.2.4 Budget replenishment

If component C_s' budget Q_s replenishes, it is guaranteed that at most one resource $R_l \in \mathcal{R}_s$ is kept *busy* via component C_s and the corresponding resource-access budget $q_s^{\texttt{rem}}(t) = 0$. If component C_s does not keep any resource busy, then nothing changes compared to default budget-replenishment policies. Otherwise, we restrict the replenishment of component C_s, so that it may continue its critical section, i.e (*i*) the budget to be restored upon unlocking, $Q_s^{\triangledown}$, replenishes to Q_s; (*ii*) the budgets q_s and Q_s that allow continuing resource access to R_l replenish with X_{sl} and (*iii*) a resource lock is mimicked by raising the system ceiling according to SRP. Hence, component C_s continues its execution with a raised system ceiling.

4.2.5 Final remarks

A component C_u may block on a busy resource R_l at a time t_b and subsequently R_l can become free at a time t_f before the replenishment of C_u's budget, Q_u. If we signal component C_u to continue its execution within $Q_u^{\texttt{rem}}(t_b)$ at time t_f, we can no longer guarantee the provisioning of budget Q_u within its period boundaries P_u due to the self-suspension interval $[t_b, t_f]$ of budget Q_u. This introduces unpredictable interferences and scheduling anomalies to other components in the system [23]. We can therefore either let a blocking task spinlock on a busy resource, so that it consumes its component's budget, or suspend a blocking component until its replenishment before allowing this component to resume its execution, similar to [8].

4.3 HSTP extended with third-party donations

The basic HSTP specification in Section 4.2 immediately depletes the budget of a component C_u upon an attempt at time t_b to lock a busy resource R_l. As a result, all remaining budget, $Q_u^{\texttt{rem}}(t_b)$, of the blocking component C_u is discarded until its next replenishment. Since C_u is unable to consume its budget due to the raised component ceiling, C_u may alternatively donate its budget to the misbehaving component C_s with the aim to reduce the waiting time on the busy resource R_l. In this section we complement HSTP with a donation mechanism.

4.3.1 Attempt to lock a busy resource

When component C_u encounters a busy resource at time t_b, we can repeatedly donate X_{ul} to misbehaving component C_s until Q_u is depleted with the aim to reduce its waiting time on resource R_l. Such a donation from donor C_u to donatee C_s (*i*) mimics a resource lock by raising the system ceiling, (*ii*) saves donor C_u's remaining budget as $Q_u^{\triangledown} \leftarrow Q_u^{\texttt{rem}}(t_b)$, (*iii*) saves donatee C_s' remaining budget as $Q_s^{\triangledown} \leftarrow Q_s^{\texttt{rem}}(t_b)$ and (*iv*) allocates budget $Q_s^{\texttt{rem}}(t_b), q_s \leftarrow X_{ul}$ to the critical section of component C_s. Due to the raised system ceiling, component C_s effectively runs at the resource-ceiling's preemption level RC_l. It is therefore unnecessary to change the deadline or priority of the donatee, because all components that may use R_l are blocked by the system ceiling.

4.3.2 Unlock resource

If an erroneous component C_s unlocks resource R_l at time t_f while consuming a donation, then any remaining resource-access budget $q_s^{\mathtt{rem}}(t_f)$ is donated back to donor C_u, i.e. $Q_u^{\mathtt{rem}}(t_f) \leftarrow \max(0, Q_u^{\nabla} - (X_{ul} - q_s^{\mathtt{rem}}(t_f)))$. Donatee C_s has consumed a part of the donated budget, $X_{ul} - q_s^{\mathtt{rem}}(t_f)$, in the place of donor C_u. This part is accounted to donor C_u rather than to donatee C_s, so that the budget of donatee C_s is restored to $Q_s^{\mathtt{rem}}(t_f) \leftarrow Q_s^{\nabla}$. As a result both resource-sharing components C_s and C_u may resume execution within their restored budgets.

4.3.3 Budget depletion

When a donated budget X_{ul} is depleted by donatee C_s, the system ceiling is decreased and X_{ul} is subtracted from donor C_u's budget, i.e. $Q_u^{\mathtt{rem}}(t) \leftarrow \max(0, Q_u^{\nabla} - X_{ul})$. The budget of donatee C_s is restored with its original value $Q_s^{\mathtt{rem}}(t) \leftarrow Q_s^{\nabla}$. If donor C_u gets re-selected by the global scheduler for execution and resource R_l is still busy, it may again donate X_{ul} to component C_s.

4.3.4 Budget replenishment

During the consumption of donated budget X_{ul} from donor C_u, the budget of donatee C_s itself can get replenished. The replenishment of budget Q_s remains unchanged compared to Section 4.2.4. However, we can only replenish a depleted resource-access budget, i.e. $q_s^{\mathtt{rem}}(t) = 0$. Otherwise, component C_s may cause double blocking to other components, i.e. $X_{ul} + X_{sl}$ instead of X_{sl}, see Figure 2. How to avoid this effect is unclearly described in [8], however. Because it is unattractive to account for double blocking in the system analysis, we avoid this blocking.

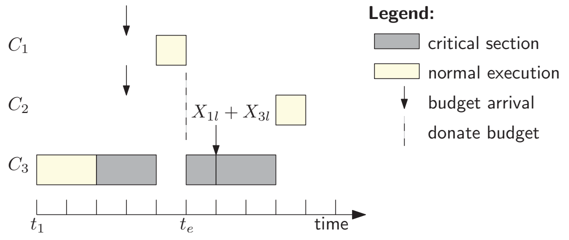

Fig. 2. If resource-access budget q_3 gets replenished and a task $\tau_{3i} \in \mathcal{T}_3$ executes in a critical section, then component C_3 may double block other components.

The key to the solution to avoid the double-blocking problem is to allow preemption *before* replenishing q_s. We can either choose to immediately deplete $q_s^{\mathtt{rem}}(t) > 0$ or we can defer replenishment until q_s depletes. Note that the first alternative is less reliable, because the donor will not receive any of its donation back. However, in both cases the system ceiling must be decreased according to SRP's rules and the global scheduler must be called *prior to* replenishing q_s.

4.4 Donation policies

When a component depletes its resource-access budget or blocks on a *busy* resource, it cannot continue its execution. We allow two alternative budget-donation policies, symmetrically for a donatee C_s via self-donation or a third-party donor C_u:

4.4.1 At-once donation

A component C_u may (self-)donate its remaining budget $Q_u^{\mathtt{rem}}(t)$ at once to a donatee C_s using BWI [8], [12]. However, component C_s *must* consume such donations *in the place of* donor C_u [24], i.e. at the donor's preemption-level rather than at the resource-ceiling level. This requires multiple budgets per component and migration of tasks to enable the execution of tasks over several budgets [8], which is costly [11] and breaks SRP's stack-sharing property. Hence, at-once donation causes additional run-time penalties compared to a regular two-level HSF implementation.

4.4.2 Repetitive donation

Similar to the misbehaving component C_s, the blocking component C_u may entirely donate its remaining budget $Q_u^{\mathtt{rem}}(t)$ with a raised system ceiling *if* it decreases the system ceiling after every X_{sl} and performs a preemption test to avoid double blocking. As a result, after a preemption test the component with the highest preemption level, which is ready to execute, resumes its execution, so that a donatee always receives donated budget from the resource-dependent component with the highest preemption level.

Similar to *at-once donation*, however, a component C_u may (repetitively) donate its budget to a component C_s with a depleted resource-access budget q_s, although donatee C_s has remaining budget $Q_s^{\mathtt{rem}}(t) > 0$. An advantage of *repetitive self-donations* is that it reduces the number of such unnecessary donations to third-parties, because a replenished component can be prevented by the system ceiling from starting its execution and donation for a duration of X_{sl}; see Figure 3.

Given the repetitive donation policy, the following lemma follows from our HSTP specification:

Lemma 1: At each time instant t there is at most one donor component C_u per donatee C_s and each component C_s can only execute a single global critical section.

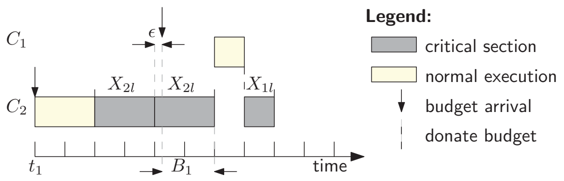

Fig. 3. Repetitive consumption of remaining budget can reduce unnecessary donations from blocking components. In this example component C_1 arrives an infinitesimal amount of time, ϵ, after the first depletion and preemption test of C_2, so that donation is deferred until component C_2 depletes its replenished budget $q_2 = X_{2l}$.

Proof: As long as a resource R_l is kept locked or busy by component C_s, its component ceiling prevents all other tasks within the component from starting their execution. By absence of nested critical sections, no other resource R_k can be kept locked or busy by the same component C_s.

A component C_u which attempts to lock R_l may donate a budget of length X_{ul} and immediately raises the system ceiling. Hence, other R_l-sharing components C_t cannot preempt while donatee C_s consumes the donated budget X_{ul}. □

In the remainder of this paper we use the repetitive-donation policy, because it eliminates donations at arbitrary moments in time, which increases the reliability of a system. Secondly, it eliminates transitive donations, so that we need to track at most a single donor component within the context of a donatee.

5 Dependable and compositional analysis

We presented a synchronization protocol, HSTP, which enables a dependable execution of components that exceed their specified resource holding times. Contrary to [8], a mechanism to monitor blocking times is unnecessary, because we will show that HSTP complies to SRP's blocking times. To make HSTP applicable in HSFs we reuse the analysis results presented in [4], [15], so that we obtain independent analysis for individual components and their integration. To summarize, the HSTP specification implies the following *system invariant*:

Invariant 2: If component C_s executes on the processor and holds resource R_l (locked or busy), then either (*i*) the system ceiling is raised to RC_l or (*ii*) C_s has the highest preemption level $\Pi_s > \Pi_u$ among all components C_u that share R_l and are ready to execute (i.e. $q_u^{\mathtt{rem}}(t) > 0$).

Lemma 2: The HSTP specification in Section 4.2 and Section 4.3 implies Invariant 2.

Proof: If component C_s executes and keeps resource R_l *locked* or *busy*, then $q_s^{\mathtt{rem}}(t) > 0$ and the system ceiling equals RC_l. Hence, no R_l-sharing component can preempt.

Similarly, if component C_s received X_{ul} from a donor C_u, then C_s executes with a raised system ceiling (RC_l) and keeps resource R_l *busy*. For donor C_u holds: $q_u = 0$, so that it is suspended and other R_l-sharing components cannot preempt.

Hence, component C_s has the highest preemption level among all ready components that share R_l. □

5.1 SRP-compliant blocking

When a task exceeds its maximum duration inside a critical section, only components that are involved in the interaction are affected. Although we decrease SRP's system ceiling when a task exceeds its specified critical-section length, HSTP has the same calculation of the global blocking terms as SRP.

Theorem 1: Invariant 2 guarantees that a component C_s does not suffer more blocking or interference on unused resources R_l, i.e. $R_l \notin \mathcal{R}_s$, compared to SRP.

Proof: As long as the *busy* state of a resource R_l is unreached, i.e. R_l is either *free* or *locked*, our protocol strictly follows SRP. Given Lemma 1, we need to consider only a single busy resource R_l for each component in the system.

