Mälardalen University Doctoral Thesis
No.191

Adaptive and Flexible Scheduling Frameworks for Component-Based Real-Time Systems

Nima Khalilzad

November 2015

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-4238
ISBN 978-91-7485-235-6
Printed by Mälardalen University, Västerås, Sweden

Populärvetenskaplig sammanfattning

Moderna datasystem är ofta utformade för att spela en mångsidig roll. De är därför kapabla till att köra flera mjukvarukomponenter (programvaror) samtidigt. Dessa mjukvarukomponenter delar systemresurser (t.ex. processorn och nätverket) under körning. Målet med mjukvarukomponentens körning är att avsluta sina beräkningar som förväntat. Vissa mjukvarukomponenter har även tidskrav vilket innebär att de inte bara kräver tillgång till systemresurser för att köra sina beräkningar, utan de har även krav på när denna tillgång sker för att mjukvarukomponenterna ska för rätt funktion kunna garantera att beräkningar utförs i rätt tid. Således finns det ett behov av att snabbt dela resurser mellan olika mjukvarukomponenter. Denna tidsdelning realiseras ofta genom att reservera en tidslucka för komponenten då den är tänkt att och får använda resursen. Reservationen måste vara tillräcklig för att mjukvarukomponenten ska kunna köra som förväntat. Reservationen måste även tilldelas resurseffektivt dvs resurstid får inte slösas bort i onödan. Genom en resurseffektiv reservation av resurser minskar komponentens fotavtryck på resursen som i sin tur möjliggör integration av flera programvarukomponenter på samma resurs. Denna avhandling fokuserar främst på resurseffektivitet i samband med reservationerna. Två fall behandlas. (I) Komponenter som tål att missa vissa enstaka tidskrav (så kallade mjuka realtidskomponenter): i det här fallet anpassas reservationerna under körning efter komponenternas ständigt föränderliga önskemål på reservationsstorlek. (II) Komponenter som inte kan hantera att tidskrav överträds (så kallade hårda realtidskomponenter): i det här fallet används flexibla strategier som möjliggör förbättrad resurseffektivitet redan vid design av systemet.

Abstract

Modern computer systems are often designed to play a multipurpose role. Therefore, they are capable of running a number of software components (software programs) simultaneously in parallel. These software components should share the system resources (e.g. processor and network) such that all of them run and finish their computations as expected. On the other hand, a number of software components have timing requirements meaning that they should not only access the resources, but this access should also be in a timely manner. Thus, there is a need to timely share the resources among different software components. The time-sharing is often realized by reserving a time-portion of resources for each component. Such a reservation should be sufficient and resource-efficient. It should be sufficient to preserve the timing properties of the components. Also, the reservations should be resource-efficient to reduce the components' footprint on the resources which in turn allows integration of more software components on a given hardware resource. In this thesis, we mainly focus on the resource-efficiency of the reservations. We consider two cases. (I) Components which can tolerate occasional timing violations (soft real-time components): in this case we adjust the reservations during run-time to match the reservation sizes based on the instantaneous requirements of the components. (II) Components which cannot tolerate any timing violations (hard real-time components): in this case we use flexible approaches which allow us to improve the resource-efficiency at the design time.

To Arefeh

Acknowledgments

First of all, I would like to offer my special thanks to my supervisors Prof. Thomas Nolte and Dr. Moris Behnam who have been supervising me from my master thesis. This thesis would not be possible without their support and encouragement. Thomas has always inspired me by his positive attitude that he brings to work, I also appreciate his incomparable support. I am particularly grateful for the useful critiques of Moris which have always improved my work. I also would like to thank Prof. Xue Liu for hosting me during my visit at McGill University.

Next, I wish to thank my coauthors for all the heated discussions which made the process of conducting research fun for me. Furthermore, I would like to thank my colleagues at IDT for the good company during courses, conference trips, PhD schools and/or lunches. Also, I wish to express my appreciation to the lecturers and professors at MDH who I have learned a lot from during my graduate courses. I would also like to thank IDT administration staff for their help with practical issues.

Last but not least, I would like to express my very great appreciation to my beloved wife, Arefeh, for the endless energy and love that she brings to my life. In addition, I wish to acknowledge my parents' and my brother's unsparing support.

Nima Khalilzad
Västerås, October, 2015

This work has been supported by the Swedish Research Council (Vetenskapsrådet) under the project ARROWS and the Swedish Foundation for Strategic Research (SSF) via the research project PRESS.

List of publications

Papers included in the PhD thesis[1]

Paper A *Bandwidth Adaptation in Hierarchical Scheduling Using Fuzzy Controllers*, Nima Khalilzad, Moris Behnam, Giacomo Spampinato and Thomas Nolte, In Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), June, 2012.

Paper B *An Adaptive Scheduling Framework for Component-Based Real-Time Systems*, Nima Khalilzad, Moris Behnam and Thomas Nolte, Under revision in the Journal of Systems and Software (JSS), Special Issue on Computers, Software, and Applications - Software Engineering in COMPSAC.

Paper C *A Feedback Scheduling Framework for Component-Based Soft Real-Time Systems*, Nima Khalilzad, Fanxin Kong, Xue Liu, Moris Behnam and Thomas Nolte, In Proceedings of the 21th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15), April, 2015.

Paper D *Adaptive Multi-Resource End-to-End Reservations for Component-Based Distributed Real-Time Systems*, Nima Khalilzad, Mohammad Ashjaei, Luis Almeida, Moris Behnam and Thomas Nolte, In Proceedings of the 13th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'15), October, 2015.

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

Paper E *Exact and Approximate Supply Bound Function for Multiprocessor Periodic Resource Model: Unsynchronized Servers*, Nima Khalilzad, Moris Behnam and Thomas Nolte, In ACM SIGBED Review special issue on the 5th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'12), Volume 10, Number 3, October, 2013.

Paper F *On Component-Based Software Development for Multiprocessor Real-Time Systems*, Nima Khalilzad, Moris Behnam and Thomas Nolte, In Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'15), August, 2015.

Additional papers, not included in the thesis

1. *Towards Energy-Aware Placement of Real-Time Virtual Machines in a Cloud Data Center*, Nima Khalilzad, Hamid Reza Faragardi and Thomas Nolte, In Proceedings of IEEE International Symposium on High Performance and Smart Computing (HPSC'15), August, 2015.

2. *Extended Support for Limited Preemption Fixed Priority Scheduling for OSEK/AUTOSAR-Compliant Operating Systems*, Matthias Becker, Nima Khalilzad, Reinder J. Bril and Thomas Nolte, In Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems (SIES'15), June, 2015.

3. *Towards Adaptive Resource Reservations for Component-Based Distributed Real-Time Systems*, Nima Khalilzad, Mohammad Ashjaei, Luis Almeida, Moris Behnam and Thomas Nolte, In ACM SIGBED Review special issue on the 7th Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'15), Volume 12, Number 3, June, 2015.

4. *Probabilistic Application Interfaces for Hierarchical Scheduling*, Nima Khalilzad, Meng Liu, Moris Behnam and Thomas Nolte, In Proceedings of the IEEE Real-Time Systems Symposium (RTSS'13) Work-in-Progress (WiP) session, December, 2013.

5. *Resource Sharing among Prioritized Real-Time Applications on Multiprocessors*, Sara Afshar, Nima Khalilzad, Farhang Nemati and Thomas Nolte, In ACM SIGBED Review special issue on the 6th International

Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'13), Volume 12, Number 1, February, 2015.

6. *Adaptive Hierarchical Scheduling Framework: Configuration and Evaluation*, Nima Khalilzad, Moris Behnam and Thomas Nolte, In Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'13), September, 2013.

7. *Towards Energy-Aware Multiprocessor Hierarchical Scheduling of Real-time Systems*, Nima Khalilzad, Juri Lelli, Giuseppe Lipari and Thomas Nolte, In Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13), Work-in-Progress (WiP) session, August, 2013.

8. *Multi-Level Adaptive Hierarchical Scheduling Framework for Composing Real-Time Systems*, Nima Khalilzad, Moris Behnam and Thomas Nolte, In Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13), August, 2013.

9. *Implementation of the Multi-Level Adaptive Hierarchical Scheduling Framework*, Nima Khalilzad, Moris Behnam and Thomas Nolte, In Proceedings of the 9th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT'13), July, 2013.

10. *Towards Implementation of Virtual-Clustered Multiprocessor Scheduling in Linux*, Syed Md Jakaria Abdullah, Nima Khalilzad, Moris Behnam and Thomas Nolte, In Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13), Work-in-Progress (WiP) session, June, 2013.

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

12. *Towards Adaptive Hierarchical Scheduling of Overloaded Real-Time Systems*, Nima Khalilzad, Thomas Nolte and Moris Behnam, In Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), Work-in-Progress (WiP) session, June, 2011.

13. *On Adaptive Hierarchical Scheduling of Real-time Systems Using a Feedback Controller*, Nima Khalilzad, Moris Behnam, Thomas Nolte and Mikael Åsberg, In Proceedings of the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'11), April, 2011.

Notes for the readers

This thesis contains two parts. The first part is the introductory part (Chapter 1 to 7). The second part includes six paper (Chapter 8 to 13). The contributions of the thesis is twofold. We present adaptive frameworks targeting soft real-time systems in the first four papers, i.e., Chapters 8 to 11. Chapter 12 and Chapter 13, however, present flexible frameworks targeting hard real-time systems. We recommend that readers study Chapter 4 before reading the first four papers, and similarly Chapter 5 before reading the last two papers for getting an overview of the frameworks. We also suggest readers to study Chapter 6 after reading all papers.

Note that we have used different notations and terminologies throughout the included papers. Therefore, it is important to read the modeling sections of the papers before their corresponding contribution sections.

Contents

I

Thesis

Chapter 1

Introduction

Complexity in the software domain has been growing rapidly. The complexity stems from the following two reasons. Firstly, the complexity of each individual functionality expected from a modern software system has been increased. Secondly, the number of functionalities performed by a software system has been escalated. For instance, thanks to the computational capacity of the recent hardware platforms, previously segregated software systems can now be integrated on a shared hardware platform. While this integration gives rise to the number of functionalities of the software system, the complexity of the integrated system is also increased. Taming this complexity in the design of software systems is of particular interest to ensure swift developments resulting in correct software systems. To this end, component-based software development provides means and techniques for developing complex software systems. This approach uses the divide and conquer principle. A complex software system is divided into a number of simpler software components. Each component is developed and validated separately. Finally, the components are integrated to build the target system. This approach also promotes reusability allowing integration of a validated component in several different systems.

When it comes to real-time systems, timing constraints of software components have to be considered at both the component development phase as well as the integration phase. In component-based systems, resource reservation techniques are often used to provide timing guarantees to the components (e.g., [1, 2]). In this approach each component is entitled to a particular resource reservation. The timing behavior of a component can be studied regardless of other components which will be integrated at the integration phase. This

is because other components do not affect the current component's reservation.

In this thesis, we use the word component to refer to run-time entities that implement the desired software functionalities. We consider component models in which a real-time software component comprises a set of real-time tasks, each task performing a specific functionality. A component also has an intra-component scheduler which coordinates task executions. The component executions, however, are coordinated by the inter-component scheduler. Therefore, the scheduling model is a hierarchical scheduling model. From the real-time scheduling perspective, the component scheduling problem is equivalent to the problem of creating adequate resource reservations for hosting the components. The adequate resource reservations provide resources to the components in such a manner that the timing requirements of the components are respected.

Real-time tasks can either have hard deadlines where deadline misses are absolutely unacceptable or they can have soft deadlines where occasional deadline misses can be tolerated. A hard real-time component is a component composed of hard real-time tasks. The size of processor reservations assigned to the hard real-time components is derived from the Worst-Case Execution Time (WCET) of the component's inner tasks. For instance in [3] and [4], targeting multiprocessor platforms, the authors provided analysis frameworks in which the reservation properties are extracted from intra-component schedulers and task parameters. Such analyses result in pessimistic allocations. The over-allocation is due to two reasons. Firstly, WCET is unlikely to happen in reality. Secondly, the analysis that derives the processor reservation sizes based on the WCET of tasks is pessimistic.

Soft real-time components are software components consisting of soft real-time tasks. When integrating soft real-time components, pessimistic allocations are not justifiable. This is because pessimistic allocations do not permit an efficient processor utilization. In addition, in a group of soft real-time tasks the processor demand is subjected to large variations during run-time. For instance, the execution time of video decoder tasks can significantly vary depending on the content of the video frames. As a result, the processor demand of a real-time component consisting of such dynamic tasks may change significantly during run-time. Therefore, assigning a fixed-size processor reservation (for instance based on the average processor demands) may result in an unacceptable number of timing violations.

The contributions of this thesis is twofold. Firstly, we target soft real-time components. We design frameworks in which the sizes of processor reservations allocated to the components are adjusted during run-time. The purpose

of adaptations is to deal with components' processor requirements dynamics. We refer to these frameworks as "adaptive frameworks". In this direction, we provide solutions for component-based systems running on single processors, multiprocessors as well as distributed systems. We use simulations as well as implementations for evaluating the proposed frameworks. Secondly, targeting hard real-time components running on multiprocessors, we focus on frameworks that provide design-time integration flexibility. We propose modeling and analysis methods to improve resource-efficiency of such frameworks. We use the term "flexible frameworks" for referring to such frameworks.

1.1 Outline of the thesis

The thesis outline is as follows. In Chapter 2 we provide a brief background of our work. We also present the assumed models (i.e., task, component and recourse models) in this chapter. We present the research goal as well as the research method in Chapter 3. We provide an overview of the adaptive frameworks in Chapter 4, while Chapter 5 presents an overview of the flexible frameworks. In Chapter 6, we summarize the contributions of this thesis and we provide a prospect of our work. An overview of included papers is presented in Chapter 7, while the included papers are presented in Chapters 8 to 13.

Chapter 2

Background

2.1 Real-time systems

Computational systems in which their correctness depend on both time and function are called *real-time systems*. In such systems, the timing behavior of the system is carefully analyzed to ensure its correctness. Thus, real-time systems have timing requirements that need to be fulfilled.

In real-time systems, different functionalities are realized through concurrent programs which are called *tasks*. Tasks often perform the same functionality repeatedly throughout the system's life-time. Each instance of a task execution is called a *job*. At each point in time, the number of jobs ready for execution may be more than the number of processors. Therefore, the jobs should be scheduled in such way that the timing requirements of the real-time tasks are met.

2.1.1 Hard real-time tasks

A group of real-time tasks in which violation of timing requirements result in a catastrophic consequence are called hard real-time tasks. Hence, when dealing with such tasks, the corresponding timing analysis will ensure that there will be absolutely zero timing violations. The task implementing the Anti-lock Braking System (ABS) of a car is an example of a hard real-time task in which incorrect timing may result in human losses.

2.1.2 Soft real-time tasks

In the context of soft real-time tasks, the violations of the timing requirement only result in performance degradations. Although timing violations are not desirable, occasional violations can be tolerated in such tasks. For instance, video players are considered as soft real-time tasks. This is because in such systems timeliness is crucial with respect to performance while incorrect timing has no catastrophic consequences such as loss of human lives.

2.1.3 Real-time component

A number of real-time tasks are often grouped together to perform a set of functionalities. Different terminologies are used in the literature to refer to such a group of tasks (e.g., subsystem, application, component, etc.). In this thesis, we use the term "component" to refer to such a task group[1].