Consider an erroneous component C_s that exceeds its worst-case critical-section length X_{sl} for resource R_l. Component C_s may receive a donation from C_w with a lower preemption level $\Pi_w < \Pi_s$ or component C_h with a higher preemption level $\Pi_h > \Pi_s$. We can therefore have independent components C_t with $R_l \notin \mathcal{R}_t$ at four different preemption levels:

1) C_t with $\Pi_t > \Pi_h > \Pi_s > \Pi_w$;
2) C_t with $\Pi_h > \Pi_t > \Pi_s > \Pi_w$ blocked by C_w or C_s and preempted by C_h;
3) C_t with $\Pi_h > \Pi_s > \Pi_t > \Pi_w$ blocked by C_w and preempted by C_s and C_h;
4) C_t with $\Pi_h > \Pi_s > \Pi_w > \Pi_t$ preempted by C_w, C_s and C_h.

Without loss of generality, we may restrict ourselves to an artificial system comprising each of these cases, i.e. a system $C_1, \ldots, C_7 \in \mathcal{C}$ with a single shared resource $R_l \in \mathcal{R}_2 \cap \mathcal{R}_4 \cap \mathcal{R}_6$ and $R_l \notin \mathcal{R}_1 \cup \mathcal{R}_3 \cup \mathcal{R}_5 \cup \mathcal{R}_7$ and resource ceiling $RC_l = \Pi_2$. Furthermore, C_4 keeps R_l busy after it has executed for X_{4l} with a raised system ceiling. We now prove the theorem by contradiction, i.e. assume there exists an independent component C_1, C_3, C_5 or C_7 that can experience more than one blocking occurrence and additional interference in the presence of HSTP while this cannot happen with SRP.

Because C_1 has $\Pi_1 > RC_l$, it cannot be blocked by (or suffer interference from) any R_l-sharing component (case 1).

After the system ceiling is decreased, component C_2 can preempt according to its preemption level, $\Pi_2 > \Pi_4$. If component C_2 tries to access R_l, it may donate at most X_{2l} to component C_4. This is already accounted as interference for $C_3 \dots C_7$. Hence, C_3, C_5 and C_7 do not suffer more blocking or more interference from C_2 (case 2).

If component C_6 tries to access R_l it may donate at most X_{6l} to C_4, which will block C_3 and C_5 no longer than X_{6l}. This donation, X_{6l}, is accounted as interference for C_7 and blocking for $C_2 \dots C_5$. Hence, C_3, C_5 and C_7 do not suffer more blocking or more interference from C_6 (case 3).

Although component C_4 may consume the remainder of its budget Q_4^{rem} while it keeps R_l busy, the interference for component C_7 remains the same compared to SRP (case 4).

By contradiction and using Lemma 2 we conclude that HSTP preserves SRP's blocking properties. □

Since Theorem 1 has shown that HSTP complies to SRP's blocking term, we can *safely* reuse existing global analysis for component integration [1], [4], [15] without the implicit assumption that tasks behave according to their contract.

5.2 Global analysis under temporal protection

In line with our specification, we first first present the global analysis of components based on HSTP and an overrun mechanism. In Section 5.4 we adapt this analysis for SIRAP and BROE which each apply a self-blocking mechanism.

The following sufficient schedulability condition holds for global EDF-based systems [25]:

$$\forall t > 0 \quad : \quad B(t) + \mathtt{dbf}_{\mathtt{EDF}}(t) \leq t. \tag{5}$$

The blocking term, $B(t)$, is defined in [25] according to SRP. The demand bound function $\mathtt{dbf}_{\mathtt{EDF}}(t)$ computes the total processor demand of all components in the system for every time interval of length t, i.e.

$$\mathtt{dbf}_{\mathtt{EDF}}(t) \quad = \quad \sum_{C_s \in \mathcal{C}} \left\lfloor \frac{t}{P_s} \right\rfloor (Q_s + O_s(t)) \tag{6}$$

A component C_s, using an overrun mechanism to prevent budget depletion during global resource access, demands $O_s(t)$ more resources in its worst-case scenario [15], where

$$O_s(t) = \begin{cases} X_s & \text{if } t \geq P_s \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

For global FPPS the following sufficient condition holds:

$$\forall 1 \leq s \leq N : \quad \exists t \in [0, P_s] : \quad B_s + \mathtt{dbf}_{\mathtt{DM}}(t, s) \leq t, \tag{8}$$

where the blocking term, B_s, is defined as in [6] and $\mathtt{dbf}_{\mathtt{DM}}(t, s)$ denotes the worst-case cumulative processor request of C_s for a time interval of length t [1], i.e.

$$\mathtt{dbf}_{\mathtt{DM}}(t, s) \quad = \quad Q_s + O_s + \sum_{1 \le r < s} \left\lceil \frac{t}{P_r} \right\rceil (Q_r + O_r). \tag{9}$$

A component C_s, using an overrun mechanism, demands $O_s = X_s$ more resources in its worst-case scenario [15].

5.3 Local analysis for donating components

By filling in task characteristics in the $\mathtt{dbf}$ of (5) and (8) and replacing their right-hand sides by (1), i.e. replace t for $\mathtt{sbf}_{\Gamma_s}(t)$, the same schedulability analysis holds for tasks within a component as for components at the global level. The processor supply to a component, as specified by (1), is only affected when the component blocks on a busy resource.

Inherent to HSFs, however, the local schedulability of tasks in $\mathcal{T}_s$ is only guaranteed when all tasks $\tau_{si} \in \mathcal{T}_s$ respect their timing characteristics. For example, when a task τ_{si} exceeds its worst-case execution time, all other tasks in $\mathcal{T}_s$ may miss their deadline, even when tasks are independent. A similar argument applies when a task shares mutually non-preemptive resources, i.e. a task may unpredictably block for an unbounded duration on a busy resource. It may be useful for a task to test whether a resource is *busy*, so that a task can *compensate* by providing a reduced functionality [7].

If a task τ_{si} in component C_s exceeds its specified critical-section length with an amount c'_{sil}, then its finishing time is potentially delayed. This can be modeled as extra interference of c'_{sil} processor time for task τ_{si} as well as all other tasks in the same component, $\tau_{sj} \in \mathcal{T}_s$. For a donor C_u the situation is symmetrical, i.e. all tasks $\tau_{uj} \in \mathcal{T}_u$ of donor C_u experience c'_{sil} extra interference. Since the local cost of consuming a donation is the same as the local cost of a donation itself, we can consider the longest duration of c'_{sil} for any component $C_s \in \mathcal{C}$ independently, so that all tasks $\tau_{si} \in \mathcal{T}_s$ can still make their deadline. However, a corresponding allocation of budgets to components while maximizing the system robustness is left as a future work.

5.4 HSTP and self-blocking

HSTP also applies to SIRAP and BROE, which cancels the allocation of overrun budgets in (6) and (9), i.e. $O_s(t) = O_s = 0$. When the budget is insufficient to complete a critical section, both protocols postpone the execution of the critical section until sufficient budget $Q_s^{\mathtt{rem}}(t) \ge X_{sl}$ is guaranteed. This self-blocking condition also applies for donations to third-parties, i.e. we can only *donate budget to others* when there is sufficient budget to donate. This gives the misbehaving component C_s the opportunity to resolve its own malicious behaviour until component C_u has sufficient resources and gets selected for execution.

Consider a misbehaving component C_s that accesses resource R_l. When component C_s needs more processor time to complete its critical section than specified by X_{sl}, we can repetitively *self-donate* any remaining budget $Q_s^{\texttt{rem}}(t) > 0$ to resource-access budget q_s. It is unnecessary to check for sufficient budget before a self-donation. To prevent budget overruns, however, we constrain a self-donation by only providing $Q_s^{\texttt{rem}}(t) \leq X_{sl}$, i.e. $q_s \leftarrow \min(Q_s^{\texttt{rem}}(t), X_{sl})$.

Assume that a resource-sharing component C_u attempts to lock the *busy* resource R_l at time t_b. Before donating a budget X_{ul} to component C_s, a self-blocking mechanism applies (either SIRAP or BROE). The first time instant t_l at which component C_u resumes its execution after budget Q_u has been replenished, is the actual time that a task tries to lock resource R_l. Within time interval $[t_b, t_l]$ the resource may get released. If resource R_l is *busy* at time t_l, then, similar to self-donations, component C_u can repetitively donate budget without further self-blocking to accelerate the resource release. When a component C_u starts donating its budget, however, its tasks will miss deadlines, unless it donates slack time.

5.4.1 An example

Consider a component C_1 serviced by BROE [5], see Figure 4, which self-blocks on its budget Q_1 upon an attempt to lock a *busy* resource R_l at time t_b. Within component C_1 virtual time advances with a rate $\frac{Q_1}{P_1}$. When the absolute time of the current budget $Q_1^{\text{rem}}(t_b)$ reaches the virtual time, i.e. component C_1 was running ahead of absolute time, a replenished budget becomes available at time $t_a = t_{d_k} - \frac{P_1}{Q_1} Q_1^{\text{rem}}(t_b)$ with a deadline $t_{d_{k+1}} = t_a + P_1$. A replenishment can happen even without self-blocking if the component was lagging behind. After we would have donated $Q_1^{\text{rem}}(t_b)$, a replenishment happens at t_{d_k}, but with a larger absolute deadline $t'_{d_{k+1}} = t_{d_k} + P_1$ where $t_{d_k} < t_{d_{k+1}} \leq t'_{d_{k+1}}$.

By refraining a donation at time t_b, potential deadline misses of tasks in $\mathcal{T}_1$ can be prevented. If the busy resource R_l is released before component C_1 continues its execution, it is unnecessary for component C_1 to donate budget.

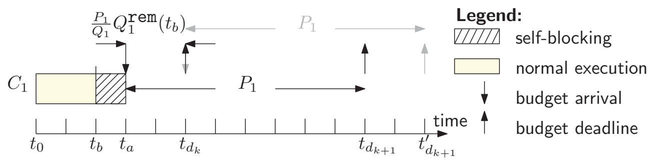

Fig. 4. Donating $Q_1^{\mathrm{rem}}(t_b)$ (in grey) instead of self-blocking (in black) can cause unnecessary deadline misses.

5.4.2 Gain-time provisioning

Contrary to BROE, SIRAP needs to account for an additional self-blocking term in its local demand bound function of DM-scheduled tasks [4], i.e.[2]

$$I_s(i,t) = b_{si} + \sum_{R_l \in \mathcal{R}_s} (c_{sil} + \sum_{1 \le j < i} \left\lceil \frac{t}{T_{sj}} \right\rceil c_{sjl}), \tag{10}$$

where the blocking term, b_{si}, for tasks is defined as in [6]. We can similarly adapt (6) for local EDF-scheduling of tasks that share resources arbitrated by SIRAP. Since we disable local preemptions during global resource access, the allocated processor time to a component for self-blocking is unused. We can make this unused processor time available as *gain time* by also donating $Q_s^{\mathrm{rem}}(t_b) < X_{sl}$, i.e. independent of the self-blocking condition.

5.5 Locally non-preemptive critical sections

Local preemptions during global resource access can decrease the required budget to make a task set $\mathcal{T}_s$ schedulable, but this may on its turn adverse the global schedulability of components due to increased resource holding times. In this paper we are not interested in this schedulability trade-off [20], [21], however, but merely in the containment of temporal faults in critical sections. For this purpose we can introduce an intermediate reservation level assigned and allocated to critical sections in order to enforce that blocking times to other components are not exceeded due to locally preempting tasks [8]. Since this approach causes performance penalties [11], HSTP disables local preemptions during global resource access. In this section we propose an efficient algorithm to check whether off-the-shelf components can be integrated in our dependable resource-sharing framework, which enables a fast design-space exploration during system composition.