2.2 Component-based real-time systems

Traditionally computational systems used to be single purpose systems. In other words, a single hardware platform was used to perform a small set of functionalities (tasks). However, advances in hardware technology enable the integration of several functionalities on a single hardware [5, 6]. When composing different systems on a single hardware, previously independent systems become components of the new system. In such integrated systems, it is desirable to perform the timing analysis compositionally, i.e., the timing correctness of the system should be inferred from the timing correctness of its components [7, 8, 9]. This approach facilitates the development process, and it promotes the component reusability. In such a component-based system the scheduling is often performed hierarchically [8, 9]. In this scheme, at the resource level, the inter-component scheduler schedules the components on the resource. Once a component is scheduled on the resource, the intra-component scheduler coordinates the execution of tasks on the resource. Figure 2.1 illustrates a component-based scheduling framework with two levels of hierarchy, three components, three tasks per component and one resource.

In the development of component-based softwares, the following two roles are often defined: (i) component developer; (ii) system integrator. The component developer is responsible for developing real-time tasks and selecting an

[1] Except paper A in which we use the term subsystem.

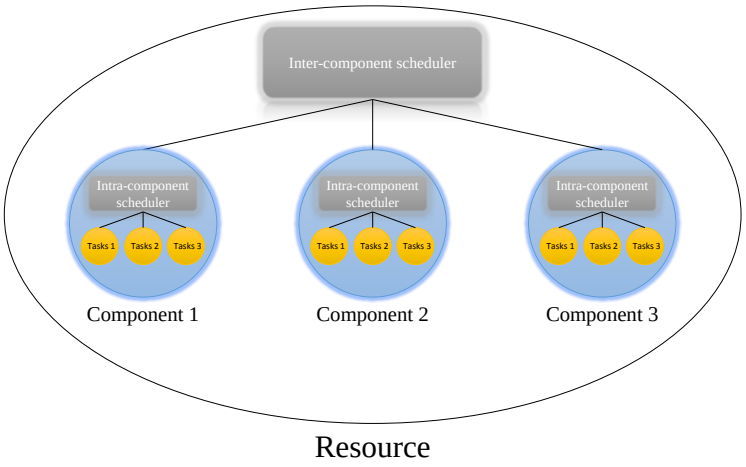


Figure 2.1: Component-based scheduling framework.

appropriate scheduling policy for them. Then, the component's timing requirements are abstracted using a number of interface parameters. For instance, the periodic resource model [7] uses two parameters (i.e. period and budget) for abstracting component's timing requirements. The system integrator, on the other hand, receives a number of components and (s)he is responsible for integrating the components such that the requirements specified in the interface parameters are respected.

2.2.1 Component-based scheduling frameworks

Component-Based Scheduling Frameworks (CBSF) provide means and techniques for developing and integrating real-time components. CBSFs often utilize a *resource reservation* scheduling technique. That is, the resource is often partitioned in the time domain, and each component is assigned to a partition (also known as a reservation). Note that we only target the processor resource and the network resource in this thesis. CBSFs provide guidelines for component developers on how to abstract the component requirements in a component interface. The component interface indeed provides specifications of the required resource reservation. CBSFs also provide integration techniques based on component interfaces for the system integrators. Such techniques ensure that the timing requirements of all integrated components are satisfied.

2.2.2 Hard real-time CBSFs

A subset of CBSFs target hard real-time components [1, 2, 10, 11]. In such frameworks the component interfaces are often derived using the following technique. The resource demand of the task set within the component is calculated given the WCET of tasks and the scheduling policy. Thereafter, an adequate reservation is derived such that the resource demand curve of the component always lies below the resource supply curve of the reservation. There are three sources of pessimism in the aforementioned schedulability analysis. (i) The over estimations in the WCETs; (ii) the pessimism that stems from the calculations of the resource demand curve; (iii) the pessimism that originates from the calculations of the resource supply curve of the reservation. The collective pessimisms caused by (ii) and (iii) is also referred as *abstraction overhead*. Although, the hard real-time nature of the components justifies such pessimism, it is still desirable to improve over resource-efficiency of such frameworks by eliminating (or mitigating) different sources of pessimism.

2.2.3 Soft real-time CBSFs

In the context of soft real-time systems, resource overallocation is not justifiable since occasional timing violations can be tolerated. Therefore, it is desirable to provide reservations based on the actual component demands rather than the worst-case demands. On the other hand, the resource demand of a set of soft real-time tasks may be highly variable during run-time. For instance, Figure 2.2 shows the distribution of processor demand percentage of a video decoder task. Consider soft real-time components consistent of such dynamic tasks. It is easy to see that any fixed reservation will not be able to efficiently serve such a component. A potential solution is to use frameworks which perform run-time adjustments of the reservations tracking the instantaneous component demands. For instance, such adaptive frameworks have been studied in the context of the AQuoSA [12] and the ACTORS [13] projects for simple component models (i.e. only one task per component).

2.3 Model

In this section we present a general model of the system used in this thesis. We present a more detailed model in each paper. In the following chapter (Section 3.1), using the following system model, we present the research goal and challenges.

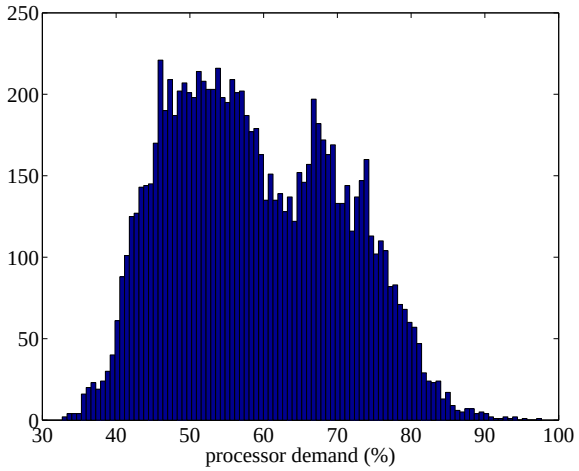


Figure 2.2: The distribution of processor demand percentage of a video decoder task [14].

2.3.1 Resources

Throughout the thesis we mainly focus on the processor resource except one paper in which we consider distributed systems, and we focus on both processor and network resources. We consider single processors as well as multiprocessors. The difference between a multiprocessor resource and a distributed system is the following. In multiprocessors different processing units are on the same chip, while in the case of distributed systems the processing elements are on separate chips connected using network links. We assume homogeneous processors. We use resource reservation techniques for scheduling the components on the resources, and we partition the resources in the time domain. Each time partition of the resources is called a resource reservation. We assign each component to a dedicated reservation.

2.3.2 Tasks and components

We assume periodic/sporadic task models in which a task is released within a minimum interarrival time. The tasks are run-time entities that perform a specific functionality. The tasks should finish their executions before their re-

spective deadlines, otherwise they violate their deadline requirements.

We assume that a component is composed of a set of tasks. The characteristics of the components are summarized in a set of *interface parameters*. The interface abstracts the timing requirements of the task set within the component. This abstraction of requirements facilitates the integration phase. This is because at the integration phase instead of dealing with the tasks, only the interfaces need to be considered. Hence, the complexity is reduced. Such a component model has received a great deal of attention in the past years (e.g. [1, 2, 15, 3, 4, 16]). When scheduling the components, our aim is to provide a resource-reservation compliant with the component's interface.

2.3.3 Scheduling scheme

We assume a hierarchical scheduling scheme. The intra-component scheduler schedules the tasks within a component. The inter-component scheduler, however, schedules the components. This scheduler uses reservation based scheduling policies. From a resource provisioning point of view, the inter-component scheduler divides the resource among the component, and the intra-component scheduler divides the component's share of the resource among its tasks.

2.3.4 Run-time adaptability versus design-time flexibility

In this thesis, we present four different CBSFs in which the resource reservations are adapted during run-time. We use the term "adaptive framework" to refer to such soft real-time CBSFs which perform run-time adaptations. Run-time adaptations can be seen as a continual component integration process where components submit their instantaneous requirements, and the integration mechanism adjusts the reservations accordingly.

On the other hand, we also study two hard real-time CBSFs. In these frameworks, the run-time reservations are fixed. We use the term "flexible framework" to denote design-time flexibility offered by such frameworks. Design-time flexibility allows system integrators to adjust the resource reservations based on the other components being integrated on the system. In an inflexible framework, the component interfaces impose rigid requirements on the properties of the reservations which does not allow any adjustment at the integration phase. For instance, assume we have a platform with two processors. Also, assume that 51 % of each processor is already occupied. Now, we want to integrate a new component which requires 50 % processor bandwidth. A

flexible interface would allow creating two reservation on each processor with the total processor bandwidth equal to 50 %. While an inflexible interface that rigidly requires the entire bandwidth from only one processor would cause the integration to fail in integrating the new component.

Chapter 3

Research Overview

In this chapter we present the research goal of this thesis followed by the research method used for reaching the goal.

3.1 Goal of the thesis

The goal of this thesis is:

To provide adaptive, flexible and resource-efficient frameworks for scheduling component-based real-time systems.

In the context of soft real-time systems, our goal is to adapt the reservation sizes, during run-time, in response to (i) dynamics of the components' resource requirements; (ii) variations in the overall load situation. The objective of the frameworks is to allocate a minimum amount of resources to the components while keeping their timing violations in an acceptable range. This acceptable range is inferred from the requirements of each particular application. Allocating a minimum amount of resource would in turn allow integrating more components on the same resource. We achieve our goal using feedback loops, and through monitoring the behavior of the components during run-time. The sizes of resource reservations are, then, adjusted based on the observed parameters (e.g. the number of timing violations).

In the context of hard real-time systems, we target multiprocessor hardware platforms. Multiprocessor platforms add a new dimension to the scheduling

problem. This is because run-time entities (tasks/components) can be allocated to different processors for performing their executions. Therefore, multiprocessor component-based scheduling frameworks should handle the processor allocation problem. In this context we target models that provide integration flexibility. An integration flexible model, from the vantage point of this thesis, allows adjusting the processor allocations, depending on the components that are currently integrated into the system. The integration flexibility is especially important in open systems where components can be added and removed during run-time. Therefore, our goal is to provide resource-efficient models that provide integration flexibility.

Following the main goal of the thesis, we have identified the following subgoals:

1. Enabling run-time adaptations of resource reservations allocated to soft real-time components assuming a simple hardware platform and a simple component model.
2. Extending Subgoal 1 to a more complex component model, i.e., a multi-level hierarchical component model.
3. Extending Subgoal 1 to more complex hardware platforms, i.e., multiprocessors and distributed hardware platforms.
4. Improving the resource-efficiency of flexible hard real-time component-based scheduling frameworks.

The first three subgoals focus on the run-time adaptability and resource-efficiency, while the last subgoal focuses on the design time flexibility and resource-efficiency. Table 3.1 shows the mapping between the subgoals and the included papers in the thesis.

3.2 Research method

In [17], Shaw has categorized the software engineering research paradigms into a few classes. Among those classes, our research lies into the "method/means" class. That is, a class of research in which the researcher is looking for new architectures for particular systems. The product of our research is a "technique" [17]. That is, we propose new development approaches along with new software architectures. Regarding the validation technique, we use implementations as well as simulations. In the following we elaborate the research process used for achieving each subgoal separately.

	Subgoal 1	Subgoal 2	Subgoal 3	Subgoal 4
Paper A	✓			
Paper B		✓		
Paper C			✓	
Paper D			✓	
Paper E				✓
Paper F				✓

Table 3.1: The relation between the subgoals and the included papers.

Figure 3.1 illustrates the generic research process [18] used in this thesis. We first (step A) formulated a subgoal. In this stage we also decided on the target system models (both software and hardware). In Step B, we designed a theoretical framework to address specific models decided in Step A. Note that Step A resulted in a set of hardware models, i.e., single processor, multiprocessor and distributed systems. Therefore, we proposed different frameworks for each hardware morel starting from a simpler model (single processors). Next, in Step C, we implemented the frameworks in a simulator and/or an operating system. Finally, we evaluated the results to analyze whether the subgoal is achieved or not. Steps B, C and D were performed in a loop, i.e., we iterated these steps until we achieved desirable results.

To achieve the first three subgoals of the thesis, our proposed frameworks provide run-time mechanisms to adapt the resource reservations based on the instantaneous component needs. Therefore, in step C, we implemented the frameworks to study their run-time behavior. We used simulations in step C of Paper A and D. While in Paper B we used Linux implementation, and in paper C we used both simulations as well as a Linux implementation. The particular choice of how to perform Step C was based on the convenience of implementing the proposed framework.

Regarding the forth subgoal, in paper E, we proposed an analytical framework (Step B) and we used formal proofs for validating it. In addition we implemented the calculation of the analysis in Step C. We, then, performed evaluations based on simulating the analysis for a set of input parameters to investigate a number of properties such as an average execution time for running the analysis. Finally, in Paper F, we provide a new development framework. In Step C, we implemented the analysis corresponding to the new framework, and in Step D we evaluated the abstraction overhead.

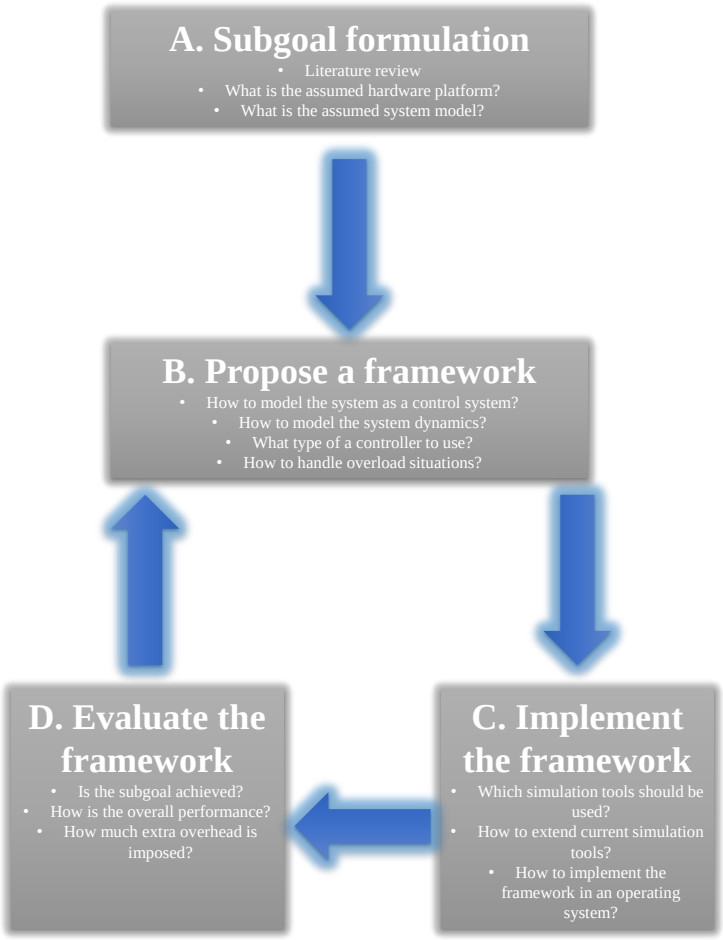


Figure 3.1: The research process used in this thesis.

Chapter 4

Adaptive Frameworks

In this chapter we provide an overview of the frameworks presented from Chapter 8 to Chapter 11. In the following sections we present the contributions related to the soft real-time components.