We consider a component C_s with a given interface description $< P_s, Q_s, \mathcal{X}_s >$, where resource ceilings are locally and globally configured according to SRP, i.e.

2. For the ease of presentation, Equation 10 assumes that each instant of a task (i.e. a job) accesses a shared resource at most one time.

see (2) and (3). We want to check whether executing the global critical sections of component C_s with local preemptions disabled hampers the schedulability of a task set $\mathcal{T}_s$ for the given periodic resource model $\Gamma_s(P_s, Q_s)$ of component C_s. In many practical cases local preemptions can be disabled temporary, because the budget Q_s assigned to a component is typically pessimistic due to abstraction overheads in its calculation [1]. Each task $\tau_{si} \in \mathcal{T}_s$ may therefore finish its execution before its deadline, so that small finalization delays do not cause a deadline miss.

5.5.1 Delay tolerance

From our assumption $2P_s \leq T_{si}(\forall \tau_{si} \in \mathcal{T}_s)$ we can deduct the following lemma:

Lemma 3: Given $2P_s \leq T_{si}(\forall \tau_{si} \in \mathcal{T}_s)$, all tasks τ_{sj} that are allowed to preempt, can preempt at most once during an access to a global shared resource R_l by task τ_{si}.

Proof: The proof for SIRAP and overrun is presented in [26]. The proof for BROE is presented in [27]. □

Using Lemma 3 we can efficiently calculate how long local preemptions can be disabled without affecting the local schedulability of tasks.

We define the *laxity* δ_{si} of a task τ_{si}, such that τ_{si} finishes its execution at least δ_{si} time units prior to its deadline D_{si} if it has the entire processor at its disposal. For each resource $R_l \in \mathcal{R}_s$ with a local ceiling $rc_{sl} < \pi_{s1}$, we compute the least laxity of all tasks with a preemption level higher than rc_{sl}. The largest common *delay tolerance* of these preempting tasks is:

$$\Delta'_{sl} = (\min \ i : \pi_{si} > rc_{sl} : \delta_{si} = D_{si} - \sum_{1 \leq j \leq i} E_{sj}). \tag{11}$$

Note that each task τ_{sj} can only preempt once and all tasks are ordered according to their preemption level, i.e. the lowest index gives the highest preemption level.

The laxity of each task must at least exceed a single blackout duration of its component, i.e. $\delta_{si} \geq BD_s(\forall \tau_{si} \in \mathcal{T}_s)$, in order to be schedulable with a periodic processor supply of $\Gamma_s(P_s, Q_s)$. An additional blackout is impossible in the processor supply of preempting tasks, because a critical section of length X_{sl} is guaranteed to fit within a single budget provisioning by virtue of the protocols in [3], [4], [5]. Tasks may nevertheless have more delay tolerance than required to bridge the blackout duration. We now need to subtract BD_s from the calculated delay tolerance, i.e.

$$\Delta_{sl} \leftarrow \Delta'_{sl} - BD_s, \tag{12}$$

so that the result is $\Delta_{sl} \geq 0$. This final value Δ_{sl} defines the longest time that a task $\tau_{si} \in \mathcal{T}_s$ that may preempt during resource access to R_l can be deferred without missing any deadline. By construction the following theorem follows:

Theorem 2: Given that a component C_s accessing resource $R_l \in \mathcal{R}_s$ with resource ceiling rc_{sl} is schedulable on a component with parameters $(P_s, Q_s, \mathcal{X}_s)$ using the periodic resource model $\Gamma_s(P_s, Q_s)$: if $c_{sil} \leq \Delta_{sl}$, then component C_s can be scheduled on the same component using $\Gamma_s(P_s, Q_s)$ and can execute each critical section of τ_{si} to R_l with local preemptions disabled, where Δ_{sl} is defined in (12).

Proof: Given Lemma 3, all tasks that may preempt a critical section with resource ceiling rc_{sl} finish within a budget of $Q_s + X_s$ and preempt only once. After increasing the resource ceiling, the total required budget to meet the resource demands of c_{sil} and the execution times of these tasks that now suffer blocking does not increase. We are given two facts: (*i*) each task instance may experience only one blocking occurrence under SRP-based resource arbitration [6] and (*ii*) using the original resource ceiling, each preempting tasks completes its execution at least Δ_{sl} time units prior to its deadline on periodic resource $\Gamma_s(P_s, Q_s)$. Hence, a blocking duration of $c_{sil} \leq \Delta_{sl}$ cannot cause a deadline miss. □

5.5.2 An algorithm

Theorem 2 makes it possible to *exactly determine* whether or not all critical sections within a component C_s can be executed with local preemptions disabled by (*i*) calculating Δ_{sl} using (11) and (12) for each $R_l \in \mathcal{R}_s$ and subsequently (*ii*) apply Theorem 2 on each task to verify $c_{sil} \leq \Delta_{sl}$. This algorithm has a time complexity of $\mathcal{O}(M_s \times n_s)$ for a single component C_s with M_s global shared resources. Note that it only needs to inspect the laxity for interfering tasks, which makes our algorithm much more efficient than the algorithms in [20], [21].

5.5.3 A special case

BROE has a *sufficient test*, i.e. $\Delta_{sl} = \frac{1}{2}BD_s = P_s - Q_s$, for disabling local preemptions of EDF-scheduled task sets [5]. Our algorithm determines *exactly* which critical sections can execute without local preemptions and it applies to any periodic resource model, independent of the local scheduler, with SRP-based resource arbitration.

6 Evaluation

We have recently extended a commercial real-time micro-kernel, μC/OS-II [28], with an HSF and synchronization support by means of SIRAP, HSRP and BROE [16]. In this section we investigate the complexity and overheads of the synchronization primitives of HSTP within our framework.

Because we need to set a budget-expiration timer to track the component's resource-access budget in every lock operation and we need to cancel the same timer in every unlock operation, HSTP primitives are more expensive than a straightforward two-level SRP implementation. If this budget-depletion timer expires during global resource access, i.e. it is not canceled before expiration, then it executes a handler which implements HSTP's unlock policy and marks the resource busy. Based on our measurements, the timer manipulations triple the execution times of the lock and unlock operations compared to the implementations of these primitives in [16]. However, this is the price for preventing the propagation of temporal faults to resource-independent components.

The self-donation mechanism can be easily implemented by extending the global scheduler. Upon a context switch the global scheduler must check whether or not the selected component keeps a resource busy and at the same time has a depleted resource-access budget. This if-statement only takes a few instructions in a reservation-based framework and is therefore relatively cheap compared to the cost of context switching itself. Only when a task within the selected component misbehaves by exceeding its critical-section length, the scheduler executes the expensive operations corresponding to locking a resource, i.e. reset a budget-expiration timer for the selected component, update its normal budget and raise the system ceiling.

Third-party donations can be implemented similarly, but in the lock operation rather than in the global scheduler. If a task attempts to lock a busy resource, the lock operation disables all local preemptions, donates budget by setting a new budget-timer for the donatee and raises the system ceiling. As a result the donor component itself is blocked from continuing its execution. These actions are repeated every time the donor task gets selected by the local scheduler to continue its execution, until the requested resource becomes free.

Since Lemma 1 tells that we only need to keep track of at most a single donor at each time instant, each component maintains a single variable to track its current donor. Upon donation, the donor writes its component identifier into this variable. When it contains a valid identifier when a busy resource is unlocked, a donation back to the donor is executed. The variable is cleared when either the resource is freed or the donated budget is depleted.

7 Conclusion

This paper presented HSTP, an SRP-based synchronization protocol, which provides temporal protection between components in which multiple tasks share a budget, even when interacting components exceed their specified critical-section lengths. Prerequisites to dependable resource sharing in HSFs are mechanisms to enforce and monitor critical-section lengths. We followed the choice in [3] to make critical sections non-preemptive for tasks within the same component, because this makes containment of temporal faults within critical sections efficient. Moreover, sufficiently short critical sections can execute with local preemptions disabled and preserve system schedulability. A reservation-based mechanism to monitor and enforce blocking times is unnecessary, see [8], because HSTP complies to existing SRP-based global analysis for HSFs.

We proposed an SRP-compliant budget donation mechanism. Our repetitive self-donation mechanism, complemented with donations from other components, limits preemptions during global resource access as long as possible and preserves the schedulability of independent components. Moreover, it enhances the reliability of resource-sharing components when one of the components misbehaves by accelerating the resource release. HSTP expands across existing protocols in the context of HSFs [3], [4], [5] and integrating its primitives into a reservation-based kernel is straightforward. Our protocol therefore promises a dependable solution to resource sharing in future safety-critical industrial applications for which temporal and functional correctness is essential.

References

[1] I. Shin and I. Lee, “Periodic resource model for compositional real-time guarantees,” in *Real-Time Systems Symp.*, Dec. 2003, pp. 2–13.

[2] T. M. Ghazalie and T. P. Baker, “Aperiodic servers in a deadline scheduling environment,” *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.

[3] R. Davis and A. Burns, “Resource sharing in hierarchical fixed priority pre-emptive systems,” in *Real-Time Systems Symp.*, 2006, pp. 257–267.

[4] M. Behnam, I. Shin, T. Nolte, and M. Nolin, “SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems,” in *Conf. on Embedded Software*, Oct. 2007, pp. 279–288.

[5] M. Bertogna, N. Fisher, and S. Baruah, “Resource-sharing servers for open environments,” *IEEE Trans. on Industrial Informatics*, vol. 5, no. 3, pp. 202–219, Aug. 2009.

[6] T. Baker, “Stack-based scheduling of realtime processes,” *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, March 1991.

[7] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[8] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, “Resource sharing in reservation-based systems,” in *Real-Time Systems Symp.*, Dec. 2001, pp. 171–180.

[9] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronisation,” *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[10] G. Banga, P. Druschel, and J. C. Mogul, “Resource Containers: A New Facility for Resource Management in Server Systems,” in *Symp. on Operating Systems Design and Implementation*, 1999, pp. 45–58.

[11] U. Steinberg, J. Wolter, and H. Härtig, “Fast component interaction for real-time systems,” in *Euromicro Conf. on Real-Time Systems*, July 2005, pp. 89–97.

[12] G. Lipari, G. Lamastra, and L. Abeni, “Task synchronization in reservation-based real-time systems,” *IEEE Trans. on Computers*, vol. 53, no. 12, pp. 1591–1601, Dec. 2004.

[13] R. Santos, G. Lipari, and J. Santos, “Improving the schedulability of soft real-time open dynamic systems: The inheritor is actually a debtor,” *Journal of Systems and Software*, vol. 81, pp. 1093–1104, July 2008.

[14] Z. Deng and J.-S. Liu, “Scheduling real-time applications in open environment,” in *Real-Time Systems Symp.*, Dec. 1997, pp. 308–319.

[15] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, “Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems,” *IEEE Trans. on Industrial Informatics*, vol. 6, no. 1, pp. 93 –104, Feb. 2010.