4.1 Enforcing resource reservations

In this thesis, we use server based scheduling to implement resource reservations. For instance we use the periodic servers [10] that are compliant with the periodic resource model [7]. These servers provide a given amount of the resource (denoted as the budget) every server period. Figure 4.1 illustrates the periodic servers. The server budget is replenished periodically to its maximum. Once the server is scheduled on the resource, its tasks may use the resource. The budget is decreased while the server holds the resource.

The mapping between the servers and the component varies in different chapters. In Chapter 8 and 9 we assume single processor resources, and thus we use a one-to-one mapping between servers and components. In Chapter 10, however, the framework is based on multiprocessor resources. In this chapter, a component can have multiple servers each running on a separate processor. Finally, in Chapter 11 we assume a distributed resource, and we assume that each component can have multiple servers each running on a separate resource. Note that in this chapter we explicitly consider network resources. Therefore, we perform resource reservation on the networks resources as well as the processor resources. The reservation enforcement on the network is implemented by the underlining network protocol.

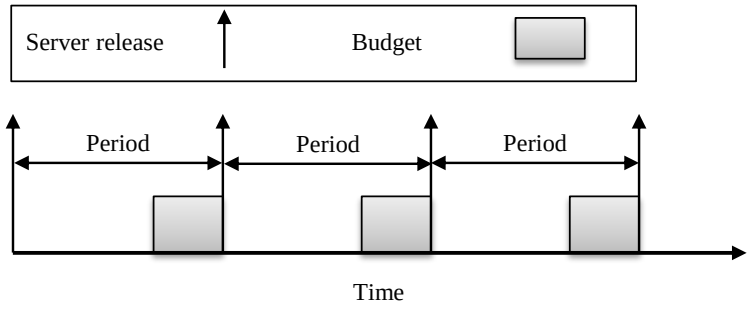


Figure 4.1: Resource reservation using the periodic servers. Note that in this figure the worst-case budget provisioning is illustrated.

4.2 Tracking the resource needs

The goal of the adaptive frameworks is to track instantaneous component requirements during run-time. Figure 4.2 illustrates this goal. In the figure the component demand is changing over the time. We illustrate three different reservations: (i) worst-case reservation; (ii) adaptive reservation; (iii) average-case reservation. The problem with (i) is that it overprovisions the resource, therefore it is not resource-efficient. Reservation (iii), on the other hand, underprovisions the resource in a number of time intervals. Thus, the component's inner tasks will suffer from deadline violations. The adaptive assignment tracks the actual needs and provides an adequate amount of the resource at each time point. We perform the tracking periodically at each adaptation point. Tracking relies on three elements: sensing, computing and actuating.

4.2.1 Sensing

At each adaptation point, we would like to understand the state of each component. That is, we would like to know if the component has received the resource more than what it required, less than what it needed or the resource provisioning was just sufficient. To this end, we monitor a number of parameters. For instance, we measure the amount of wasted budget, i.e. the budget that is given to the component but not used by its inner tasks.

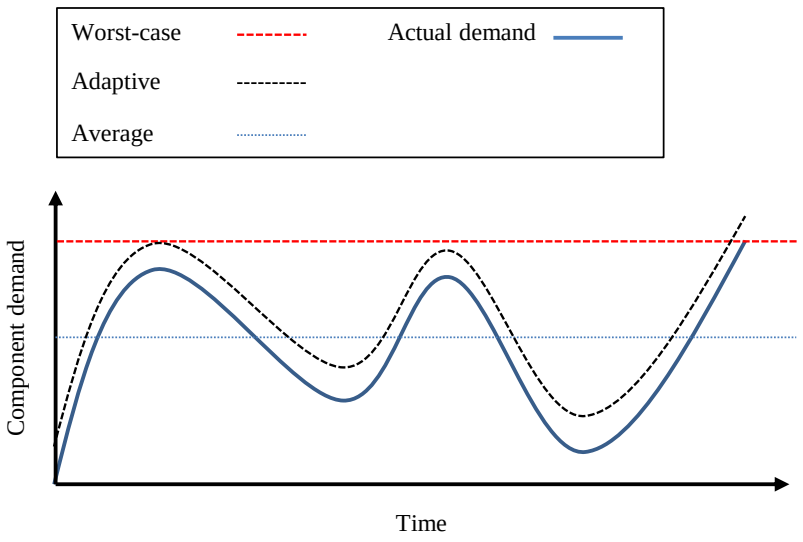


Figure 4.2: The goal of the adaptive frameworks is to track the actual component demands and adjust the server parameters based on the actual demands.

4.2.2 Computing

The next step is to decide how much resource we want to reserve for the component until the next adaptation point. We devise different techniques in each chapter for this purpose. In Chapter 8 we use a model free control approach using fuzzy controllers [19]. In Chapter 9 we use an ad hoc approach which works based on estimation and compensation. It estimates the amount of next workload based on the previous observed workloads. The reservation size is calculated by adding the amount of backlog to the estimated workload. In Chapter 10 and Chapter 11 we approximate the component's workload dynamics using linear models, and we use optimal controllers [20] for deriving the reservation properties.

4.2.3 Actuating

The periodic servers are characterized using two parameters: period and budget. Therefore, we have two options for performing the actuations. The reser-

vation bandwidth can be modified by manipulating either of the above parameters. In Chapter 8, 9 and 11 we perform actuations by modifying the amount of budget. In Chapter 10, however, we modify both budgets and periods simultaneously.

4.3 Performance metrics

We take the number of deadline violations as our Quality of Service (QoS) metric. The lower the number of deadline misses the better the performance. The number of deadline misses is often expressed in the form of deadline miss ratio, which is the number of missed deadlines divided by the total number of jobs. On the other hand, we want to be resource-efficient. That is, our objective is to reach a low number of deadline misses while keeping the resource reservation sizes to a minimum. Thus, the amount of wasted (idled) budget by the servers is another performance metric.

4.4 Evaluation environment

We have evaluated the proposed framework using two approaches: (i) simulation studies (ii) implementation studies. Since there was no available framework which implements/simulates the behavior of an adaptive hierarchical scheduling framework, we had to either extend the available frameworks or to implement a new framework. In the case of using a simulation framework, we extended the TrueTime [21] simulator. However, for the implementation studies we implemented a new framework from scratch. The evaluation environment setup comprises approximately 30 % of the time spent for completing this thesis. In this section, we provide an overview of the evaluation environments used in the thesis.

4.4.1 TrueTime

TrueTime is a simulator that uses the infrastructure available in Matlab and Simulink. This tool is suitable for simulating the behavior of control tasks. TrueTime supports different scheduling algorithms. When it comes to reservation based algorithms, TrueTime supports the Constant Bandwidth Server (CBS) [22] and also the hard CBS [23]. We use the hard CBS to implement the periodic server [10]. Basically, we run an idle task inside each reservation to mimic the behavior of periodic servers. In addition, we have added a local

scheduler inside the hard CBS such that the tasks attached to a hard CBS are scheduled according to the EDF scheduling policy.

4.4.2 Linux implementation

We have implemented two schedulers inside the Linux kernel for evaluating the two frameworks presented in Chapter 9 and 10. Both schedulers are implemented as loadable kernel modules. When the real-time tasks attach themselves to our module, then the Linux kernel delegates the scheduling of these tasks to our scheduler. We provide more details about this module in Chapter 9 (Paper B).

4.5 Related work

In the following we review a few adaptive frameworks related to the adaptive frameworks of this thesis. In the literature, the term "feedback scheduling" is often used to refer to the scheduling schemes in which the scheduling parameters are adapted using feedback control loops. We also present adaptive frameworks that use reservation-based scheduling policies. Such frameworks are closer to the contributions of this thesis since we use resource reservation techniques.

4.5.1 Feedback scheduling of real-time systems

Feedback control has found its way in computing systems for helping system designers to deal with uncertainties and dynamicity. For instance, in high-performance computing, load is unpredictable and dynamic. Gandhi *et al.* used a MIMO controller to control CPU and memory utilizations in an Apache web server [24]. Diao *et al.* used a MIMO LQR controller to solve a load balancing problem [25]. The controller equalizes the load among different resources to improve response times as well as the throughput.

In the context of real-time scheduling, Lu *et al.* proposed a feedback scheduling scheme to cope with unpredictable workloads [26]. In their framework the deadline miss ratio and the system utilization is used as sensors, while the admission control is used as an actuator. The problem of task reweighting under multiprocessor scheduling algorithms is studied by Block *et al.* in [27] and [28]. In these papers it is assumed that, tasks ask for a new processor utilization during run-time. A number of reweighting rules for partitioned and global scheduling algorithms are presented. Block *et al.* combined task

reweighting with feedback loops that estimate the weight of the next job [29]. In distributed real-time systems, utilization control is performed through rate adaptation to provide QoS guarantees [30]. Stankovic *et al.* presented a framework in which the service levels are adapted based on monitoring the number of deadline misses and the processor utilizations [31]. Utilization control is coupled with processor frequency adjustment in [32] and [33]. Targeting end-to-end task models, DEUCON [34] employs a decentralized approach in which task rates (periods) are adapted using MIMO model predictive controllers. The control objective is to minimize the difference between the utilization set points and current utilizations. The main difference of the frameworks presented in this thesis with the aforementioned works is the following. Since we consider component-based systems in which a component is comprised of a set of tasks, a reservation-based scheduling policy is needed to isolate the timing behavior of the components in run-time. While this separation of run-time behavior for components is not supported by the above frameworks.

4.5.2 Adaptive reservations

Adaptive reservation schemes, first introduced by Abeni *et al.* [35], are powerful approaches for controlling the amount of processor allocated to individual tasks that demonstrate dynamic processor requirement. The mathematical model of a such scheme using CBS is derived in [36]. PI controllers are used for controlling the bandwidth of CBS. Cucinotta *et al.* used stochastic controllers for the same purpose [37]. Regarding adaptive reservations in which multiple parameters are adapted, in [38] both periods and budgets of the CBS are adapted. This framework targets legacy tasks which do not communicate with the scheduler. Two different modules are used (i) period detector (ii) budget estimator. One centralized controller is used for adapting the periods and the budgets. In the context of the ACTORS project [13], a cascade controller is used on top of processor reservations for adapting their bandwidths. Our work is different from the above reservation-based approaches in the following main aspect. We consider a more general component model in which multiple tasks may be in a single component. In our model, the intra-component scheduler coordinates the execution sequence of the tasks inside a component. Hence, the control input used by the above frameworks is not applicable to our model. Our general model indeed provides a higher degree of flexibility for component developers since they can include several tasks inside one component. This component model has been widely used by several researchers (e.g. [1, 2, 15, 3, 4, 16]).

4.5.3 Resource reservations on network

In Chapter 11 we present a framework which uses resource reservation techniques on network. Therefore, we review the related literature to this line of work. A general category of the resource management in network is traffic shapers [39]. The purpose of these shapers is to limit the amount of traffic that a node submits to the network in a given time interval. Similar to the techniques used in processor reservations, the traffic shapers use methods based on capacity which is replenished with different policies, e.g. credit-based shaping in Ethernet AVB. Moreover, some real-time Ethernet protocols enforce a cyclic-based transmission and reserve windows for different classes of traffic (e.g., Ethernet POWERLINK [40], FTT-SE [41] and HaRTES [42]). Also, a hierarchical server model [43] is proposed for the Ethernet switches in the context of the FTT-SE protocol to reserve a portion of bandwidth for different traffic types, hence providing temporal isolation among them. An online QoS management [44] is proposed in the context of a multimedia real-time application, which adapts the video compression parameters and the network bandwidth reservations to provide the best possible QoS to the streams.

In order to reserve resources for streams in the network several protocols have been proposed, where they use similar concepts. For instance, Stream Reservation Protocol (SRP) [45] defines a set of procedures to reserve network resources for the specific traffic streams, which are crossing through an Ethernet Audio Video Bridging (AVB) network. The SRP protocol forces the traffic to be registered on the AVB switches through its path, before being transmitted.

4.5.4 Resource reservations in distributed systems.

Few authors have addressed the end-to-end reservation of resources for distributed systems, including processor and network resources. A distributed kernel framework with a resource manager in each node has been designed and implemented to provide an end-to-end timeliness guarantee [46]. Also, a resource management system, called D-RES [47], has been developed to handle shared resources among multiple applications in distributed systems. Cucinotta *et al.* presented a model in which a pipeline task is considered [48]. Tasks may use one of the resources available in the system to carry on their computations. Adaptive CBS is used to track the resource demand of the tasks. In addition, a general model, called Q-RAM [49], has been developed to manage the resources shared among multiple applications. The applications in this framework have different operation levels with different qualities depending

on the available resources. However, they have to satisfy their needs such as timeliness, reliability and data quality. The model allocates the resources to the applications considering that the overall system utility becomes maximum while the applications meet their minimum needs. This model has been extended in [50] for the systems with rapidly changing resource usage.

The main difference of our framework, presented in Chapter 11, with [48] and [49], is that we consider adaptation for components which may in turn be composed of multiple tasks. The existence of multiple tasks inside one component makes the system dynamics model in those works inapplicable to our setting. Besides, in our framework we explicitly consider network resources, and we use a common network technology for evaluating our framework.

Chapter 5

Flexible Frameworks

In this chapter we present an overview of the flexible frameworks designed for the hard real-time components. These frameworks are presented in Chapter 12 and 13.

5.1 Component-based development for multiprocessor platforms

The processes of developing component-based software with hard real-time requirements is often performed in two steps: (i) component development; (ii) component integration. The timing requirements of the component is abstracted in a number of interface parameters. At the component integration phase, only the component interfaces are considered. The component interfaces impose requirements on the bandwidth that needs to be assigned for that particular component. In single processor platforms, the system integrator needs to reserve the specified bandwidth by the component on the processor. In multiprocessor platforms, however, a new dimension to the problem rises. The component bandwidth may be provided by multiple processors if it is allowed by the interface. The component interface may be rigid by imposing rigid bandwidth requirements. For instance, the interface may denote that the component requires two full processors and one partial processor with 25 % bandwidth. In this case the following problem may happen. The system may have a total slack bandwidth more than 225 %, however, the slack bandwidth may be spread over four processors. On the other hand, the interface may

be flexible, i.e., it may only specify how much total bandwidth the component needs without imposing constraints on the exact allocations. Consider the above example. A flexible interface will only specify that the component needs 225 % of the multiprocessor time. It is easy to see that in the former case (rigid interface), the integration fails, while in the latter case (flexible interface) the integration can be performed successfully.

5.2 The MPR model

The Multiprocessor Periodic Resource (MPR) model [3] provides a flexible abstraction method for components. In this model the component interface specifies the total bandwidth requirements as well as the maximum number of processors that can contribute to the total bandwidth. This model provides a great deal of flexibility at the integration phase because the total required bandwidth can be allocated based on available slacks on the multiprocessor. We have identified two issues with this model, and in this thesis we extend this model in two direction. Firstly, as Lipari and Bini state in [4], this model requires a synchronization mechanism among the processors of the multiprocessor platform. In Chapter 12 we present an analysis framework to address this problem. Secondly, we observed that the MPR model may impose large abstraction overhead. That is, the difference between the processor bandwidth of the task set within the component and the specified bandwidth in the interface may be large. Therefore, in Chapter 13, we present a new framework which extends the MPR model to cope with this problem.