[16] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, “Protocol-transparent resource sharing in hierarchically scheduled real-time systems,” in *Conf. Emerging Technologies and Factory Automation*, 2010.
[17] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, “Implementation of overrun and skipping in VxWorks,” in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010.
[18] L. Almeida and P. Peidreiras, “Scheduling with temporal partitions: response-time analysis and server design,” in *Conf. on Embedded Software*, Sept. 2004, pp. 95–103.
[19] G. Buttazzo and P. Gai, “Efficient implementation of an EDF scheduler for small embedded systems,” in *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.
[20] M. Bertogna, N. Fisher, and S. Baruah, “Static-priority scheduling and resource hold times,” in *Parallel and Distrib. Processing Symp.*, 2007.
[21] N. Fisher, M. Bertogna, and S. Baruah, “Resource-locking durations in EDF-scheduled systems,” in *Real-Time and Embedded Technology and Applications Symp.*, April 2007, pp. 91–100.
[22] R. Rajkumar, L. Sha, and J. Lehoczky, “Real-time synchronization protocols for multiprocessors,” in *Real-Time Systems Symp.*, Dec. 1988, pp. 259–269.
[23] F. Ridouard, P. Richard, and F. Cottet, “Negative results for scheduling independent hard real-time tasks with self-suspensions,” in *Real-Time Systems Symp.*, Dec. 2004, pp. 47–56.
[24] R. J. Bril, W. F. J. Verhaegh, and C. C. Wüst, “A cognac-glass algorithm for conditionally guaranteed budgets,” in *Real-Time Systems Symp.*, Dec. 2006, pp. 388–397.
[25] S. K. Baruah, “Resource sharing in EDF-scheduled systems: A closer look,” in *Real-Time Systems Symp.*, 2006, pp. 379–387.
[26] M. Behnam, T. Nolte, M. Åsberg, and R. J. Bril, “Overrun and skipping in hierarchically scheduled real-time systems,” in *Conf. on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 519–526.
[27] M. Behnam, T. Nolte, and N. Fisher, “On optimal real-time subsystem-interface generation in the presence of shared resources,” in *Conf. on Emerging Technologies and Factory Automation*, Sept. 2010.
[28] Micrium, “RTOS and tools,” March 2010. [Online]. Available: http://micrium.com/

PAPER F:

Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources

M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien

In proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT),
pp. 39–48, July 2011

Abstract

Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating temporal isolation between components that need to be integrated on a single shared processor. To support resource sharing within two-level, fixed priority scheduled HSFs, two synchronization protocols based on the stack resource policy (SRP) have recently been presented, i.e. HSRP [1] and SIRAP [2]. In the presence of shared resources, however, temporal isolation may break when one of the accessing components executes longer than specified during global resource access. As a solution we propose a SRP-based synchronization protocol for HSFs, named Basic Hierarchical Synchronization protocol with Temporal Protection (B-HSTP). The schedulability of those components that are independent of the unavailable resource is unaffected.

This paper describes an implementation to provide HSFs, accompanied by SRP-based synchronization protocols, with means for temporal isolation. We base our implementations on the commercially available real-time operating system μC/OS-II, extended with proprietary support for two-level fixed priority preemptive scheduling. We specifically show the implementation of B-HSTP and we investigate the system overhead induced by its synchronization primitives in combination with HSRP and SIRAP. By supporting both protocols in our HSF, their primitives can be selected based on the protocol's relative strengths.

1 Introduction

The increasing complexity of real-time systems demands a decoupling of (*i*) development and analysis of individual components and (*ii*) integration of components on a shared platform, including analysis at the system level. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm for facilitating this decoupling [3]. A component that is validated to meet its timing constraints when executing in isolation will continue meeting its timing constraints after integration (or admission) on a shared platform. The HSF therefore provides a promising solution for current industrial standards, e.g. the AUtomotive Open System ARchitecture (AUTOSAR) [4] which specifies that an underlying OSEK-based operating system should prevent timing faults in any component to propagate to different components on the same processor. The HSF provides temporal isolation between components by allocating a *budget* to each component, which gets mediated access to the processor by means of a *server*.

An HSF without further resource sharing is unrealistic, however, since components may for example use operating system services, memory mapped devices and shared communication devices which require mutually exclusive access. Extending an HSF with such support makes it possible to share logical resources between arbitrary tasks, which are located in arbitrary components, in a mutually exclusive manner. A resource that is used in more than one component is denoted as a *global shared resource*. A resource that is only shared by tasks within a single component is a *local shared resource*. If a task that accesses a global shared resource is suspended during its execution due to the exhaustion of its budget, excessive blocking periods can occur which may hamper the correct timeliness of other components [5].

Looking at existing industrial real-time systems, fixed-priority preemptive scheduling (FPPS) is the de-facto standard of task scheduling, hence we focus on an HSF with support for FPPS within a component. Having such support will simplify migration to and integration of existing legacy applications into the HSF. Our current research efforts are directed towards the conception and realization of a two-level HSF that is based on (*i*) FPPS for both *global scheduling* of servers allocated to components and *local scheduling* of tasks within a component and (*ii*) the Stack Resource Policy (SRP) [6] for both local and global resource sharing.

To accommodate resource sharing between components, two synchronization protocols [1], [2] have been proposed based on SRP for two-level FPPS-based HSFs. Each of these protocols describes a run-time mechanism to handle the depletion of a component's budget during global resource access. In short, two general approaches are proposed: (*i*) *self-blocking* when the remaining budget is insufficient to complete a critical section [2] or (*ii*) *overrun* the budget until the critical section ends [1]. However, when a task exceeds its specified worst-case critical-section length, i.e. it *misbehaves* during global resource access, temporal isolation between components is no longer guaranteed. The protocols in [1], [2] therefore break the temporal encapsulation and fault-containment properties of an HSF without the presence of complementary protection.

1.1 Problem description

Most off-the-shelf real-time operating systems, including μC/OS-II [7], do not provide an implementation for SRP nor hierarchical scheduling. We have extended μC/OS-II with support for idling periodic servers (IPS) [8] and two-level FPPS. However, existing implementations of the synchronization protocols in our framework [9], [10], as well as in the framework presented in [11], do not provide any temporal isolation during global resource access.

A solution to limit the propagation of temporal faults to those components that share global resources is considered in [12]. Each task is assigned a dedicated budget per global resource access and this budget is synchronous with the period of that task. However, in [12] they allow only a single task per component.

We consider the problem to limit the propagation of temporal faults in HSFs, where multiple concurrent tasks are allocated a shared budget, to those components that share a global resource. Moreover, we present an efficient implementation and evaluation of our protocol in μC/OS-II. The choice of operating system is driven by its former OSEK compatibility[1].

1.2 Contributions

The contributions of this paper are fourfold.

- To achieve temporal isolation between components, even when resource-sharing components misbehave, we propose a modified SRP-based synchronization protocol, named *Basic Hierarchical Synchronization protocol with Temporal Protection* (B-HSTP).
- We show its implementation in a real-time operating system, extended with support for two-level fixed-priority scheduling, and we efficiently achieve fault-containment by disabling preemptions of other tasks within the same component during global resource access.
- We show that B-HSTP complements existing synchronization protocols [1], [2] for HSFs.
- We evaluate the run-time overhead of our B-HSTP implementation in μC/OS-II on the OpenRISC platform [13]. These overheads become relevant during deployment of a resource-sharing HSF.

1. Unfortunately, the supplier of μC/OS-II, Micrium, has discontinued the support for the OSEK-compatibility layer.

1.3 Organization

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 presents our system model. Section 4 presents our resource-sharing protocol, B-HSTP, which guarantees temporal isolation to independent components. Section 5 presents our existing extensions for μC/OS-II comprising two-level FPPS-based scheduling and SRP-based resource arbitration. Section 6 presents B-HSTP's implementation using our existing framework. Section 7 investigates the system overhead corresponding to our implementation. Section 8 discusses practical extensions to B-HSTP. Finally, Section 9 concludes this paper.

2 Related work

Our basic idea is to use two-level SRP to arbitrate access to global resources, similar as [1], [2]. In literature several alternatives are presented to accommodate task communication in reservation-based systems. De Niz et al. [12] support resource sharing between reservations based on the immediate priority ceiling protocol (IPCP) [14] in their FPPS-based Linux/RK resource kernel and use a run-time mechanism based on resource containers [15] for temporal protection against misbehaving tasks. Steinberg et al. [16] showed that these resource containers are expensive and efficiently implemented a capacity-reserve donation protocol to solve the problem of priority inversion for tasks scheduled in a fixed-priority reservation-based system. A similar approach is described in [17] for EDF-based systems and termed bandwidth-inheritance (BWI). BWI regulates resource access between tasks that each have their dedicated budget. It works similar to the priority-inheritance protocol [14], i.e. when a task blocks on a resource it donates its remaining budget to the task that causes the blocking. However, all these approaches assume a one-to-one mapping from tasks to budgets, and inherently only have a single scheduling level.

In HSFs a group of concurrent tasks, forming a component, are allocated a budget [18]. A prerequisite to enable independent analysis of interacting components and their integration is the knowledge of which resources a task will access [2], [19]. When a task accesses a global shared resource, one needs to consider the priority inversion between components as well as local priority inversion between tasks within the component. To *prevent budget depletion* during global resource access in FPPS-based HSFs, two synchronization protocols have been proposed based on SRP [6]: HSRP [1] and SIRAP [2]. Although HSRP [1] originally does not integrate into HSFs due to the lacking support for independent analysis of components, Behnam et al. [19] lifted this limitation. However, these two protocols, including their implementations in [9], [10], [11], assume that components respect their timing contract with respect to global resource sharing. In this paper we present an implementation of HSRP and SIRAP protocols that limits the unpredictable interferences caused by contract violations to the components that share the global resource.

3 REAL-TIME SCHEDULING MODEL

We consider a two-level FPPS-scheduled HSF, following the periodic resource model [3], to guarantee processor allocations to components. We use SRP-based synchronization to arbitrate mutually exclusive access to global shared resources.

3.1 Component model

A system contains a set $\mathcal{R}$ of M global logical resources R_1, R_2, ..., R_M, a set $\mathcal{C}$ of N components C_1, C_2, ..., C_N, a set $\mathcal{B}$ of N budgets for which we assume a periodic resource model [3], and a single shared processor. Each component C_s has a dedicated budget which specifies its periodically guaranteed fraction of the processor. The remainder of this paper leaves budgets implicit, i.e. the timing characteristics of budgets are taken care of in the description of components. A server implements a policy to distribute the available budget to the component's workload.

The timing characteristics of a component C_s are specified by means of a triple $< P_s, Q_s, \mathcal{X}_s >$, where $P_s \in \mathbb{R}^+$ denotes its period, $Q_s \in \mathbb{R}^+$ its budget, and $\mathcal{X}_s$ the set of maximum access times to global resources. The maximum value in $\mathcal{X}_s$ is denoted by X_s, where $0 < Q_s + X_s \leq P_s$. The set $\mathcal{R}_s$ denotes the subset of M_s global resources accessed by component C_s. The maximum time that a component C_s executes while accessing resource $R_l \in \mathcal{R}_s$ is denoted by X_{sl}, where $X_{sl} \in \mathbb{R}^+ \cup \{0\}$ and $X_{sl} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$.

3.2 Task model

Each component C_s contains a set $\mathcal{T}_s$ of n_s sporadic tasks τ_{s1}, τ_{s2}, ..., τ_{sn_s}. Timing characteristics of a task $\tau_{si} \in \mathcal{T}_s$ are specified by means of a triple $< T_{si}, E_{si}, D_{si} >$, where $T_{si} \in \mathbb{R}^+$ denotes its minimum inter-arrival time, $E_{si} \in \mathbb{R}^+$ its worst-case computation time, $D_{si} \in \mathbb{R}^+$ its (relative) deadline, where $0 < E_{si} \leq D_{si} \leq T_{si}$. The worst-case execution time of task τ_{si} within a critical section accessing R_l is denoted c_{sil}, where $c_{sil} \in \mathbb{R}^+ \cup \{0\}$, $E_{si} \geq c_{sil}$ and $c_{sil} > 0 \Leftrightarrow R_l \in \mathcal{R}_s$. All (critical-section) execution times are accounted in terms of processor cycles and allocated to the calling task's budget. For notational convenience we assume that tasks (and components) are given in priority order, i.e. τ_{s1} has the highest priority and τ_{sn_s} has the lowest priority.