5.2.1 Unsynchronized processors

In modeling component-based real-time systems, the system model often consists of two parts: resource supply model and component demand model. The resource supply model abstracts the underlying hardware resource such that each component has the illusion of running solo on an independent hardware. The resource supply model represents the minimum amount of resource that a reservation provides in a given time interval. The amount of provided resource is often represented using a Supply Bound Function ($\texttt{sbf}(t)$). The resource demand model, however, represents the resource demand of the tasks within the component. Similarly, the maximum demand is often represented using a Demand Bound Function ($\texttt{dbf}(t)$) [51]. Consequently, the schedulability test is performed using the $\texttt{sbf}(t)$, which is dependent on the resource model, and

the $\texttt{dbf}(t)$ which is dependent on the scheduling policy.

The sbf function must provide the absolute worst-case resource provisioning. As Lipari and Bini state in [4], in multiprocessor platforms the $\texttt{sbf}(t)$ depends on the fact that whether different processors are synchronized together or not. The MPR model [52] implicitly assumes that the processors are synchronized. While synchronization on some hardware platforms can be expensive, therefore, in Chapter 12 we simplify the implementation phase of the composition for the system developers by relaxing this assumption. Our proposed frameworks works as follows. Given an MPR interface, we construct all possible combination of the distribution of the total budget on different processors. We call each combination a possible platform. Thereafter, for each possible platform, the overall sbf is equal to the sum of all single processor sbf of the processors involved in the budget provisioning. The MPR sbf at each point in time is the least of the sbf of all possible platforms.

5.2.2 Extended MPR

The schedulability of the entire component-based system is examined using the component interfaces. In this regard, it is desirable to use the same schedulability techniques used for studying the schedulability of real-time tasks, and investigate the schedulability of the components. However, the task schedulability tests cannot be directly applied to the components for which their interface utilizations are more than 100 % of a single processor (i.e. one). This is because the basic assumption in all of the schedulability tests is that the task utilization is less than or equal to one. Therefore, components with interface utilization more than one have to be decomposed to smaller subcomponents with utilization less than or equal to one. The component schedulability test, then, can be performed using the decomposed subcomponent interfaces.

In all of the proposed approaches for developing component-based real-time systems on multiprocessors (e.g. [52, 4, 16]) the component decomposition is performed after abstracting the components. In Chapter 13, we investigate an alternative approach. We first decompose components for which their utilization is more than one. This step gives us a number of subcomponents. Thereafter, we abstract the component processor requirements using the abstraction technique proposed in [7]. We show that, using extensive simulations, performing the decomposition before abstraction significantly reduces the abstraction overhead. In Chapter 13 we present a new interface model (we call it the extended periodic resource model) in the form of a matrix. In this matrix, cell $\{i, j\}$ (row i column j) denotes the amount of budget required by

subcomponent j given that i processors will contribute in processor provisioning. At the integration phase, the system integrator has to select one row for each column, i.e., one budget for each subcomponent. Easwaran *et al.* [52] showed that smaller the number of processors contributing in the budget provisioning (i) the more efficient the interface. To this end, at the integration phase, the upper rows of the interface matrix are more desirable for selection. We, in Chapter 13, present integration algorithms which receive a set of component matrices and select a suitable row for each subcomponent.

5.3 Related work

Several component-based scheduling frameworks have been proposed in the real-time scheduling community. The basic idea behind most of these frameworks is to time partition the processor, and assign each partition to a single component. In doing so, the components are isolated with respect to their timing behavior. A timing anomaly in one component will not be propagated to the other ones. In addition, the timing behavior of systems can be studied only by investigating the properties of the processor partitions rather than analyzing the task parameters. Several modeling techniques have been proposed for time partitioning the processor on single processors and multiprocessors.

5.3.1 Single processors

Since Deng and Liu [1] presented a two level hierarchical scheduling framework, there has been a growing attention for using hierarchical scheduling in complex real-time systems. Schedulability analysis for two-level frameworks is presented by Kuo and Li [53]. For EDF-based global schedulers, a schedulability analysis is presented by Lipari and Baruah [54, 55]. In addition, the virtual processor model is presented in [56, 7]. Different schedulability analyses under fixed priority scheduling [57, 58] and EDF [11, 7] are presented. In single processors two parameters affect the amount of time provided by the processor partition to the components. (i) the granularity of processor partitioning; (ii) the utilization (also referred as bandwidth) of the processor partition. The bounded delay abstraction model, introduced in [56], specifies the bandwidth and the maximum blackout time of the processor supply. The maximum blackout time indicates the largest time interval that the processor may be unavailable. This resource model targets single processor platforms. Shin *et al.* present another abstraction model for the processor supply of single proces-

sors, namely the Periodic Resource (PR) model [7]. The PR model specifies period Π and budget Θ in its interface meaning that the processor becomes available every Π time units for duration of Θ time units. In this model Π specifies the time granularity of processor provisioning, while both Π and Θ specify the utilization of the processor partition.

5.3.2 Multiprocessors

When it comes to multiprocessors, three parameters play roles in the amount of the processor provisioning at each time point. In addition to the time granularity and utilization, the number of processors involved in contributing the overall utilization influence the processor provisioning. Leontyev and Anderson [59] proposed a model that only specifies bandwidth w in the component interface. In this model $\lfloor w \rfloor$ dedicated processors are assigned to the components and the remaining $w - \lfloor w \rfloor$ bandwidth is provided using a periodic server. This model provides limited flexibility at the integration stage for the system integrator as it requires $\lfloor w \rfloor$ dedicated processors. The MPR model [3] extends the PR model to multiprocessor platforms. The MPR model specifies budget Θ, period Π and number of processors m' in its interface. The MPR model guarantees provisioning of Θ processor time every Π time units using maximum m' physical processors. Xu *et al.* proposed the Deterministic MPR (DMPR) model in [60]. This model is different from the MPR model in the following aspect. The DMPR model, similar to [59], allows at most one partial processor allocation. Xi *et al.* [61] have investigated the application of the MPR modeling technique in the Xen virtual machine manager.

Bini *et al.* presented the Multi Supply Function (MSF) model in [62] for modeling the resource supply in hierarchical scheduling on multiprocessor platforms. The MSF is indeed a set of supply functions, one associated with each server. The Parallel Supply Function (PSF) model [63] is also proposed as an alternative for modeling the resource supply of hierarchical multiprocessor systems. This model indicates a set of supply functions where each of them represent the minimum available supply at a certain parallelism level. Since the MPR model offers a greater deal of abstraction compared to the MSF and the PSF models, from a system integrator perspective, the MPR can be more suitable when composing real-time systems. Lipari and Bini [4] suggested a new interface model, namely the Bounded-Delay Multipartition (BDM) model. The BDM interface consists of m, Δ and [β_1,...,β_m] parameters which represent the number of virtual processors, the blackout duration and the bandwidth at each parallelism level respectively. In fact, the DBM model replaces the notion of

period Π in the MPR with delay (the longest interval with no resource) Δ. The BDM model does not require the servers to be synchronized. Nevertheless, due to the nature of the delay based models, the BDM can be very pessimistic which can result in low system utilization and consequently higher cost of the system production. Besides, from an implementation point of view, periodic servers are more straight forward to implement, and the BDM model perhaps should be mapped to the MPR or any other periodic server based model for the implementation. Zhu *et al.* have extended deferrable servers to the context of multiprocessor platforms [64] where m deferrable servers with a common period and different budgets are running on m processors. In this thesis we extend the MPR model in the following two aspects: (i) we provide a more resource efficient abstraction; (ii) we relax the synchronization assumption between different processors.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we present frameworks for scheduling real-time software developed using the component-based development paradigm. The frameworks use the resource reservation techniques for scheduling the components. We consider two types of real-time components: (i) soft real-time components; (ii) hard real-time components. In the former case, our adaptive frameworks match the resource reservations according to the instantaneous component needs during run-time. In this context, we consider single processor, multiprocessor and distributed resource models. We use adaptation mechanisms based on fuzzy controllers, estimation-based approach and optimal controllers.

In the context of hard real-time systems, on the other hand, we only consider multiprocessor platforms. We extend the current models in the following two directions. First, we relax the assumption of processor synchronization by proposing a new approach to derive the worst-case resource provisioning. Furthermore, we improve the resource efficiency of such component-based development paradigms by proposing a new development approach. In our approach, we decompose large components into a number of smaller subcomponents before abstracting their resource requirements.

6.2 Discussion

The research presented in this thesis lies into the "method/means" class of research [17]. In other words, we present new frameworks that facilitate the software development practice for real-time software systems. Throughout the process of designing the presented frameworks, we have made several design decisions, e.g., we have selected particular control variables. We believe a new line of research that considers different design decisions and compares them against each other would be beneficial. Such line of research would lie in the "selection" research class [17]. A particularly important challenge in designing adaptive frameworks is the challenge of modeling system dynamics. We have investigated fuzzy modeling and simple linear models along with off-line and on-line parameter identification approaches in this thesis. It would be beneficial to investigate using more complex non-linear models to see whether they can provide a better performance or not.

Regarding the hard real-time frameworks, common global multiprocessor scheduling algorithms (gEDF and gFP) allow building flexible resource abstraction models without the need to specify the component's bandwidth distribution. However, the significant overhead imposed by the schedulability analysis of such global multiprocessor scheduling algorithms, limits the applicability of such flexible models. Therefore, we believe it is beneficial to devise models (similar to what we present in Chapter 13) that allow the trade-off between the integration time flexibility and resource efficiency by considering partitioned scheduling algorithms.

6.3 Future work

The work presented in this thesis can be extended in a number of directions. Regarding the adaptive frameworks, the following paths can be further explored.

- In the context of a multiresource adaptive framework running on a distributed resource infrastructure, we did not investigate scenarios in which different components compete with each other on acquiring the shared resources. Our proposed framework can be extended to incorporate a module which handles such scenarios.

- As mentioned in the discussions, a possible extension of our adaptive frameworks is to investigate taking different design decisions. In this di-

rection, for instance, we can consider (i) more complex nonlinear models for modeling the system dynamics; (ii) model predictive controllers for performing adaptation.

- We only considered adaptation by the means of adjusting the resource reservations. It would be interesting to incorporate other adaptation means such as bandwidth borrowing and bandwidth reclamation into our framework.

In the context of flexible frameworks, the following open directions may be explored.

- Our development approach presented in Chapter 13 can be extended to incorporate resource sharing into account. In this regard, new component decomposition algorithms are perhaps needed which tries to assign tasks sharing a resource into one subcomponent.
- We only considered EDF scheduling policy. It would be interesting to investigate other scheduling algorithms to see whether the overhead reduction is still significant using our approach.

Chapter 7

Overview of the Papers

7.1 Contributions

The contributions of the thesis are presented in the form of a collection of papers. The following papers are included in the thesis.

7.1.1 Paper A

Bandwidth Adaptation in Hierarchical Scheduling Using Fuzzy Controllers, Nima Khalilzad, Moris Behnam, Giacomo Spampinato, Thomas Nolte, In Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), June, 2012.

Abstract: *In our previous work, we have introduced an adaptive hierarchical scheduling framework as a solution for composing dynamic real-time systems, i.e., systems where the CPU demand of their tasks are subjected to unknown and potentially drastic changes during run-time. The framework uses the PI controller which periodically adapts the system to the current load situation. The conventional PI controller despite simplicity and low CPU overhead, provides acceptable performance. However, increasing the pressure on the controller, e.g., with an application consisting of multiple tasks with drastically oscillating execution times, degrades the performance of the PI controller. Therefore, in this paper we modify the structure of our adaptive framework by replacing the PI controller with a fuzzy controller to achieve better performance. Furthermore, we conduct a simulation-based case study in which we compose dynamic tasks such as video decoder tasks with a set of*

static tasks into a single system, and we show that the new fuzzy controller outperforms our previous PI controller.

My contribution in this paper: I was the main driver of the work. The co-authors contributed by discussions and reviewing the paper.

This paper addresses Subgoal 1.

7.1.2 Paper B

An Adaptive Scheduling Framework for Component-Based Real-Time Systems, Nima Khalilzad, Moris Behnam, Thomas Nolte, This paper is under second revision in the Journal of Systems and Software (JSS), Special Issue on Computers, Software, and Applications - Software Engineering in COMPSAC.

Abstract: *Processor partitioning techniques have been widely used for scheduling component-based hard real-time systems. Due to the safety critical nature of hard real-time systems, conservative partition sizes are often reserved for the components. A considerable capacity of the processor is wasted using such conservative techniques. When designing a component-based soft real-time system, however, conservative partitioning is unacceptable, because occasional timing violations can be tolerated by such systems. In this paper, we present a multi-level adaptive hierarchical scheduling framework for scheduling component-based real-time systems. In our framework, for efficiently utilizing the processor capacity, we adapt the partition sizes of soft real-time components based on their actual needs at run-time. The adaptation is based on on-line monitoring of the processor demand of the components. We have implemented our framework in the Linux kernel. We present the implementation details of our framework. Finally, we report our evaluation results.*

This paper is based on our conference paper published in RTCSA'13 and a workshop paper published in OSPERT'13. In the above papers we presented a multilevel hierarchical scheme for scheduling component-based systems. Our model allows multilevel of hierarchy, i.e., a component may be composed of a number of subcomponents. We implemented the scheme in the Linux kernel for performing evaluations. We also measured the overhead of our scheme.

My contribution in this paper: I was the main driver of the work. The co-authors contributed by discussions and reviewing the paper.

This paper addresses Subgoal 2.

7.1.3 Paper C

A Feedback Scheduling Framework for Component-Based Soft Real-Time Systems, Nima Khalilzad, Fanxin Kong, Xue Liu, Moris Behnam, Thomas Nolte, In Proceedings of the 21th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15), April, 2015.

Abstract: *Component-based software systems with real-time requirements are often scheduled using processor reservation techniques. Such techniques have mainly evolved around hard real-time systems in which worst-case resource demands are considered for the reservations. In soft real-time systems, reserving the processors based on the worst-case demands results in unnecessary over-allocations. In this paper, targeting soft real-time systems running on multiprocessor platforms, we focus on components for which processor demand varies during run-time. We propose a feedback scheduling framework where processor reservations are used for scheduling components. The reservation bandwidths as well as the reservation periods are adapted using MIMO LQR controllers. We provide an allocation mechanism for distributing components over processors. The proposed framework is implemented in the TrueTime simulation tool for system identification. We use a case study to investigate the performance of our framework in the simulation tool. Finally, the framework is implemented in the Linux kernel for practical evaluations. The evaluation results suggest that the framework can efficiently adapt the reservation parameters during run-time by imposing negligible overhead.*

In this paper we proposed an adaptive framework for multiprocessor platforms. We used system identification for deriving the model for dynamics of the components.

My contribution in this paper: I was the main driver of the work. The co-authors contributed by discussions and reviewing the paper. Fanxin helped in the optimization formulation and optimality proof of one of the algorithms.

This paper addresses Subgoal 3.

7.1.4 Paper D

Adaptive Multi-Resource End-to-End Reservations for Component-Based Distributed Real-Time Systems, Nima Khalilzad, Mohammad Ashjaei, Luis Almeida, Moris Behnam and Thomas Nolte, In Proceedings of the 13th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'15), October, 2015.

Abstract: *Complexity in the real-time embedded software domain has been growing rapidly. The component-based software development approach facilitates the development process of such software systems by dividing a complex system into a number of simpler components. Resource reservation techniques have been widely used for providing resources to real-time software components. In this paper we target real-time components operating on a distributed infrastructure. Furthermore, we target a class of software components which demonstrate dynamic resource consumption behavior. A prime example of such components is a multimedia software component. In the paper we present a framework supporting multi-resource end-to-end resource reservations. We reserve resource bandwidths on both processor resources as well as on the network resources. The proposed framework utilizes a Multiple Input Multiple Output (MIMO) controller which adjusts the sizes of reservations tracking the dynamic resource demands of the software components. Finally, we present a case study using a multimedia component to demonstrate the performance and efficiency of our framework.*

This works extends the contributions of the thesis towards distributed systems. We use the same component model as our previous work, while we assume the component may be spread over a distributed system.

My contribution in this paper: I was the main driver of the work. The co-authors contributed by discussions and reviewing the paper. Mohammad helped with modeling the network resources and developing the simulator.

This paper addresses Subgoal 3.

7.1.5 Paper E

Exact and Approximate Supply Bound Function for Multiprocessor Periodic Resource Model: Unsynchronized Servers, Nima Khalilzad, Moris Behnam, Thomas Nolte, In ACM SIGBED Review special issue on the 5th International Workshop on Compositional Theory and Technology for Real-Time

Embedded Systems (CRTS'12), Volume 10, Number 3, October, 2013.
Abstract: *The Multi Processor Periodic Resource (MPR) model has been proposed for modeling compositional real-time systems which run on a shared multi processor hardware. In this paper we extend the MPR model such that the execution of virtual processors (servers) is not assumed to be synchronized i.e., the servers can have different phases. We believe that relaxing the server synchronization requirement provides greater deal of compatibility for implementing such a compositional method on various hardware platforms. We derive the resource supply bound function of the extended MPR model using an algorithm. Furthermore, we suggest an approach to calculate an approximate supply bound function with lower computational complexity for systems where calculating their supply bound function is computationally expensive.*

My contribution in this paper: I was the main driver of the work. The co-authors contributed by discussions and reviewing the paper.

This paper addresses Subgoal 4.

7.1.6 Paper F

On Component-Based Software Development for Multiprocessor Real-Time Systems, Nima Khalilzad, Moris Behnam and Thomas Nolte, In Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'15), August, 2015.
Abstract: *Component-based software development provides a modular approach to develop complex software systems. In the context of real-time systems, it is desirable to abstract the timing properties of software components using an interface for each component. The timing properties of the whole system, composed of multiple components, is studied using the component interfaces. In this paper we focus on periodic interface models. In the case of components developed for single processor platforms, for examining the system schedulability, the interfaces can be regarded as periodic tasks. Thus, making it possible to use the conventional schedulability analyses for the system level schedulability test. In the case of components developed for multiprocessors, since interfaces may have utilization larger than 100 % of a single processor, it is not possible to directly use the component interfaces for the system schedulability test. Therefore, the interfaces have to be decomposed before performing the system level schedulability test.*

In this paper, we target the special case of partitioned EDF for scheduling the components integrated on a multiprocessor. Therefore, the system level schedulability test is equivalent to finding a feasible allocation of component interfaces on the multiprocessor. We propose two algorithms for allocating the multiprocessor periodic interfaces. In addition, we propose an orthogonal approach for developing component-based real-time systems on multiprocessors in which components with utilization more than 100 % of a single processor are divided into smaller subcomponents before abstracting their interfaces. We show, through extensive evaluations, that our alternative approach significantly reduces the interface overhead.

My contribution in this paper: I was the main driver of the work. The co-authors contributed by discussions and reviewing the paper.

This paper addresses Subgoal 4.

References

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

[2] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 26–35, May 2001.

[3] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems, (ECRTS'08)*, pages 181–190, July 2008.

[4] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*, pages 249–258, December 2010.

[5] M. Di Natale and A.L. Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, 98(4):603–620, 2010.

[6] R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):956–965, 2009.

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

[8] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems (TECS'08)*, pages 30:1–30:39, April 2008.

[9] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium, (RTSS'07)*, pages 129–138, December 2007.

[10] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 10–398, December 2005.

[11] F. Zhang and A. Burns. Analysis of hierarchical EDF pre-emptive scheduling. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 423–434, December 2007.

[12] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, January 2009.

[13] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino. Resource management on multicore systems: The ACTORS approach. *Micro, IEEE*, 31(3):72–81, May-June 2011.

[14] C. C. Wust, L. Steffens, W. F. J. Verhaegh, R. J. Bril, and C. Hentschel. QoS control strategies for high-quality video processing. *Real-Time Systems*, pages 3–12, 2005.

[15] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 57–67, December 2004.

[16] A. Burmyakov, E. Bini, and E. Tovar. Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376, 2014.

[17] M. Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd international conference on Software engineering (ICSE'01)*, page 656. IEEE Computer Society, 2001.

[18] G. Dodig-Crnkovic. Scientific methods in computer science. *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130, 2002.

[19] K.M. Passino and S Yurkovich. *Fuzzy Control*. Addison-Wesley, 1998.

[20] K. J. Åstrom and R. M. Murray. *Feedback Systems, An Introduction for Scientists and Engineers*. Prentice University Press, 2012.

[21] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzen. How does control timing affect performance? analysis and simulation of timing using Jitterbug and TrueTime. *Control Systems, IEEE*, 23(3):16–30, June 2003.

[22] L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, July 2004.

[23] L. Abeni, C. Scordino, and L. Palopoli. Serving non real-time tasks in a reservation environment. In *Proceedings of the 9th Real-Time Linux Workshop*, November 2007.

[24] N. Gandhi, D.M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. MIMO control of an apache web server: modeling and controller design. In *Proceedings of the American Control Conference (ACC'02)*, volume 6, pages 4922–4927, 2002.

[25] Y. Diao, J. L Hellerstein, A. J Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *Proceedings of the 2004 American Control Conference (ACC'04)*, volume 3, pages 2045–2050, 2004.

[26] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23:85–126, 2002.

[27] A. Block, J. H. Anderson, and U. C. Devi. Task reweighting under global scheduling on multiprocessors. *Real-Time Systems*, 39(1-3):123–167, 2008.

[28] A. Block, J. H. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *Proceedings of the 11th IEEE International*

Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), pages 429–435, 2005.

[29] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 23–33, July 2008.

[30] J. Yao, X. Liu, X. Chen, X. Wang, and J. Li. Online decentralized adaptive optimal controller design of cpu utilization for distributed real-time embedded systems. In *Proceedings of the American Control Conference (ACC'10)*, pages 283–288, June 2010.

[31] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 59–70, December 2001.

[32] X. Wang, X. Fu, X. Liu, and Z. Gu. PAUC: Power-aware utilization control in distributed real-time systems. *IEEE Transactions on Industrial Informatics*, 6(3):302–315, Aug 2010.

[33] X. Chen, X. W. Chang, and X. Liu. SyRaFa: Synchronous rate and frequency adjustment for utilization control in distributed real-time embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1052–1061, May 2013.

[34] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. DEUCON: Decentralized end-to-end utilization control for distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):996–1009, July 2007.

[35] L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 70–77, December 1999.

[36] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'2)*, pages 71–80, December 2002.

[37] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 238–245, May 2004.

[38] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli. Adaptive real-time scheduling for legacy multimedia applications. *ACM Transactions on Embedded Computing Systems*, 11(4):86:1–86:23, January 2013.

[39] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, June 2004.

[40] Ethernet POWERLINK Standardisation Group. *EPSG Draft Standard 301 Ethernet POWERLINK Communication Profile Specification Version 1.2.0*, 2013.

[41] M. Ashjaei, M. Behnam, L. Almeida, and T. Nolte. Performance analysis of master-slave multi-hop switched ethernet networks. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13)*, June 2013.

[42] R. Santos, A Vieira, P. Pedreiras, A Oliveira, L. Almeida, R. Marau, and T. Nolte. Flexible, efficient and robust real-time communication with server-based Ethernet switching. In *Proceedings of the 8th IEEE International Workshop on Factory Communication Systems (WFCS'10)*, May 2010.

[43] R. Santos, M. Behnam, T. Nolte, P. Pedreiras, and L. Almeida. Multi-level hierarchical scheduling in ethernet switches. In *Proceedings of the of the International Conference on Embedded Software (EMSOFT'11)*, October 2011.

[44] J. Silvestre-Blanes, L. Almeida, R. Marau, and P. Pedreiras. Online QoS management for multimedia real-time transmission in industrial networks. *IEEE Transaction on Industrial Electronics*, 58(3), March 2011.

[45] IEEE 802.1Qat, draft standard for local and metropolitan area networks virtual bridged local area networks amendment 9: Stream reservation protocol (SRP).

[46] K. Lakshmanan and R. Rajkumar. Distributed resource kernels: OS support for end-to-end resource isolation. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, April 2008.

[47] A. Oliveira, A. Azim, S. Fischmeister, R. Marau, and L. Almeida. D-RES: Correct transitive distributed service sharing. In *Proceedings of the Work-in-Progress Session of the Conference on Emerging Technologies and Factory Automation (ETFA'14)*, September 2014.

[48] T. Cucinotta and L. Palopoli. QoS control for pipelines of tasks using multiple resources. *IEEE Transactions on Computers*, 59(3):416–430, March 2010.

[49] R. Rajkumar, C. Lee, J. Lehoczky, and Dan Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, December 1997.

[50] S. Ghosh, J. Hansen, R. Rajkumar, and J. Lehoczky. Integrated resource management and scheduling with multi-resource constraints. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, December 2004.

[51] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium (RTSS'90)*, pages 182 –190, December 1990.

[52] A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[53] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 256–267, December 1999.

[54] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the 6th IEEE Real Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, May 2000.

[55] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Proceedings of the 21st IEEE Real-time Systems Symposium (RTSS'00)*, pages 217–226, November 2000.

[56] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, May 2001.

[57] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pages 95–103, September 2004.

[58] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, July 2003.

[59] H. Leontyev and J. H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 191–200, July 2008.

[60] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium (RTSS'13)*, pages 1–10, December 2013.

[61] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the International Conference on Embedded Software (EMSOFT'14)*, pages 1–10, Oct 2014.

[62] E. Bini, G. Buttazzo, and M. Bertogna. The multi supply function abstraction for multiprocessors. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, (RTCSA'09)*, pages 294–302, August 2009.

[63] E. Bini, M. Bertogna, and S. Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, (RTSS'09)*, pages 437–446, December 2009.

[64] H. Zhu, S. Goddard, and M. B. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In *32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 239–248, December 2011.

II

Included Papers

Chapter 8

Paper A: Bandwidth Adaptation in Hierarchical Scheduling Using Fuzzy Controllers

Nima Khalilzad, Moris Behnam, Giacomo Spampinato and Thomas Nolte.
In Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), June, 2012.

Abstract

In our previous work, we have introduced an adaptive hierarchical scheduling framework as a solution for composing dynamic real-time systems, i.e., systems where the CPU demand of their tasks are subjected to unknown and potentially drastic changes during run-time. The framework uses the PI controller which periodically adapts the system to the current load situation. The conventional PI controller despite simplicity and low CPU overhead, provides acceptable performance. However, increasing the pressure on the controller, e.g., with an application consisting of multiple tasks with drastically oscillating execution times, degrades the performance of the PI controller.

Therefore, in this paper we modify the structure of our adaptive framework by replacing the PI controller with a fuzzy controller to achieve better performance. Furthermore, we conduct a simulation-based case study in which we compose dynamic tasks such as video decoder tasks with a set of static tasks into a single system, and we show that the new fuzzy controller outperforms our previous PI controller.

8.1 Introduction

The Hierarchical Scheduling Framework (HSF) is a component-based technique for scheduling complex real-time systems [1, 2]. Using such a framework, each component is allocated a portion of the CPU and, in turn, it guarantees that with this portion all its internal tasks will be scheduled such that their corresponding timing constraints are respected. The CPU portions are often specified by the component period and budget (interface parameters). The interface parameters can be calculated either based on the Worst Case Execution Time (WCET) of the tasks such as the method presented in [3] and kept fixed during run-time, or be initiated using such a method and then be adapted during run-time based on the current workload [4]. The dynamic resource allocation techniques are especially efficient when the system components are composed of dynamic tasks in which their execution times are changing significantly during run-time. For example, when a component consists of control tasks or video decoder tasks, since the execution time of such tasks are dynamic during run-time, fixed resource allocation techniques are not efficient and may result in underutilized systems and consequently the CPU resource will be wasted.

We have introduced the Adaptive Hierarchical Scheduling Framework (AHSF) [4] as a solution for composing dynamic components. In this hierarchical framework we assume a fixed period for each subsystem, however, each subsystem is equipped with a budget controller which adapts the subsystem budget based on two feedback loops. The feedback loops are controlling the number of deadline misses and the amount of idle time in the subsystem. In that work, we used the well known PI controller, designed based on an approximate system model.

In this paper we investigate a more advanced controller which does not require any pre knowledge about the system. We introduce the following contributions in this paper.

- (i) We investigate using a model-free fuzzy controller instead of conventional PI controllers and we define a new control variable based on the consumed budget after missing the deadlines instead of the number of deadline misses.

- (ii) We study the stability of our controller using Lyapunov's direct method which gives boundaries on the budget controller's gain values.

- (iii) We tune the designed fuzzy controller using a multi-criteria Genetic Algorithm (GA).

- (iv) We evaluate the performance of the proposed controller by conducting a case study and we compare the result of using the proposed controller against the PI controller approach presented in [4].

The remainder of this paper is organized as follows. Related work is presented in Section 8.2. Section 8.3 describes the structure of our AHSF. In Section 8.4 we give insight into our fuzzy budget controller. The stability study is presented in Section 8.5. We describe the controller tuning in Section 8.6. A simulation-based case study is presented in Section 8.7. The implementation complexity of the proposed approach is discussed in 8.8. Finally, we conclude the paper in Section 8.9.

8.2 Related work

The idea of closed-loop real-time scheduling emerged in late 90's [5] and since then there has been a growing attention in combining the real-time scheduling theory with the well-established control techniques. The deadline miss ratio is controlled in [6]. In [7], in addition to the deadline miss ratio, the CPU utilization is controlled as well. In a similar context, the CPU utilization is controlled by modifying task periods using a fuzzy controller in [8]. The idea is also applied to scheduling of control tasks where the quality-of-control is regulated using a feedback-feedforward method [9].

Resource reservation [10] and hierarchical scheduling [11, 12, 3, 13, 14, 15, 16] techniques have received increasingly more attention over the past two decades since they provide temporal isolation and consequently predictability in integrating different task models. Hierarchical scheduling is used in scheduling of soft real-time systems in [17, 18, 19]. All aforementioned methods assign fixed CPU portions to the subsystems and therefore it makes them inefficient when composing dynamic tasks.

Recently, there has been some work to enable the adaptability feature for resource reservation scheduling techniques by using feedback control techniques. In [20] Abeni *et al.* have introduced an adaptive Constant Bandwidth Server (CBS) as an extension to the CBS [21] in which the server budgets are adjusted during run-time. Their control variable is limited to existence of one task per server. Adaptive CPU resource management is presented in [22] where the hard CBS scheduling algorithm is used and the server budgets are adapted during run-time. Although in theory this approach can support existence of multiple tasks in a server, it is evaluated by using only one task per server. In our AHSF [4], we bring the feedback scheduling techniques in the context of

resource reservation scheduling in which the servers (components) consist of multiple tasks and they are scheduled using a real-time scheduler (hierarchical scheduling). Besides, we use periodic servers [23] instead of CBSs in our framework, however, our work can be extended to work under other type of servers such as CBSs with minor modifications.