3.3 Synchronization protocol

Traditional synchronization protocols such as PCP [14] and SRP [6] can be used for *local* resource sharing in HSFs [20]. This paper focuses on arbitrating *global* shared resources using SRP. To be able to use SRP in an HSF for synchronizing global resources, its associated ceiling terms need to be extended and excessive blocking must be prevented.

3.3.1 Resource ceilings

With every global resource R_l two types of resource ceilings are associated; a *global* resource ceiling RC_l for global scheduling and a *local* resource ceiling rc_{sl} for local scheduling. These ceilings are statically calculated values, which are defined as the highest priority of any component or task that shares the resource. According to SRP, these ceilings are defined as:

$$\begin{aligned} RC_l &= \min(N, \min\{s \mid R_l \in \mathcal{R}_s\}), && (1)\\ rc_{sl} &= \min(n_s, \min\{i \mid c_{sil} > 0\}). && (2)\end{aligned}$$

We use the outermost $\min$ in (1) and (2) to define RC_l and rc_{sl} in those situations where no component or task uses R_l.

3.3.2 System and component ceilings

The system and component ceilings are dynamic parameters that change during execution. The system ceiling is equal to the highest global resource ceiling of a currently locked resource in the system. Similarly, the component ceiling is equal to the highest local resource ceiling of a currently locked resource within a component. Under SRP a task can only preempt the currently executing task if its priority is higher than its component ceiling. A similar condition for preemption holds for components.

3.3.3 Prevent excessive blocking

HSRP [1] uses an overrun mechanism [19] when a budget depletes during a critical section. If a task $\tau_{si} \in \mathcal{T}_s$ has locked a global resource when its component's budget Q_s depletes, then component C_s can continue its execution until task τ_{si} releases the resource. These budget overruns cannot take place across replenishment boundaries, i.e. the analysis guarantees $Q_s + X_s$ processor time before the relative deadline P_s [1], [19].

SIRAP [2] uses a self-blocking approach to prevent budget depletion inside a critical section. If a task τ_{si} wants to enter a critical section, it enters the critical section at the earliest time instant so that it can complete the critical section before the component's budget depletes. If the remaining budget is insufficient to lock and release a resource R_l before depletion, then (*i*) the task blocks itself until budget replenishment and (*ii*) the component ceiling is raised to prevent tasks $\tau_{sj} \in \mathcal{T}_s$ with a priority lower than the local ceiling rc_{sl} to execute until the requested critical section has been finished.

The relative strengths of HSRP and SIRAP have been analytically investigated in [21] and heavily depend on the chosen system parameters. To enable the selection of a particular protocol based on its strengths, we presented an implementation supporting both protocols with transparent interfaces for the programmer [9]. In this paper we focus on mechanisms to extend these protocols with temporal protection and merely investigate their relative complexity with respect to our temporal-protection mechanisms.

4 SRP WITH TEMPORAL PROTECTION

Temporal faults may cause improper system alterations, e.g. due to unexpectedly long blocking or an inconsistent state of a resource. Without any protection a self-blocking approach [2] may miss its purpose under erroneous circumstances, i.e. when a task overruns its budget to complete its critical section. Even an overrun approach [1], [19] needs to guarantee a maximum duration of the overrun situation. Without such a guarantee, these situations can hamper temporal isolation and resource *availability* to other components due to unpredictable blocking effects. A straightforward implementation of the overrun mechanism, e.g. as implemented in the ERIKA kernel [22], where a task is allowed to indefinitely overrun its budget as long as it locks a resource, is therefore not *reliable*.

4.1 Resource monitoring and enforcement

A common approach to ensure temporal isolation and prevent propagation of temporal faults within the system is to group tasks that share resources into a single component [20]. However, this might be too restrictive and lead to large, incoherent component designs, which violates the principle of HSFs to independently develop components. Since a component defines a coherent piece of functionality, a task that accesses a global shared resource is critical with respect to all other tasks in the same component.

To guarantee temporal isolation between components, the system must *monitor* and *enforce* the length of a global critical section to prevent a malicious task to execute longer in a critical section than assumed during system analysis [12]. Otherwise such a misbehaving task may increase blocking to components with a higher priority, so that even independent components may suffer, as shown in Figure 1.

To prevent this effect we introduce a *resource-access budget* q_s in addition to a component's budget Q_s, where budget q_s is used to enforce critical-section

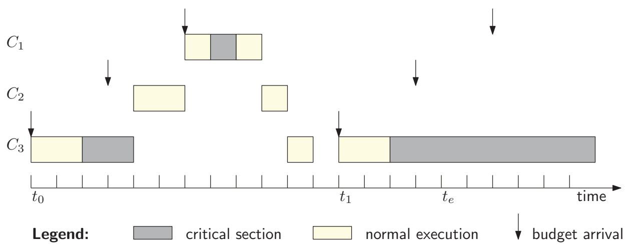

Fig. 1. Temporal isolation is unassured when a component, C_3, exceeds its specified critical-section length, i.e. at time instant t_e. The system ceiling blocks all other components.

lengths. When a resource R_l gets locked, q_s replenishes to its full capacity, i.e. $q_s \leftarrow X_{sl}$. To monitor the available budget at any moment in time, we assume the availability of a function $Q_s^{\texttt{rem}}(t)$ that returns the remaining budget of Q_s. Similarly, $q_s^{\texttt{rem}}(t)$ returns the remainder of q_s at time t. If a component C_s executes in a critical section, then it consumes budget from Q_s and q_s in parallel, i.e. depletion of either Q_s or q_s forbids component C_s to continue its execution. We maintain the following invariant to prevent budget depletion during resource access:

Invariant 1: $Q_s^{\texttt{rem}}(t) \geq q_s^{\texttt{rem}}(t)$.

The way of maintaining this invariant depends on the chosen policy to prevent budget depletion during global resource access, e.g. by means of SIRAP [2] or HSRP [1].

4.1.1 Fault containment of critical sections

Existing SRP-based synchronization protocols in [2], [19] make it possible to choose the local resource ceilings, rc_{sl}, according to SRP [6]. In [23] techniques are presented to trade-off preemptiveness against resource holding times. Given their common definition for local resource ceilings, a resource holding time, X_{sl}, may also include the interference of tasks with a priority higher than the resource ceiling. Task τ_{si} can therefore lock resource R_l longer than specified, because an interfering task τ_{sj} (where $\pi_{sj} > rc_{sl}$) exceeds its computation time, E_{sj}.

To prevent this effect we choose to disable preemptions for other tasks within the same component during critical sections, i.e. similar as HSRP [1]. As a result X_{sl} only comprises task execution times within a critical section, i.e.

$$X_{sl} = \max_{1 \leq i \leq n_s} c_{sil}. \tag{3}$$

Since X_{sl} is enforced by budget q_s, temporal faults are contained within a subset of resource-sharing components.

4.1.2 Maintaining SRP ceilings

To enforce that a task τ_{si} resides no longer in a critical section than specified by X_{sl}, a resource $R_l \in \mathcal{R}$ maintains a state *locked* or *free*. We introduce an extra state *busy* to signify that R_l is locked by a misbehaving task. When a task τ_{si} tries to exceed its maximum critical-section length X_{sl}, we update SRP's *system ceiling* by mimicking a resource unlock and mark the resource *busy* until it is released. Since the system ceiling decreases after τ_{si} has executed for a duration of X_{sl} in a critical section to resource R_l, we can no longer guarantee absence of deadlocks. Nested critical sections to global resources are therefore unsupported. One may alternatively aggregate global resource accesses into a simultaneous lock and unlock of a single artificial resource [24]. Many protocols, or their implementations, lack deadlock avoidance [11], [12], [16], [17].

Although it seems attractive from a schedulability point of view to release the *component ceiling* when the critical-section length is exceeded, i.e. similar to the system ceiling, this would break SRP compliance, because a task may block on a busy resource instead of being prevented from starting its execution. Our approach therefore preserves the SRP property to share a single, consistent execution stack per component [6]. At the global level tasks can be blocked by a depleted budget, so that components cannot share an execution stack anyway.

4.2 An overview of B-HSTP properties

This section presented a basic protocol to establish hierarchical synchronization with temporal protection (B-HSTP). Every lock operation to resource R_l replenishes a corresponding resource-access budget q_s with an amount X_{sl}. After this resource-access budget has been depleted, the component blocks until its normal budget Q_s replenishes. We can derive the following convenient properties from our protocol:

1) as long as a component behaves according to its timing contract, we strictly follow SRP;
2) because local preemptions are disabled during global resource access and nested critical sections are prohibited, each component can only access a single global resource at a time;
3) similarly, each component can at most keep a single resource in the busy state at a time;
4) each access to resource R_l by task τ_{si} may take at most X_{sl} budget from budget Q_s, where $c_{sil} \leq X_{sl}$.
5) after depleting resource-access budget q_s, a task may continue in its component normal budget Q_s with a decreased system ceiling. This guarantees that independent components are no longer blocked by the system ceiling;

6) when a component blocks on a busy resource, it discards all remaining budget until its next replenishment of Q_s. This avoids budget suspension, which can lead to scheduling anomalies [25].

As a consequence of property 2, we can use a simple non-preemptive locking mechanism at the local level rather than using SRP. We therefore only need to implement SRP at the global level and we can use a simplified infrastructure at the local level compared to the implementations in [9], [10], [11].

5 μC/OS-II AND ITS EXTENSIONS RECAPITULATED

The μC/OS-II operating system is maintained and supported by Micrium [7], and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Micrium provides the full μC/OS-II source code with accompanying documentation [26]. The μC/OS-II kernel provides preemptive multitasking for up to 256 tasks, and the kernel size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

Most real-time operating systems, including μC/OS-II, do not include a reservation-based scheduler, nor provide means for hierarchical scheduling. In the remainder of this section we outline our realization of such extensions for μC/OS-II, which are required basic blocks to enable the integration of global synchronization with temporal protection.

5.1 Timed Event Management

Intrinsic to our reservation-based component scheduler is timed-event management. This comprises timers to accommodate (*i*) periodic timers at the global level for budget replenishment of periodic servers and at the component level to enforce minimal inter-arrivals of sporadic task activations and (*ii*) virtual timers to track a component's budget. The corresponding timer handlers are executed in the context of the timer interrupt service routine (ISR).

We have implemented a dedicated module to manage *relative timed event queues* (RELTEQs) [27]. The basic idea is to store events relative to each other, by expressing the expiration time of an event relative to the arrival time of the previous event. The arrival time of the head event is relative to the current time, see Figure 2

A *system queue* tracks all server events. Each server has its own *local queue* to track its tasks' events, e.g. task arrivals. When a server is suspended its local queues are deactivated to prevent that expiring events interfere with other servers. When a server resumes, its local queues are synchronized with global time. A mechanism to synchronize server queues with global time is implemented by means of a *stopwatch queue*, which keeps track of the time passed since the last server switch.

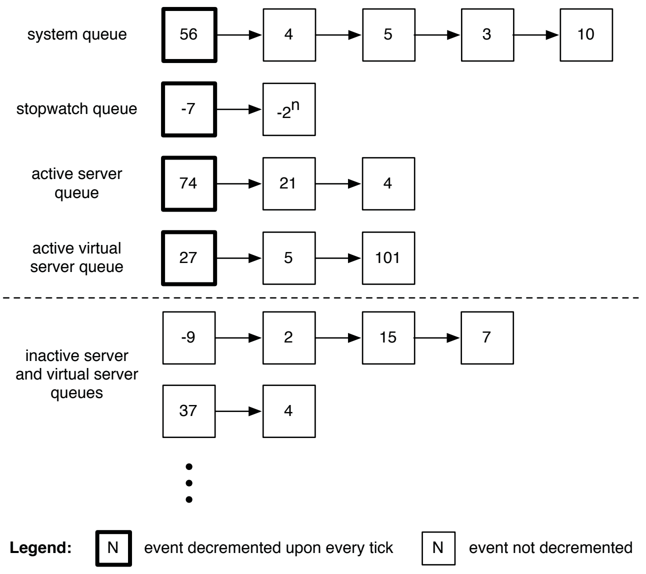

Fig. 2. RELTEQ-based timer management for two-level HSFs.

A dedicated server queue provides support for *virtual timers* to trigger timed events *relative to the consumed budget*. Since an inactive server does not consume any of its budget, a virtual timer queue is not synchronized when a server is resumed. We consider only budget depletion as a virtual event, so that a component can inspect its virtual-timer in constant time.

5.2 Server Scheduling

A *server* is assigned to each component to distribute its allocated budget to the component's tasks. A global scheduler is used to determine which server should be allocated the processor at any given time. A local scheduler determines which of the chosen server's tasks should actually execute. Although B-HSTP is also applicable to other server models, we assume that a component is implemented by means of an *idling periodic server* (IPS) [8]. Extending μC/OS-II with basic HSF support requires a realization of the following concepts:

5.2.1 Global Scheduling

At the system level a RELTEQ queue is introduced to keep track of server periods. We use a bit-mask to represent whether a server has capacity left. When the scheduler is called, it traverses the RELTEQ and activates the *ready* server with the earliest deadline in the queue. Subsequently, the μC/OS-II fixed-priority scheduler determines the highest priority ready task within the server.

5.2.2 Periodic Servers

Since μC/OS-II tasks are bundled in groups of sixteen to accommodate efficient fixed-priority scheduling, a server can naturally be represented by such a group. The implementation of periodic servers is very similar to implementing periodic tasks using our RELTEQ extensions [27]. An idling server contains an idling task at the lowest, local priority, which is always ready to execute.

5.2.3 Greedy Idle Server

In our HSF, we reserve the lowest priority level for a idle server, which contains μC/OS-II's idle task at the lowest local priority. Only if no other server is eligible to execute, then the idle server is switched in.

5.3 Global SRP implementation

The key idea of SRP is that when a component needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Nice properties of SRP are its simple locking and unlocking operations. In turn, during run-time we need to keep track of the system ceiling and the scheduler needs to compare the highest ready component priority with the system ceiling. Hence, a preemption test is performed during run time by the scheduler: A component cannot preempt until its priority is the highest among those of all ready components *and* its priority is higher than the *system ceiling*. In the original formulation of SRP [6], it introduces the notion of preemption-levels. This paper considers FPPS, which makes it possible to unify preemption-levels with priorities.

The system ceiling is a dynamic parameter that changes during execution. Under SRP, a component can only preempt the currently executing component if its priority is higher than the system ceiling. When no resources are locked the system ceiling is zero, meaning that it does not block any tasks from preempting. When a resource is locked, the system ceiling is adjusted dynamically using the resource ceiling, so that the system ceiling represents the highest resource ceiling of a currently locked resource in the system. A run-time mechanism for tracking the system ceiling can be implemented by means of a stack data structure.

5.3.1 SRP data and interface description

Each resource accessed using an SRP-based mutex is represented by a `Resource` structure. This structure is defined as follows:

```
typedef struct resource{
    INT8U  ceiling;
    INT8U  lockingTask;
    void*  previous;
} Resource;
```

The `Resource` structure stores properties which are used to track the system ceiling, as explained in below. The corresponding mutex interfaces are defined as follows:

- Create a SRP mutex:
  ```
  Resource* SRPMutexCreate(INT8U ceiling,
                           INT8U *err);
  ```
- Lock a SRP mutex:
  ```
  void SRPMutexLock(Resource* r, INT8U *err);
  ```
- Unlock a SRP mutex:
  ```
  void SRPMutexUnlock(Resource* r);
  ```

The lock and unlock operations only perform bookkeeping actions by increasing and decreasing the system ceiling.

5.3.2 SRP operations and scheduling

We extended μC/OS-II with the following SRP rules at the server level:

Tracking the system ceiling: We use the `Resource` data-structure to implement a *system ceiling stack*. `ceiling` stores the resource ceiling and `lockingTask` stores the identifier of the task currently holding the resource. From the task identifier we can deduct to which server it is attached. The `previous` pointer is used to maintain the stack structure, i.e. it points to the previous `Resource` structure on the stack. The `ceiling` field of the `Resource` on top of the stack represents the current system ceiling.

Resource locking: When a component tries to lock a resource with a resource ceiling higher than the current system ceiling, the corresponding resource ceiling is pushed on top of the system ceiling stack.

Resource unlocking: When unlocking a resource, the value on top of the system ceiling stack is popped. The absence of nested critical sections guarantees that the system ceiling represents the resource to be unlocked. The scheduler is called to allow for scheduling ready components that might have arrived during the execution of the critical section.

Global scheduling: When the μC/OS-II scheduler is called it calls a function which returns the highest priority ready component. Accordingly to SRP we extend this function with the following rule: when the highest ready component has a priority lower than or equal to the current system ceiling, the priority of the *task* on top of the resource stack is returned. The returned priority serves as a task identifier, which makes easily allows to deduct the corresponding component.

Local scheduling: The implementations of two-level SRP protocols in [9], [10], [11] also keep track of component ceilings. We only have a binary local ceiling to indicate whether preemptions are enabled or disabled, because we explicitly chose local resource ceilings equal to the highest local priority. During global resource access, the local scheduler can only select the resource-accessing task for execution.

6 B-HSTP IMPLEMENTATION

In this section we extend the framework presented in Section 5 with our proposed protocol, B-HSTP. In many microkernels, including μC/OS-II, the only way for tasks to share data structures with ISRs is by means of disabling interrupts. We therefore assume that our primitives execute non-preemptively with interrupts disabled.

Because critical sections are non-nested and local preemptions are disabled, at most one task τ_{si} at a time in each component may use a global resource. This convenient system property makes it possible to multiplex both resource-access budget q_s and budget Q_s on a single budget timer by using our virtual timer mechanism. The remaining budget $Q_s^{\texttt{rem}}(t)$ is returned by a function that depends on the virtual timers mechanism, see Section 5.1. A task therefore merely blocks on its component's budget, which we implement by adjusting the single available budget timer $Q_s^{\texttt{rem}}(t)$.

6.0.1 Resource locking

The lock operation updates the local ceiling to prevent other tasks within the component from interfering during the execution of the critical section. Its pseudo-code is presented in Algorithm 1.

In case we have enabled SIRAP, rather than HSRP's overrun, there must be sufficient remaining budget within the server's current period in order to successfully lock a resource. If the currently available budget $Q_s^{\texttt{rem}}(t)$ is insufficient, the task will spinlock until the next replenishment event expires. To avoid a race-condition between a resource unlock and budget depletion, we require that $Q_s^{\texttt{rem}}(t)$ is strictly larger than X_{sr} before granting access to a resource R_r.

A task may subsequently block on a busy resource, until it becomes free. When it encounters a busy resource, it suspends the component and discards all remaining budget. When the resource becomes free and the task which attempted to lock the resource continues its execution, it is guaranteed that there is sufficient budget to complete the critical section (assuming that it does not exceed its specified length, X_{sr}). The reason for this is that a component discards its budget when it blocks on a busy resource and can only continue with a fully replenished budget. This resource holding time X_{sr} defines the

Algorithm 1 void HSF_lock(`Resource* r`);

```
1: $updateComponentCeiling(r)$;
2: if HSF_MUTEX_PROTOCOL == SIRAP then
3:     while $X_{sr} >= Q_s^{\mathrm{rem}}(t)$ do {apply SIRAP's self-blocking}
4:         $enableInterrups$;
5:         $disableInterrups$;
6:     end while
7: end if
8: while $r.status = busy$ do {self-suspend on a busy resource}
9:     $setComponentBudget(0)$;
10:    $enableInterrups$;
11:    $Schedule()$;
12:    $disableInterrups$;
13: end while
14: $Q_s^{\triangledown} \leftarrow Q_s^{\mathrm{rem}}(t)$;
15: $setComponentBudget(X_{sr})$;
16: $C_s.lockedResource \leftarrow r$;
17: $r.status \leftarrow locked$
18: $SRPMutexLock(r)$;
```

resource-access budget of the locking task and component. The component's remaining budget is saved as $Q_s^{\triangledown}$ and reset to X_{sl} before the lock is established.

$setComponentBudget(0)$, see line 9, performs two actions: (*i*) the server is blocked to prevent the scheduler from rescheduling the server before the next replenishment, and (*ii*) the budget-depletion timer is canceled.

6.0.2 Resource unlocking

Unlocking a resource means that the system and component ceilings must be decreased. Moreover, the amount of consumed budget is deducted from the components stored budget, $Q_s^{\triangledown}$. We do not need to restore the component's budget, if the system ceiling is already decreased at the depletion of its resource-access budget, i.e. when a component has exceeded its specified critical-section length. The unlock operation in pseudo-code is as follows:

Algorithm 2 void HSF_unlock(`Resource* r`);

```
1: $updateComponentCeiling(r)$;
2: $r.status \leftarrow free$;
3: $C_s.lockedResource \leftarrow 0$;
4: if $System\_ceiling == RC_r$ then
5:     $setComponentBudget(\max(0, Q_s^{\triangledown} - (X_{sr} - Q_s^{\mathrm{rem}}(t))))$;
6: else
7:     ; {we already accounted the critical section upon depletion of $X_{sr}$}
8: end if
9: $SRPMutexUnlock(r)$;
```

6.0.3 Budget depletion

We extend the budget-depletion event handler with the following rule: if any task within the component holds a resource, then the global resource ceiling is decreased according to SRP and the resource is marked busy. A component C_s may continue in its restored budget with a decreased system ceiling. The pseudo-code of the budget-depletion event handler is as follows:

Algorithm 3 on budget depletion:

```
1: if $C_s.lockedResource \neq 0$ then
2:     $r \leftarrow C_s.lockedResource$;
3:     $r.status \leftarrow busy$;
4:     $SRPMutexUnlock(r)$;
5:     $setComponentBudget(\max(0, Q_s^{\triangledown} - X_{sr}))$;
6: else
7:     ; {apply default budget-depletion strategy}
8: end if
```

6.0.4 Budget replenishment

For each periodic server an event handler is periodically executed to recharge its budget. We extend the budget-replenishment handler with the following rule: if any task within the component holds a resource busy, then the global resource ceiling is increased according to SRP and the resource-access budget is replenished with X_{sr} of resource R_r. A component C_s may continue in its restored budget with an increased system ceiling for that duration, before the remainder of its normal budget Q_s becomes available. The pseudo-code of this event handler is shown in Algorithm 4.