8.3 The Adaptive Hierarchical Scheduling Framework

We consider a two-level Adaptive Hierarchical Scheduling Framework (AHSF) in which a system S consisting of N components, here denoted as subsystems $S_s \in S$, is executed on a single processor. In the AHSF, a global scheduler schedules subsystems, and a local scheduler in each subsystem is responsible for scheduling its corresponding internal tasks. We use the EDF scheduling algorithm in both global and local level in this paper, however the presented approach can be extended easily to include other scheduling algorithms, e.g., fixed priority scheduling. Figure 8.1 shows the architecture of our two-level AHSF. We use one fuzzy budget controller per subsystem to adapt its budget according to the CPU resource demand of its tasks. In addition, there is an overload controller which deals with overload situations, i.e., when the total resource request of the subsystems are more than the available CPU resource.

8.3.1 Subsystem model

Each subsystem S_s is represented by its temporal interface parameters (T_s, B_s, D_s, ζ_s) where T_s, B_s, D_s and ζ_s are subsystem period, budget, relative deadline and criticality respectively. The relative deadline of a subsystem is assumed to be equal to its corresponding subsystem period ($D_s = T_s$). Each subsystem S_s consists of a set of n_s tasks τ_s and a local scheduler. The criticality of a subsystem ζ_s, which shows how critical a subsystem is in comparison to other subsystems, is used only in overload situations. We assume that subsystems are sorted according to their criticality, in the order of decreasing criticality, and $\zeta_s = s$, i.e., S_1 has the highest criticality in the system while S_N has the lowest criticality.

We use periodic servers (subsystems) which works as follows. The required CPU portion is always allocated to the subsystems every predefined period, and in the case that there is no active task in the subsystem, it will idle its budget.

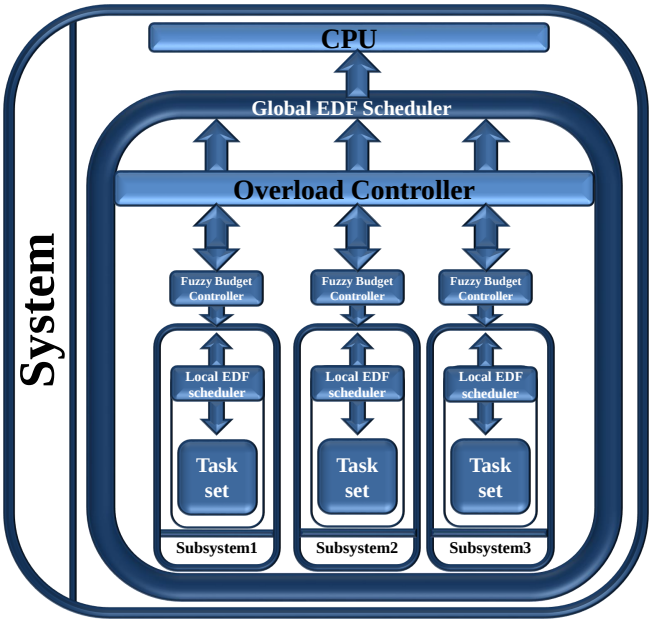


Figure 8.1: The Adaptive Hierarchical Scheduling Framework.

8.3.2 Task model

We assume the periodic soft real-time task model $\tau_{i,s}$ ($T_{i,s}$, $C_{i,s}$, $D_{i,s}$), where $T_{i,s}$, $C_{i,s}$ and $D_{i,s}$ are period, execution time and relative deadline of task i in subsystem S_s respectively. The relative deadline of a task is assumed to be equal to its corresponding task period ($D_{i,s} = T_{i,s}$). When a task misses its deadline it can continue its execution to the end.

8.3.3 The budget controller

We use two feedback loops in the structure of our budget controller. For the first loop controller computations, the amount of subsystem budget that is used by tasks after missing their deadlines to finish their executions, is monitored. Therefore, the error in this loop is defined as follows:

$$e_m(t) = \sum_{\tau_{i,s} \in \tau_s} \frac{\beta_{i,s}(t)}{T_s}, \tag{8.1}$$

where $\beta_{i,s}(t)$ is the control variable of the feedback loop which is the amount of subsystem budget of S_s used by task i after missing its deadline at sampling time t. We call this feedback loop the "m-loop" in the rest of the paper. Note that in [4] we measure the number of deadline misses in the "m-loop", however, we believe that our new control variable (Equation 8.1) provides the controller with more precise information about the state of the system, consequently the controller can take more effective actions in controlling the environment. It goes without saying that Equation 8.1 can only be used if tasks are allowed to continue executing after missing their deadlines.

In the second loop, the amount of idle time (unused budget) in each subsystem is monitored. Therefore, the error in this loop is defined as follows:

$$e_u(t) = \frac{\alpha_s(t)}{T_s}, \tag{8.2}$$

where $\alpha_s(t)$ is the control variable of the second feedback loop which is the amount of idle time in subsystem S_s measured at sampling time t. Similar to the m-loop, we call the second feedback loop the "u-loop" in the rest of the paper. Figure 8.2 illustrates the defined control variables in subsystem S_1 of an example system. There are two tasks in S_1 where $\tau_{2,1}$ misses its deadline at t_0, however, $\tau_{1,1}$ finishes its execution before its deadline. If we consider t_0, t_1 and t_2 as the sampling times, the value of the control variables in Figure 8.2 at the sampling times are as follows:

$$\begin{gathered}\alpha_1(t_0) = \alpha_1(t_1) = \alpha_1(t_2) = q_1, \\ \beta_{1,1}(t_0) = \beta_{1,1}(t_1) = \beta_{1,1}(t_2) = 0, \\ \beta_{2,1}(t_0) = 0, \beta_{2,1}(t_1) = x_1, \beta_{2,1}(t_2) = x_1 + x_2.\end{gathered}$$

The controllers of both loops are executed periodically and both error values are reset to zero at each control period. The controller periods are assumed to be proportional to the subsystem periods. In addition to the error value, the controller should be provided with the error difference value which is calculated as follows:

$$\Delta e(t) = e(t) - e(t-1), \tag{8.3}$$

where $e(t)$ is the error value at sampling time t. The error is either e_m or e_u depending on the control loop, therefore, there is an error difference variable per control loop: $\Delta e_m(t)$ corresponding to the m-loop and $\Delta e_u(t)$ corresponding to the u-loop. We provide the fuzzy controller with $e(t)$ and $\Delta e(t)$, and the controller computes a CPU portion $\Delta w(t)$ which affects the subsystem budget, i.e., it might increase the subsystem budget if there are deadline misses

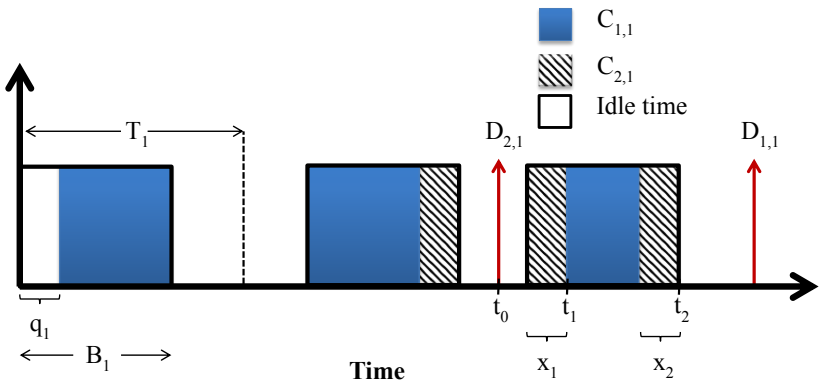


Figure 8.2: The control variables in S_1.

or decrease the subsystem budget if there is much unused budget. Hence, the new subsystem budget is calculated using the controller output and subsystem period:

$$B(t) = B(t-1) + T_s \Delta w(t). \tag{8.4}$$

The fuzzy budget controller is indeed an integral controller which adds the controller output to the current budget. The controller output is calculated as follows:

$$\Delta w(t) = K_f e(t), \tag{8.5}$$

where K_f is the proportional gain which is extracted from the fuzzy rule-base given $e(t)$ and $\Delta e(t)$. The rule-base and the fuzzy logic control is explained in detail in Section 8.4. The block diagram of the fuzzy budget controller is illustrated in Figure 8.3.

8.3.4 Integration of feedback loops

As mentioned earlier in this section, we use two feedback loops in our framework. Each loop calculates a budget change value $\Delta w(t)$, however, a mechanism should be provided for integrating these two values. We design a fuzzy multiplexer which combines the output of the control loops. The multiplexer simply looks at the system state, if the m-loop error is large, the output of the m-loop $\Delta w_m(t)$ will have the main impact on the final output, otherwise the final output is mainly based on the output of the u-loop $\Delta w_u(t)$. The block diagram of the fuzzy multiplexer is shown in Figure 8.3. There are two fuzzy sets

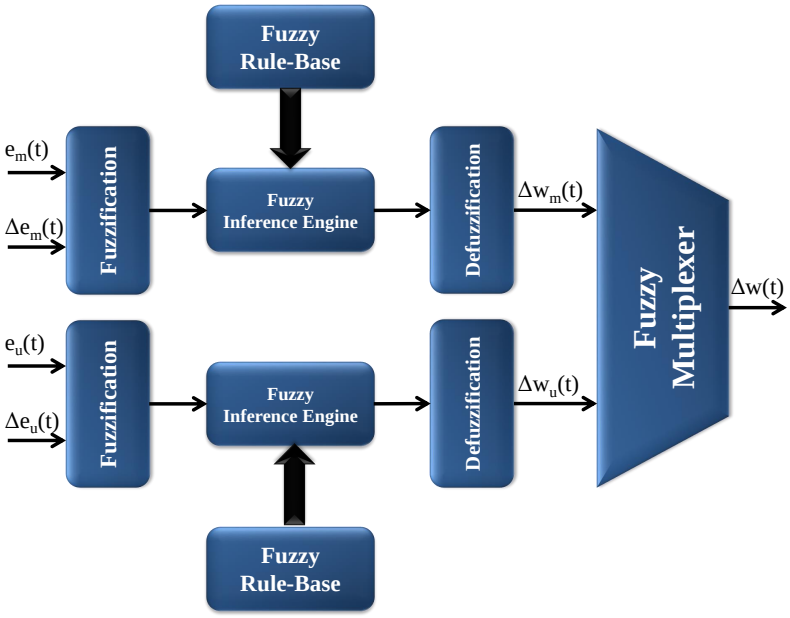


Figure 8.3: Block diagram of the fuzzy budget controller. $\Delta w_m(t)$: m-loop output, $\Delta w_u(t)$: u-loop output.

in the structure of the multiplexer: large and zero. Basically, the fuzzification and defuzzification steps in the integration phase are very similar to the steps presented in Section 8.4 for the budget controller, hence to avoid redundancy we do not explain them in this section. However, it is important to highlight that we use different fuzzy sets and a different rule-base than the budget controller. The rule-base presented in Table 8.1 is used in the fuzzy multiplexer.

8.3.5 The overload controller

The overload situation can happen when the total system utilization is more than 100 % and since the EDF scheduling algorithm is assumed, it is detected

	$e_m(t)$	
$e_u(t)$	Zero	Large
Zero	$\Delta w_u(t)$	$\Delta w_m(t)$
Large	$\Delta w_u(t)$	$\Delta w_m(t)$

Table 8.1: Fuzzy multiplexer rule-base.

by performing the following test:

$$\sum_{\forall S_s \in S} \frac{B_s}{T_s} > 1. \tag{8.6}$$

If the controller detects the overload situation, it redistributes the CPU resource among subsystems according to their criticality values ζ_s. It starts from the highest criticality subsystem S_1 and provides it with required budget. Thereafter, it moves to a lower criticality subsystem. The lower criticality subsystem can at most receive a budget value which corresponds to the CPU resource that is left after allocation to the highest criticality subsystems. This process continues until the lowest criticality subsystem receives CPU resources, which happens after all other subsystems have been assigned a new budget. In other words, when the controller finds out that there are not enough resources for all the subsystems, it tries to satisfy the high criticality subsystems by sacrificing the lower criticality subsystems. Note that in this approach, the low criticality subsystems might receive very small CPU portions or be completely shut down which is unavoidable due to the limited CPU resources.

Without having an overload controller, in the overload mode high priority subsystems will receive more resources than the low priority ones. Since we use the EDF scheduler at the global level, the shorter period subsystem are more likely to have higher priorities. However, the shorter period subsystems are often not the more important ones in the system since the periods are usually describing the temporal requirement and not the importance of the subsystems.

8.4 Fuzzy logic control

In this section, we explain how the control input $e(t)$ and $\Delta e(t)$ are mapped to the control output $\Delta w(t)$ using Fuzzy Logic Control (FLC) [24]. As illustrated in Figure 8.3, the first step in the FLC is fuzzification in which we map the crisp

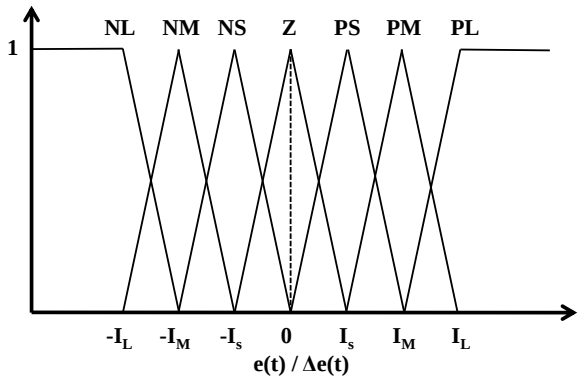


Figure 8.4: Membership function (NL:Negative Large, NM:Negative Medium, NS:Negative Small, Z:Zero, PS:Positive Small, PM: Positive Medium, PL:Positive Large, I_S: Ceiling of Small, I_M: Ceiling of Medium, I_L: Ceiling of Large).

input error to a linguistic value. We use the membership function presented in Figure 8.4 for the fuzzification purpose. Given an input value and using the membership function we get a set of truth values indicating how much the input belongs to each fuzzy set. According to the definition of the control variables the error value is always positive, therefore only the positive side of the membership function is needed for $e(t)$, however, $\Delta e(t)$ can be either negative or positive. We use the same membership function for both $e(t)$ and $\Delta e(t)$. In the next step we apply fuzzy rules to our fuzzy inputs to get a fuzzy control output. We use different rule-bases for each feedback loop which are presented in Table 8.2.

We use the minimum operator as "fuzzy and" to calculate the truth value of each fuzzy rule for the inference purpose. The final step in FLC is defuzzification in which a linguistic control action is mapped to a crisp value. Let $o(k)$ and $\mu(k)$ represent the rule consequent and the truth value of the k^{th} fuzzy rule respectively. Finally, according to the Sugeno's defuzzification model [25] the proportional gain is the weighted average of all rule outputs:

$$K_f = \frac{\sum o(k).\mu(k)}{\sum \mu(k)}. \tag{8.7}$$

	e(t)			
$\Delta e(t)$	Z	PS	PM	PL
NL	NM / PL	NM / PM	NS / PS	Z / Z
NM	NS/PM	NS / PS	Z / Z	PS / NS
NS	NS / PS	Z / Z	PS / NS	PM / NM
Z	NZ / PM	PS / Z	PM / NM	PL / NL
PS	PS / PM	PM / Z	PL / NL	PL / NL
PM	PM / PS	PL / Z	PL / NL	PL / NL
PL	PL / Z	PL / Z	PL / NL	PL / NL

Table 8.2: Fuzzy controller rule-base (m-loop / u-loop).