Algorithm 4 on budget replenishment:

```
1: $Q_s^{\triangledown} \leftarrow Q_s$;
2: if $C_s.lockedResource \neq 0$ then {$C_s$ keeps a resource busy}
3:     $r \leftarrow C_s.lockedResource$;
4:     $setComponentBudget(X_{sr})$;
5:     $SRPMutexLock(r)$;
6: else
7:     ; {Apply default replenishment strategy}
8: end if
```

7 Experiments and results

This section evaluates the implementation costs of B-HSTP. First, we present a brief overview of our test platform. Next, we experimentally investigate the system overhead of the synchronization primitives and compare these to our earlier protocol implementations. Finally, we illustrate B-HSTP by means of an example system.

7.1 Experimental setup

We recently created a port for μC/OS-II to the OpenRISC platform [13] to experiment with the accompanying cycle-accurate simulator. The OpenRISC simulator allows software-performance evaluation via a cycle-count register. This profiling method may result in either longer or shorter measurements between two matching calls due to the pipelined OpenRISC architecture. Some instructions in the profiling method interleave better with the profiled code than others. The measurement accuracy is approximately 5 instructions.

7.2 Synchronization overheads

In this section we investigate the overhead of the synchronization primitives of B-HSTP. By default, current analysis techniques do not account for overheads of the corresponding synchronization primitives, although these overheads become of relevance upon deployment of such a system. Using more advanced analysis methods, for example as proposed in [28], these measures can be included in the existing system analysis. The overheads introduced by the implementation of our protocol are summarized in Table 1 and compared to our earlier implementation of HSRP and SIRAP in [9], [10].

7.2.1 Time complexity

Since it is important to know whether a real-time operating system behaves in a time-wise predictable manner, we investigate the disabled interrupt regions caused by the execution of B-HSTP primitives. Our synchronization primitives are independent of the number of servers and tasks in a system, but introduce overheads that interfere at the system level due to their required timer manipulations.

Manipulation of timers makes our primitives more expensive than a straightforward two-level SRP implementation. However, this is the price for obtaining temporal isolation. The worst-case execution time of the lock operation increases with 380 instructions in every server period in which a component blocks, so that the total cost depends on the misbehaving critical-section length which causes the blocking. The budget replenishment handler only needs to change the amount to be replenished, so that B-HSTP itself does not contribute much to the total execution time of the handler. These execution times of the primitives must be included in the system analysis by adding these to the critical section execution times, X_{sl}. At the local scheduling level B-HSTP is more efficient, however, because we use a simple non-preemptive locking mechanism.

7.2.2 Memory complexity

The code sizes in bytes of B-HSTP's lock and unlock operations, i.e. 1228 and 612 bytes, is higher than the size of plain SRP, i.e. 196 and 192 bytes. This

TABLE 1
Best-case (BC) and worst-case (WC) execution times for SRP-based protocols, including B-HSTP, measured on the OpenRISC platform in number of processor instructions.

Event	**single-level SRP** [10]		**HSRP** (see [9], [10])		**SIRAP** (see [9], [10])		**B-HSTP**	
	BC	WC	BC	WC	BC	WC	BC	WC
Resource lock	124	124	196	196	214	224	763	-
Resource unlock	106	106	196	725	192	192	688	697
Budget depletion	-	-	0	383	-	-	59	382
Budget replenishment	-	-	0	15	-	-	65	76

includes the transparently implemented self-blocking and overrun mechanisms and timer management. μC/OS-II's priority-inheritance protocol has similar sized lock and unlock primitives, i.e. 924 and 400 bytes.

Each SRP resource has a data structure at the global level, i.e. we have M shared resources. Each component only needs to keep track of a single globally shared resource, because local preemptions are disabled during global resource access. However, each component C_s needs to store its resource-access durations for all its resources $R_l \in \mathcal{R}_s$.

7.3 SIRAP and HSRP re-evaluated

From our first implementation of SIRAP and HSRP, we observed that SIRAP induces overhead locally within a component, i.e. the spin-lock, which checks for sufficient budget to complete the critical section, adds to the budget consumption of the particular task that locks the resource. SIRAP's overhead consists at least of a single test for sufficient budget in case the test is passed. The overhead is at most two of such tests in case the initial test fails, i.e. one additional test is done after budget replenishment before resource access is granted. All remaining tests during spinlocking are already included as self-blocking terms in the local analysis [2]. The number of processor instructions executed for a single test is approximately 10 instructions on our test platform.

HSRP introduces overhead that interferes at the global system level, i.e. the overrun mechanism requires to manipulate event timers to replenish an overrun budget when the normal budget of a component depletes. This resulted in a relatively large overhead for HSRP's unlock operation compared to SIRAP, see Table 1. Since similar timer manipulations are required for B-HSTP, the difference in overhead for HSRP and SIRAP becomes negligible when these protocols are complemented with means for temporal isolation. Furthermore, the absolute overheads are in the same order of magnitude.

7.4 B-HSTP: an example

We have recently extended our development environment with a visualization tool, which makes it possible to plot a HSF's behaviour [29] by instrumenting the code, executed on the OpenRISC platform, of our μC/OS-II extensions. To demonstrate the behavior of B-HSTP, consider an example system comprised of three components (see Table 2) each with two tasks (see Table 3) sharing a single global resource R_1. We use the following conventions:

1) the component or task with the lowest number has the highest priority;
2) the computation time of a task is denoted by the consumed time units after locking and before unlocking a resource. For example, the scenario

– $E_{s1,1}$; $Lock(R_1)$; $E_{s1,2}$; $Unlock(R_1)$; $E_{s1,3}$ – is denoted as $E_{s1,1}+E_{s1,2}+E_{s1,3}$ and the term $E_{s1,2}$ represents the critical-section length, c_{s1l};

3) the resource holding time is longest critical-section length within a component, see Equation 3.
4) the component ceilings of the shared resource, R_1, are equal to the highest local priority, as dictated by B-HSTP.

The example presented in Figure 3 complements HSRP's overrun mechanism with B-HSTP.

TABLE 2
Example System: component parameters

Server	Period (P_s)	Budget (Q_s)	Res. holding time (X_s)
IPS 1	110	12	4.0
IPS 2	55	8	0.0
IPS 3	50	23	7.4

TABLE 3
Example System: task parameters

Server	Task	Period	Computation time
IPS 1	Task 11	220	6.5 +4.0+6.5
IPS 1	Task 12	610	0.0+0.17+0.0
IPS 2	Task 21	110	5.0
IPS 2	Task 22	300	7.0
IPS 3	Task 31	100	12+7.4+12
IPS 3	Task 32	260	0.0+0.095+0.0

At every time instant that a task locks a resource, the budget of the attached server is manipulated according to the rules of our B-HSTP protocol, e.g. see time 11 where task 31 locks the global resource and the budget of IPS 3 is changed. After task 31 has executed two resource accesses within its specified length, in the third access it gets stuck in an infinite loop, see time instant 211. Within IPS 3, the lower priority task is indefinitely blocked, since B-HSTP does not concern the local schedulability of components. IPS 1 blocks on the busy resource at time instant 227 and cannot continue further until the resource is released. However, the activations of the independent component, implemented by IPS 2, are unaffected, because IPS 3 can only execute 7.4 time units with a raised system ceiling, e.g. see time interval $[220, 235]$ where IPS 3 gets preempted after by IPS 1 that blocks on the busy resource and IPS 2 that continues its execution normally. Moreover, IPS 3 may even use its overrun budget to continue its critical section with a decreased system ceiling, see time interval $[273, 280]$, where IPS 3 is preempted by IPS 2. This is possible due

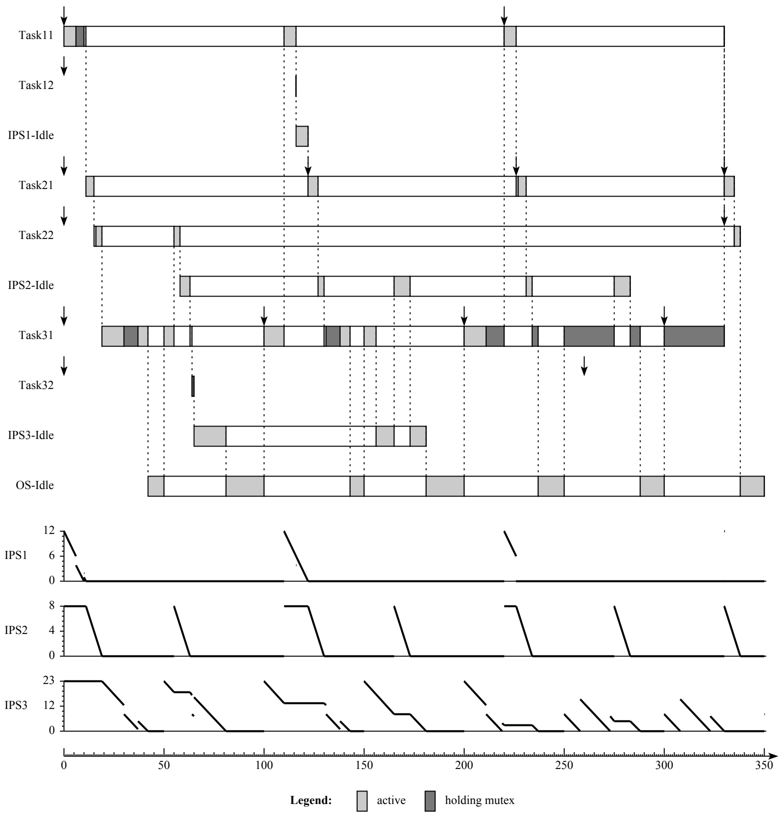

Fig. 3. Example trace, generated from instrumented code [29], combining HSRP and B-HSTP to arbitrate access between IPS 1 and IPS 3 to a single shared resource. IPS 2 is independent and continues its normal execution even when task 31 exceeds its specified critical-section length, i.e. starting at time 219. IPS 1 blocks on a busy resource and looses its remaining budget at time 227.

to the inherent pessimism in the overrun analysis [1], [19] which allocates an overrun budget at the global level without taking into account that in normal situations the system ceiling is raised for that duration.

8 Discussion

8.1 Component ceilings revisited

We assume locally non-preemptive critical sections, which may reduce the component's schedulability. Suppose we allow preemptions of tasks that are not blocked by an SRP-based component ceiling, see (2). The blocking times of all tasks with a lower preemption level than the component ceiling do not change, providing no advantage compared to the case where critical sections are non-preemptive. Moreover, enforcement of critical-section lengths is a prerequisite to guarantee temporal isolation in HSFs, see Section 4.

As a solution we could introduce an intermediate reservation level assigned and allocated to critical sections. In addition, we need to enforce that blocking times to other components are not exceeded due to local preemptions [12]. This requires an extension to our two-level HSF and therefore complicates an implementation. It also affects performance, because switching between multiple budgets for each component (or task) is costly [16] and breaks SRP's stack-sharing property.

8.2 Reliable resource sharing

To increase the reliability of the system, one may artificially increase the resource-access budgets, X_{sl}, to give more slack to an access of length c_{sil} to resource R_l. Although this alleviates small increases in critical-section lengths, it comes at the cost of a global schedulability penalty. Moreover, an increased execution time of a critical section of length c_{sil} up to X_{sl} should be compensated with additional budget to guarantee that the other tasks within the same component make their deadlines. Without this additional global schedulability penalty, we may consume the entire overrun budget X_s when we choose HSRP to arbitrate resource access, see Figure 3, because the analysis in [1], [19] allocate an overrun budget to each server period at the server's priority level. In line with [1], [19], an overrun budget is merely used to complete a critical section. However, we leave budget allocations while maximizing the system reliability as a future work.