The rule consequent can be either zero gain K_z, small gain K_s, medium gain K_m or large gain K_l. We assume the zero gain is always 0. For example, assume $e(t) = \Delta e(t) = \frac{I_s}{2}$ (I_s is the ceiling of small fuzzy set). Then, for both $e(t)$ and $\Delta e(t)$, $\mu_Z = \mu_{PS} = 0.5$ where μ_Z and μ_{PS} are the truth value of the "Zero" and "Positive Small" fuzzy sets. Therefore the output of the m-loop is:

$$K_f = \frac{0 \times 0.5 + K_s \times 0.5 + K_s \times 0.5 + K_m \times 0.5}{0.5 + 0.5 + 0.5 + 0.5}, \tag{8.8}$$

and the output of the u-loop is:

$$K_f = \frac{K_m \times 0.5 + 0 \times 0.5 + K_m \times 0.5 + 0 \times 0.5}{0.5 + 0.5 + 0.5 + 0.5}. \tag{8.9}$$

8.5 Stability study

In control systems, stability is one of the important properties which should be studied after designing the controller. We use the direct method of the Lyapunov's stability analysis [26] to study the stability of our system. Assume that $y(t)$ is a function representing the distance of the current budget $B(t)$ from the equilibrium budget B_{eq} at sampling time t. The equilibrium budget is the budget that the subsystem neither experiences idle time nor its tasks miss their deadlines:

$$y(t) = B(t) - B_{eq}. \tag{8.10}$$

We define the Lyapunov function as follows:

$$V(y(t)) = y(t)^2. \tag{8.11}$$

Now, we should prove that the following conditions are satisfied.

1. $V(y(t)) = 0$, if the system is in its equilibrium state.
2. $V(y(t)) > 0$, if the system is in other states than its equilibrium.
3. $V(y(t+1)) - V(y(t)) = y^2(t+1) - y^2(t) < 0$.

From the definition of our Lyapunov function, condition 1 and 2 are obviously satisfied. Now we need to study the third condition. From the definition of $y(t)$ we have:

$$y^2(t+1) - y^2(t) = (B(t+1) - B_{eq})^2 - (B(t) - B_{eq})^2. \tag{8.12}$$

Expanding 8.12 and replacing $B(t+1)$ using Equation 8.4 and 8.5 we get:

$$y^2(t+1) - y^2(t) = K_f^2 T_s^2 e(t)^2 + 2K_f T_s e(t)(B(t) - B_{eq}). \tag{8.13}$$

Using Equation 8.11 we get:

$$y^2(t+1) - y^2(t) = K_f^2 T_s^2 e(t)^2 + 2K_f T_s e(t) y(t). \tag{8.14}$$

Therefore, according to the third condition the following inequality should be valid:

$$K_f^2 T_s^2 e(t)^2 + 2K_f T_s e(t) y(t) < 0, \tag{8.15}$$

which yields to the following two results:

1. $sign(K_f) = -sign(y(t))$ because all other variables are positive.
2. $|K_f e(t)| < |2\frac{y(t)}{T_s}|$.

Result 1 is used in design of the rule-base meaning that when the distance from the equlibrium budget is positive and consequently we have some idle time in the subsytem, the gain value K_f is negative, and if y is negative and there are some deadline misses in the subsytem K_f is positive.

Figure 8.5 shows a simple scenario that two tasks exist in a subsystem. In this case both $\tau_{1,1}$ and $\tau_{2,1}$ are missing their deadlines, and at any sampling time t: $y(t) = x_1 + x_2 + x_3$ meaning that if we add $y(t)$ to the current budget it is guaranteed that the tasks can finish their execution times before their deadlines. However, assuming that the controller period is proportional to the subsystem period (see Section 8.7), the controller can sample the subsystem at either t_0, t_1 or t_2 depending on the controller period. The control variables $\beta_{1,1}$ and $\beta_{2,1}$ at these sampling times are as follows.

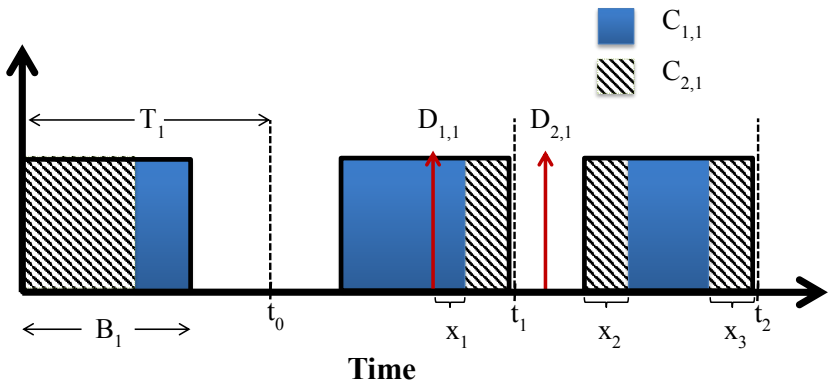


Figure 8.5: The control variables in subsystem S_1 of a sample system.

- At t_0: $\beta_{1,1} = \beta_{2,1} = 0$ and $\sum_{\tau_{i,1} \in \tau_1} \beta_{i,1}(t_0) < y(t_0)$.
- At t_1: $\beta_{1,1} = x_1, \beta_{2,1} = 0$ and $\sum_{\tau_{i,1} \in \tau_1} \beta_{i,1}(t_1) < y(t_1)$.
- At t_2: $\beta_{1,1} = x_1, \beta_{2,1} = x_2 + x_3$ and $\sum_{\tau_{i,1} \in \tau_1} \beta_{i,1}(t_2) = y(t_2)$.

Hence, we conclude that:

$$\sum_{\tau_{i,s} \in \tau_s} \beta_{i,s}(t) \leq y(t) \tag{8.16}$$

at any sampling time t. A similar reasoning can be done for the u-loop and its control variable $\alpha_s(t)$. Therefore, according to Equation 8.16 and the error definitions we derive:

$$e(t) \leq \frac{y(t)}{T_s}. \tag{8.17}$$

From result 2 and Equation 8.17 we derive that in order for the system to be stable, $|K_f|$ should be less than 2. The upper bound for $|K_f|$ can be considered higher than 2 depending on the controller frequency. The higher the controller frequency the greater the gain boundary, although from the stability point of view, it is always safe for the system to fulfill $|K_f| < 2$ condition.

The stability analysis gives some guidelines on how to design the rule-base and how to configure the controller. Although we do not get exact boundaries on the gain value K_f, the stability study provides us with approximate

boundaries which we use in tuning of the controller. In Section 8.6 we use evolutionary search to find an optimum configuration for our controller. Indeed, confining K_f, using the stability study in this section, limits the search space that we need to explore and speed-ups the convergence of the search.

8.6 Tuning the controller using evolutionary search

There are many parameters in our designed fuzzy controller that need to be tuned. For instance the fuzzy set intervals (both in the budget controller and in the multiplexer) are crucial parameters that should be carefully chosen. In addition, the gain values that are used in the output of the fuzzy rule-base need tuning. We use the GA to find the optimum parameters that maximize the performance of the controller.

There is a set of parameters associated with each control loop. For each loop the interval of small, medium and large fuzzy sets in addition to the three gain values: small gain K_s, medium gain K_m and large gain K_l should be tuned. To define the set intervals we only need to consider three values (see Figure 8.4): the ceiling of the "small" set I_s, the ceiling of the "medium" set I_m and the ceiling of the "large" set I_l. Note that we have different sets for each of the loops. We show the u-loop parameters using the super script u, and if the parameter belongs to m-loop we show it using the super script m. For example K_s^u is the small gain value corresponding to u-loop while K_s^m is the small gain value in m-loop.

The first step in using the GA is to design the structure of the so called chromosomes. We assume that each chromosome contains the information of all the parameters. However, we store the information indirectly to bias the GA. We assume that the fuzzy sets are harmonic meaning that:

$$I_l - I_m = I_m - I_s = I_s - 0 = h,$$

where h is the base interval size. This assumption limits the search space and helps the GA to converge faster, however, it might prevent it to find the absolute optimum solution. Recall from Section 8.3.4 that the fuzzy multiplexer has two fuzzy sets. The ceiling of the "large" set is assumed to be $1.5 \times h$ while the ceiling of the "zero" set is 0. In addition, assuming that $K_l > K_m > K_s$, we can write:

- $K_m = K_s + d_1$,

Figure 8.6: Structure of the chromosomes used in the GA.

- $K_l = K_m + d_2$,

where d_1 and d_2 are the difference of the medium gain with the small gain and the difference of the large gain with the medium gain respectively. Figure 8.6 illustrates the structure of the chromosomes showing that they contain all the tunable parameters.

Designing the mutation and crossover operators are the next stage in using the GA. We use the one point crossover operator on the randomly selected parents from the breeding pool. The mutation point is selected randomly as well. Thereafter, we add a random value between -0.1 and 0.1 to the selected variable. However, there are some boundaries on the variables, for instance none of the fields can be less than zero, therefore, if the result of the mutation is out of the valid region we immediately conduct another mutation.

The fitness function should be designed such that it directs the generations towards more optimum generations. We have two criteria in evaluating each chromosome. The lower the number of deadline missed tasks, the higher the efficiency. In addition, the amount of the idle time in the subsystems should be as low as possible. Therefore, we use a multi-objective GA approach called Vector Evaluated GA (VEGA) [27]. In this approach there are multiple fitness variables associated with each chromosome based on different criteria. When we want to select a parent, first we randomly choose the effective criterion, meaning that the chromosomes that are fit with respect to the selected criterion have a higher chance in being selected as a parent. Thereafter, we select the first parent and repeat the same procedure to select the second parent. After conducting extensive simulations we came to the conclusion that the solution converges faster when using three objectives which are based on (i) number of deadline misses (ii) amount of idle time (iii) combination of (i) and (ii).

Algorithm 1 shows the multi-objective GA used for tuning the parameters. It starts by generating random chromosomes and runs the simulation using their parameters. Thereafter it finds the fitness value of the chromosomes based on the three objectives. Afterward, the next generation is created based on the previous generation given that fitter chromosomes have more chance in being

Algorithm 1: Tuning the control parameters using VEGA

```
for i = 0 to i = populationSize do
    population(i) = random();
end for
for j = 0 to i = maxGeneration do
    for i = 0 to i = populationSize do
        simulation(population(i));
        fitnessIdle(i) =calculateFitness(idle);
        fitnessDl(i) =calculateFitness(dl);
        fitnessTotal(i) =calculateFitness(total);
    end for
    pool = population;
    for k = 0 to k = populationSize/2 do
        objective = randomInt(1,3);
        parent1 = selectParent(objective);
        objective = randomInt(1,3);
        parent2 = selectParent(objective);
        crossoverPoint = randomInt(1,8);
        children = crossover(parent1, parent2, crossoverPoint);
        population(2*i-1) = mutate(children(1));
        population(2*i) = mutate(children(2));
    end for
end for
```

selected as a parent. The functions and variables involved in the algorithm are:

- *"populationSize"* is the size of population.
- *"population"* is an array of chromosomes.
- *"random()"* is a function that returns a random variable between zero and one.
- *"maxGeneration"* is the number of generations that the GA tries to perform the optimization.
- *"simulation(population(i))"* given task sets, execution time change patterns and the variables in the chromosome of population i performs the simulation.

- *"fitnessIdle(i)"*, *"fitnessDl(i)"* and "*fitnessTotal(i)*" store the fitness value of the population i based on the idle time, deadline miss and combination of both objectives.
- "*calculateFitness()*" function calculates the fitness value based on the input objective.
- "*idle*", "*dl*" and "*total*" are the thee objectives that we use in the GA.
- *"pool"* stores the current generation which is used to generate the next generation.
- "*randomInt(a, b)*" returns an integer random variable between a and b.
- *"objective"* stores the randomly selected objective.
- *"parent1"* and *"parent2"* are the selected parents for the crossover.
- *"selectParent(objective)"* randomly selects an individual from *"pool"* given the input objective meaning than the individuals that are fit with respect to the input objective have a higher chance to be selected.
- "*crossover()*" does the crossover operation on its input chromosomes and given the crossover point. It returns two children.
- *"crossoverPoint"* stores the gene number that the crossover should be performed on it.
- "*chidden*" is an array which stores the output of the crossover operators.
- "*mutate()*" chooses a random mutation point and performs the mutation.

8.7 Evaluation

In this section we design a case study using real task execution times measured from running a video decoder task on a sequence of TV frames. We use the same data as the authors of [28], which they used for evaluating their work. We have used the TrueTime [29] simulation tool for our evaluation purposes. The TrueTime kernel has been modified such that our AHSF has been implemented.

In this study, we assume a system consisting of three subsystems where each of them is composed of three tasks. Subsystem S_1 is composed of a decoder task with two other tasks having fixed execution times during run-time.

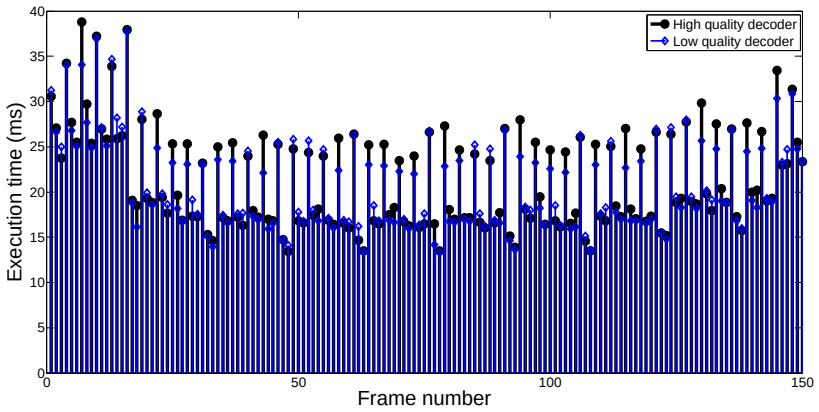


Figure 8.7: Execution times of the decoder tasks in two quality levels decoding the first 150 frames of the TV stream.

Subsystem S_2 contains two fixed execution time tasks and a dynamic task. The dynamic task operates in two modes: low and high, where its execution time is doubled when it is in the high mode. We assume that this task changes its mode each 2 seconds. The reason that we add this task to our sample system is to increase the pressure on the budget controllers, hence their difference with respect to the control performance can be revealed and compared easily. Subsystem S_3 has the same type of tasks as S_1. The only difference between them is that the decoder task in S_1 decodes the frames in a higher quality level than the decoder task in S_3. The execution times of the decoder tasks decoding various frames are shown in Figure 8.7. The figure illustrates that the execution time is fluctuating depending on the content of the frames. Table 8.3 shows the task specifications in detail. The executions times reported for the dynamic tasks are the mean execution times.

Three types of budget allocation techniques are studied for scheduling the described system.