8.3 Watchdog timers revisited

If we reconsider Figure 1 and Figure 3 we observe that the time-instant at which a maximum critical-section length is exceeded can be easily detected using B-HSTP, i.e. when a resource-access budget depletes. We could choose to execute an error-handler to terminate the task and release the resource at that time instant, similar to the approach proposed in AUTOSAR. However, instead of using expensive timers, we can defer the execution of such an error handler until component C_s is allowed to continue its execution. This means that the error handler's execution is accounted to C_s' budget of length Q_s. A nice result is that an eventual user call-back function can no longer hamper temporal isolation of other components than those involved in resource sharing.

9 Conclusion

This paper presented B-HSTP, an SRP-based synchronization protocol, which achieves temporal isolation between independent components, even when resource-sharing components misbehave. We showed that it generalizes and extends existing protocols in the context of HSFs [1], [2]. Prerequisites to dependable resource sharing in HSFs are mechanisms to enforce and monitor maximum critical-section lengths. We followed the choice in [1] to make critical sections non-preemptive for tasks within the same component, because this makes an implementation of our protocol efficient. The memory requirements of B-HSTP are lower than priority-inheritance-based protocols where tasks may pend in a waiting queue. Furthermore, B-HSTP primitives have bounded execution times and jitter. Both HSRP [1] and SIRAP [2], which each provide a run-time mechanism to prevent budget depletion during global resource access, have a negligible difference in implementation complexity when complemented with B-HSTP. Our protocol therefore promises a reliable solution to future safety-critical industrial applications that may share resources.

References

[1] R. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Real-Time Systems Symp.*, 2006, pp. 257–267.

[2] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems," in *Conf. on Embedded Software*, Oct. 2007, pp. 279–288.

[3] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symp.*, Dec. 2003, pp. 2–13.

[4] AUTOSAR GbR, "Technical overview," 2008. [Online]. Available: http://www.autosar.org/

[5] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Syst.*, vol. 9, no. 1, pp. 31–67, 1995.

[6] T. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, March 1991.

[7] Micrium, "RTOS and tools," March 2010. [Online]. Available: http://micrium.com/

[8] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Real-Time Systems Symp.*, Dec. 2005, pp. 389–398.

[9] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Protocol-transparent resource sharing in hierarchically scheduled real-time systems," in *Conf. Emerging Technologies and Factory Automation*, 2010.

[10] M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam, "Extending a HSF-enabled open-source real-time operating system with resource sharing," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010, pp. 71–81.

[11] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, "Implementation of overrun and skipping in VxWorks," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2010.

[12] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Real-Time Systems Symp.*, Dec. 2001, pp. 171–180.
[13] OpenCores. (2009) OpenRISC overview. [Online]. Available: http://www.opencores.org/project,or1k
[14] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronisation," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.
[15] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Symp. on Operating Systems Design and Implementation*, 1999, pp. 45–58.
[16] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *Euromicro Conf. on Real-Time Systems*, July 2005, pp. 89–97.
[17] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Trans. on Computers*, vol. 53, no. 12, pp. 1591–1601, Dec. 2004.
[18] Z. Deng and J.-S. Liu, "Scheduling real-time applications in open environment," in *Real-Time Systems Symp.*, Dec. 1997, pp. 308–319.
[19] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 1, pp. 93 –104, Feb. 2010.
[20] L. Almeida and P. Peidreiras, "Scheduling with temporal partitions: response-time analysis and server design," in *Conf. on Embedded Software*, Sept. 2004, pp. 95–103.
[21] M. Behnam, T. Nolte, M. Åsberg, and R. J. Bril, "Overrun and skipping in hierarchically scheduled real-time systems," in *Conf. on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 519–526.
[22] G. Buttazzo and P. Gai, "Efficient implementation of an EDF scheduler for small embedded systems," in *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.
[23] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *Parallel and Distrib. Processing Symp.*, 2007.
[24] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symp.*, Dec. 1988, pp. 259–269.
[25] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Real-Time Systems Symp.*, Dec. 2004, pp. 47–56.
[26] J. J. Labrosse, *Microc/OS-II*. R & D Books, 1998.
[27] M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, and J. J. Lukkien, "Constant-bandwidth supply for priority processing," *IEEE Trans. on Consumer Electronics*, vol. 57, no. 2, May 2011.
[28] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Real-Time Systems Symp.*, Dec. 2003, pp. 25–36.
[29] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2010, pp. 37–42.

Curriculum Vitae

Martijn Marianus Henricus Petrus van den Heuvel was born in Eindhoven, The Netherlands, on December 9th, 1984. After finishing secondary school at Heerbeeck College, Best, in 2003, he started his studies in computer science. He received the B.Sc. degree in computer science (2008) and the M.Sc. degree in embedded systems (2009) from the Technische Universiteit Eindhoven, Eindhoven, The Netherlands. During his studies he worked part-time in industry as a junior software engineer for two years. He also did a one-semester internship at Brandenburg University of Technology (BTU), Germany, where he conducted research on managing the quality-of-service (QoS) of video applications. After completing his master studies in 2009, Martijn started working towards the Ph.D. degree in the System Architecture and Networking (SAN) Group, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven. The main results of his research are presented in this dissertation. His main research interests are in the area of real-time embedded systems. He has published more than 20 papers in peer-reviewed fora (see below).

Accepted publications (April 2013)

Journal/Transactions

1) M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Transparent Synchronization Protocols for Compositional Real-Time Systems*, IEEE Transactions on Industrial Informatics (TII), pp. 322–336, vol. 8, issue 2, May 2012.
2) M.M.H.P. van den Heuvel, R.J. Bril, S. Schiemenz, C. Hentschel and C. Tempelaars, *Real-Time Priority Processing on an Embedded CE Device*, IEEE Transactions on Consumer Electronics (TCE), pp. 1969–1977, vol. 57, issue 4, November 2011.
3) M.M.H.P. van den Heuvel, M. Holenderski, R.J. Bril and J.J. Lukkien, *Constant-Bandwidth Supply for Priority Processing*, IEEE Transactions on Consumer Electronics (TCE), pp. 873–881, vol. 57, issue 2, May 2011.
4) M.M.H.P. van den Heuvel, R.J. Bril, S. Schiemenz and C. Hentschel, *Dynamic Resource Allocation for Real-time Priority Processing Applications*, IEEE Transactions on Consumer Electronics (TCE), pp. 879–887, vol. 56, issue 2, May 2010.

International refereed Conferences

1) M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Virtual scheduling for compositional real-time guarantees*, 8th IEEE International Symposium on Industrial Embedded Systems (SIES), in press, Porto, Portugal, June 2013.

2) I. David, M.M.H.P. van den Heuvel, R. Mak and J.J. Lukkien, *Resource-usage Modes Detection for Run-Time Resource Prediction of Video Components*, 31st IEEE International Conference on Consumer Electronics (ICCE), Digest of Technical Papers, pp. 167–168, Las Vegas, NV, USA, January 2013.
3) M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien, D. Isovic and G. Sankar Ramachandran, *RTOS support for mixed time-triggered and event-triggered task sets*, 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC), pp. 578–585, Paphos, Cyprus, Dec. 2012.
4) R.J. Bril, M.M.H.P. van den Heuvel, U. Keskin, J.J. Lukkien, *Generalized fixed-priority scheduling with limited preemptions*, 24th Euromicro Conference on Real-Time Systems (ECRTS), pp. 209–220, Pisa, Italy, July 2012.
5) S.A.B. Rao, T. Ozcelebi, M.M.H.P. van den Heuvel, R. Verhoeven and J.J. Lukkien, *Dependable Partitioning for Autonomous Agents in Adaptive Lighting Environments*, 30th IEEE International Conference on Consumer Electronics (ICCE), Digest of Technical Papers, pp. 435–436, Las Vegas, NV, USA, January 2012.
6) M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Dependable resource sharing for compositional real-time systems*, 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 153–163, Toyama, Japan, August 2011.
7) U. Keskin, M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien, M. Behnam and T. Nolte, *An engineering approach to synchronization based on overrun for compositional real-time systems*, 6th IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 274–283, Västerås, Sweden, June 2011.
8) M.M.H.P. van den Heuvel, M. Holenderski, R.J. Bril and J.J. Lukkien, *Constant-Bandwidth Supply for Priority Processing*, 29th IEEE International Conference on Consumer Electronics (ICCE), Digest of Technical Papers, pp. 884–885, Las Vegas, NV, USA, January 2011.
9) C. Tempelaars, M.M.H.P. van den Heuvel, R.J. Bril, S. Schiemenz and C. Hentschel, *Real-Time Priority Processing on the Cell Platform*, 29th IEEE International Conference on Consumer Electronics (ICCE), Digest of Technical Papers, pp. 157–158, Las Vegas, NV, USA, January 2011.
10) M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Protocol-Transparent Resource Sharing in Hierarchically Scheduled Real-Time Systems*, 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Bilbao, Spain, September 2010.
11) M.M.H.P. van den Heuvel, R.J. Bril, S. Schiemenz and C. Hentschel, *Dynamic Resource Allocation for Real-time Priority Processing Applications*, 28th IEEE International Conference on Consumer Electronics (ICCE), Digest of Technical Papers, pp. 67–68, Las Vegas, NV, USA, January 2010.

International refereed Workshops/Work-in-Progress

1) M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien, D. Isovic and G. Sankar Ramachandran, *Towards RTOS support for mixed time-triggered and event-triggered task sets*, Work-in-progress (WiP) session of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Kraków, Poland, September 2012.
2) M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte, *Opaque analysis for resource sharing in compositional real-time systems*, 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS), pp. 3–10, Vienna, Austria, November 2011.
3) M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Temporal isolation in an HSF-enabled real-time kernel in the presence of shared resources*, 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 39–48, Porto, Portugal, July 201.
4) C.G.U. Okwudire, M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Exploiting Harmonic Periods to Improve Linearly Approximated Response-Time Upper Bounds*, Work-in-progress (WiP) session of the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Bilbao, Spain, September 2010.
5) M. Holenderski, M.M.H.P. van den Heuvel, R.J. Bril and J.J. Lukkien, *Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems*, 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), pp. 37–42, Brussels, Belgium, July 2010.
6) M.M.H.P. van den Heuvel, R.J. Bril, J.J. Lukkien and M. Behnam, *Extending a HSF-enabled Open-Source Real-Time Operating System with Resource Sharing*, 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 71–81, Brussels, Belgium, July 2010.
7) M.M.H.P. van den Heuvel, R.J. Bril, P. van de Velde and J.J. Lukkien, *Towards Verification-based Development of In-Vehicle Safety Critical Software: A Case Study*, 1st ACM Workshop on Critical Automotive applications: Robustness and Safety (CARS), pp. 35–38, Valencia, Spain, April 2010.
8) M.M.H.P. van den Heuvel, M. Holenderski, W. Cools, R.J. Bril and J.J. Lukkien, *Virtual Timers in Hierarchical Real-time Systems*, Work-in-Progress (WiP) session of the 30th IEEE Real-time Systems Symposium (RTSS), pp. 37–40, Washington D.C., USA, December 2009.

Titles in the IPA Dissertation Series since 2007

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of

Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of As-*

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18