- First of all we allocate a fixed budget for each subsystem using the analytical approach presented in [3]. The execution time of the dynamic tasks are assumed to be equal to the mean value of their execution times in the budget calculation analysis.

- Secondly, we use the PI controller [4] for dynamically allocating the

$\tau_{i,s}/S_s$	$T_{i,s}/T_s$	$C_{i,s}/C_s$
$\tau_{1,1}$	40	2.4
$\tau_{2,1}$	30	5
$\tau_{3,1}$	30	4
S_1	10	2.5
$\tau_{1,2}$	60	8
$\tau_{2,2}$	50	5
$\tau_{3,2}$	90	4
S_2	15	5
$\tau_{1,3}$	40	2.3
$\tau_{2,3}$	70	7
$\tau_{3,3}$	80	6
S_3	20	8.5

Table 8.3: Specifications of tasks and subsystems used in the case study.

budgets.

- Finally, we use the fuzzy controller introduced in this paper which allocates the budgets during run-time.

The control frequency is an important parameter which should be taken into account when designing an adaptive scheduler. Although frequently sampling and manipulating the environment might give a good control performance, due to the control overhead on the CPU, it is desirable to invoke the controller in a lower frequency. A reasonable approach for setting the control period is to set it proportional to its subsystem period. Therefore, in the PI controller, the controller period of each subsystem is set to be equal to the corresponding subsystem period times two. However, since the fuzzy controller has more overhead than the PI controller, we assign longer control periods which are equal to the subsystem periods times six.

The fuzzy controller is tuned using the first 10 seconds of the simulation with the help of the GA presented in Section 8.6. We started with 150 individuals and stopped the GA after 50 generations. Then we picked the best individual from the 50'th generation which was fit with respect to the third objective (see Section 8.6). Afterwards, we ran the simulations for 200 seconds and compared the performance of the budget allocation techniques with each other.

Figure 8.8 illustrates the budget value of the three subsystems during run-

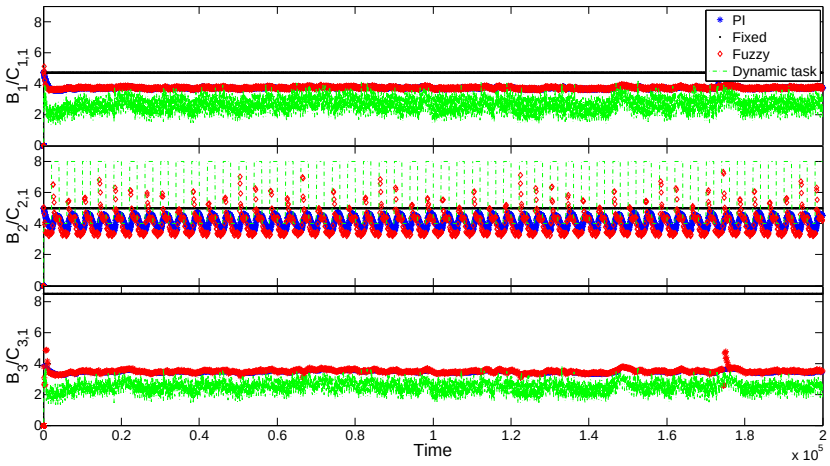


Figure 8.8: Budget adaptation in the case study.

time, which are allocated using the three techniques. The dynamic tasks are also shown in the figure. The y-axes of the figure shows the value of subsystem budgets and the value of execution times of the dynamic tasks. Since the execution times of the dynamic tasks are changing, the subsystem budgets (in the case that they are adaptive) are changing as well. Note that S_1 has the highest criticality in the system and $\zeta_1 > \zeta_2 > \zeta_3$. The figure clearly shows that assigning fixed budgets using the analysis is very pessimistic and it results in resources being wasted, although we used the mean value of the dynamic tasks in the analysis. Therefore, if we use the maximum execution times in the analysis, it will give even more pessimistic budgets.

Table 8.4 summarizes the performance metrics that we are interested in after running the simulation using the three techniques. The table shows that a fixed budget allocation is not efficient at all since the system is overloaded, and between the PI controller and the fuzzy controller, despite the fact that the fuzzy controller has a longer control period, the fuzzy controller is more successful in reducing task deadline misses. The main difference between the performance of the two controllers is in deadline miss ratio of S_2, where the fuzzy controller managed to reduce the deadline miss ratio with an additional approximately 6 % compared to the PI controller.

In the second simulation we modify task $\tau_{1,2}$ in the previous sample sys-

Performance metric	Technique	S_1	S_2	S_3
# DL misses	Fixed	7566	3027	2
	PI	44	1004	77
	Fuzzy	76	335	64
DL miss ratio	Fixed	29.68 %	24.08 %	0.03 %
	PI	0.24 %	9.51 %	0.97 %
	Fuzzy	0.41 %	3.39 %	0.81 %
idle time	Fixed	26	2661	44175
	PI	2149	6003	2393
	Fuzzy	2352	4304	3037

Table 8.4: Comparison of the performance of the three budget allocation approaches. DL: deadline. idle time is in millisecond.

Controller	S_1	S_2	S_3
PI	0.24 %	17.70 %	1.33 %
Fuzzy	0.41 %	3.10 %	7.58 %

Table 8.5: Deadline miss ratio in the second simulation.

tem such that the execution time of the dynamic task is tripled in the high mode which imposes even more pressure on the budget controllers. However, this time we do not tune our fuzzy controller and we use the previously tuned controller to see how well it works in a scenario similar to the one that it is tuned to work in. The deadline miss ratio for the three subsystems is presented in Table 8.5. The table shows that the fuzzy controller is able to reduce the deadline miss ratio with an additional approximately 14 % compared to the performance of the PI controller in S_2, however the PI controller is 6 % better in S_3. Since the system is overloaded one subsystem should be sacrificed, and since the PI controller is slower in adaptation, it sacrifices the higher criticality subsystem S_2 and serves the lower criticality one S_3. Therefore in total the fuzzy controller successfully schedules 8 % more tasks, and taking the criticalities into account, the value of avoiding deadline misses in S_2 is higher than S_3.

A potential drawback with the fuzzy budget controller could be its tuning. Since the tuning process is application specific, a tuned controller for a specific application could be less efficient for other applications. However, the simulation results suggest that the fuzzy controller works fine in relatively similar

dynamic scenarios to the scenario that it is tuned for. In general, the closer the tuning scenario is to the test scenario, the better the performance.

8.8 Implementation complexity

In this section we explain the implementation complexity of the different stages in the fuzzy budget controller.

The input values $e(t)$ and $\Delta e(t)$ can at most belong to two neighbor fuzzy sets. The corresponding fuzzy sets can be found by a couple of "if" statements. Thereafter, their membership value should be calculated. Given the set boundaries calculating μ requires a sum operation together with a multiplication. Afterwards, the "fuzzy and" operator should be performed on the two membership value. The "fuzzy and" operator consist of an "if" statement. At most four "fuzzy and" operations should be performed. Finally Equation 8.7 should be executed for the defuzzification purpose. These stages are done for both control loops.

The fuzzy multiplexer has the same stages as the budget controller, however, since the number of fuzzy sets are fewer, finding the corresponding fuzzy set for the input value requires less computations.

The overload controller, which only gets activated in the overload situation, consist of sum and assignment operations. The controller loops through the subsystems, assigns a new budget to them if necessary and updates the available CPU resource. The number of iterations in the loop is equal to the number of subsystems in the system.

As a conclusion, given that the fuzzy controller requires running in lower frequency than the PI controller, it does not add significant overhead when it is implement.

8.9 Conclusion

In this paper, we studied the use of a more advanced controller (fuzzy controller) than the conventional PI controller in our adaptive hierarchical scheduling framework for controlling the subsystem budgets during run-time. Thereafter, we showed how the fuzzy budget controller is tuned using a multi-objective genetic algorithm. To study the performance of the new budget controller we conducted a case study using video decoder tasks where the fuzzy controller outperformed the PI controller.

We intend to extend our work to the context of multi-core systems where an adaptive hierarchical framework runs on a multi-core CPU. Furthermore, since the control overhead is one of the main issues in our adaptive framework, we want to deeply study this issue by implementing the controllers in the Linux kernel.

References

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

[2] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 26–35, May 2001.

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

[4] N. Khalilzad, T. Nolte, M. Behnam, and M. Åsberg. Towards adaptive hierarchical scheduling of real-time systems. In *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11)*, pages 1–8, September 2011.

[5] J. A. Stankovic, C. Lu, S.H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS'99)*, pages 11–20, June 1999.

[6] C. Lu, J. A. Stankovic, G. Tao, and S.H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 56–67, December 1999.

[7] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23:85–126, 2002.

[8] C. Basaran, M. H. Suzer, K.-D. Kang, and X. Liu. Robust fuzzy CPU utilization control for dynamic workloads. *Journal of Systems and Software*, pages 1192–1204, July 2010.

[9] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzen. Feedbackfeedforward scheduling of control tasks. *Real-Time Systems*, pages 25–53, 2002.

[10] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[11] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, May 2001.

[12] F. Zhang and A. Burns. Analysis of hierarchical EDF pre-emptive scheduling. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 423–434, December 2007.

[13] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 256–267, December 1999.

[14] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pages 95–103, September 2004.

[15] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the 6th IEEE Real Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, May 2000.

[16] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Proceedings of the 21st IEEE Real-time Systems Symposium (RTSS'00)*, pages 217–226, November 2000.

[17] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 3–14, December 2001.

[18] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on OS Design and Implementation (OSDI'96)*, 1996.

[19] H. Leontyev and J. H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 191–200, July 2008.

[20] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'2)*, pages 71–80, December 2002.

[21] L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, July 2004.

[22] V. Romero Segovia. Adaptive CPU resource management for multicore platforms. Licentiate thesis, September 2011.

[23] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritised preemptive scheduling. In *Proceedings of the Real-Time Systems Symposium. (RTSS'86)*, pages 181–191, July 1986.

[24] K.M. Passino and S Yurkovich. *Fuzzy Control*. Addison-Wesley, 1998.

[25] M. Sugeno. *Industrial applications of fuzzy control*. Elsevier Science Pub. Co., 1985.

[26] M. Gopal I. J. Nagrath. *Control systems engineering*. Anshan, 2008.

[27] A. Konak, D. W. Coit, and A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91:992–1007, 2006.

[28] C. C. Wust, L. Steffens, W. F. J. Verhaegh, R. J. Bril, and C. Hentschel. QoS control strategies for high-quality video processing. *Real-Time Systems*, pages 3–12, 2005.

[29] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzen. How does control timing affect performance? analysis and simulation of timing using Jitterbug and TrueTime. *Control Systems, IEEE*, 23(3):16–30, June 2003.

Chapter 9

Paper B: An Adaptive Scheduling Framework for Component-Based Real-Time Systems

Nima Khalilzad, Moris Behnam and Thomas Nolte.
Under revision in Journal of Systems and Software (JSS), Special Issue on Computers, Software, and Applications - Software Engineering in COMPSAC.

Abstract

Processor partitioning techniques have been widely used for scheduling component-based hard real-time systems. Due to the safety critical nature of hard real-time systems, conservative partition sizes are often reserved for the components. A considerable capacity of the processor is wasted using such conservative techniques. When designing a component-based soft real-time system, however, conservative partitioning is unacceptable, because occasional timing violations can be tolerated by such systems.

In this paper, we present a multi-level adaptive hierarchical scheduling framework for scheduling component-based real-time systems. In our framework, for efficiently utilizing the processor capacity, we adapt the partition sizes of soft real-time components based on their actual needs at run-time. The adaptation is based on on-line monitoring of the processor demand of the components. We have implemented our framework in the Linux kernel. We present the implementation details of our framework. Finally, we report our evaluation results.

9.1 Introduction

Component-based software development is a modular approach for designing and developing complex software systems. When developing real-time systems using the component-based development paradigm, the timing correctness as well as the functional correctness of the components have to be carefully studied. To this end, an enormous number of works have focused on developing scheduling mechanisms for component-based real-time systems (e.g. [1, 2]). Similar to these studies, we consider a run-time component model in which a component is composed of a consistent set of real-time tasks and/or subcomponents. A real-time task is a sequential program that performs a specific functionality. Processor partitioning and reservation-based scheduling are well established techniques for scheduling component-based systems. In these methods, the processor time is divided into a number of partitions (also referred as reservations). Each component, then, is assigned to one processor partition [3]. The sizes of the processor partitions are determined based on the Worst Case Execution Time (WCET) of their inner tasks. The processor partitioning method allows developers to study the timing behavior of the components in isolation. The composition mechanism guarantees that the timing properties will be preserved after the composition. Therefore, the correctness of the entire system is inferred from the correctness of the individual components. This compositional timing study is especially useful in open systems in which components are added or removed during the system's life time [1].

While the problem of composing hard real-time systems is well studied, composition of soft real-time systems together or with hard real-time systems has not received as much attention. The processor partitioning for soft real-time components has to be different from that of the hard real-time components. This is because, the conservative processor overallocations cannot be justified due to the softness of the timing requirements. In other words, since the soft real-time systems can tolerate occasional timing violations, we do not need to over-allocate the processor. Furthermore, the execution time of some real-time tasks may be unknown a priori to run-time while being highly dynamic. For instance, a video decoder task where its execution time is depending on the content of the input video may experience significantly large variations in its execution time depending on which video that is being played. Therefore, new partitioning techniques are required to address soft real-time components with unknown and dynamic workloads.

When it comes to the structure of the component-based systems, the system may contain multiple levels of hierarchy. For instance consider a system in

which multiple Virtual Machines (VM) sharing a processor where in order to guarantee the timing requirements of the VMs, they are allocated to processor partitions. In addition, inside each VM there may be a number of independent real-time components running in parallel. In order to isolate the timing behavior of the components, the processor share of the VMs might also be partitioned, and each component may be assigned to a processor partition of its parent VM. Similarly, the components may contain subcomponents which can result in further processor partitioning.

In this paper we propose a multi-level hierarchical scheduling scheme for composing hard and soft real-time components. We allocate static processor partitions to the hard real-time systems, and we adapt the sizes of the partitions allocated to the soft real-time components. More specifically we present the following contributions in this paper:

1. We propose an adaptation mechanism for adjusting the processor partition sizes during run-time. Through on-line monitoring, we derive the instantaneous processor demands of the components inside the processor partitions, and based on the actual demands we adjust the partition sizes.

2. We show that the timing isolation between the soft real-time components and the hard real-time components are preserved using our adaptation technique. In other words, we show that adapting the soft real-time partition sizes does not affect the timing behavior of the hard real-time partitions.

3. We present our implementation method in which we used a Linux kernel loadable module that implements our proposed adaptive framework in the Linux kernel.

4. We evaluate the performance of our proposed framework. In the evaluation, we used video decoder tasks which exhibit wide processor demand variations.

5. We present an overhead evaluation of the scheduler and the processor partition adapter.

The first two contributions are presented in our previous work [4]. In this paper we present the implementation details of our framework (third contribution). We also provide a more extensive performance and overhead evaluations than the our previous work (forth and fifth contributions).

The rest of the paper is organized as follows. In Section 9.2 we review the related work. Our adaptive framework is presented in Section 9.3. The details of the Linux implementation is presented in Section 9.4. We present performance evaluations in Section 9.5. We also report the additional overhead imposed by our framework in this section. Finally, we conclude the paper in Section 9.6.

9.2 Related work

In this section we review three lines of work related to hierarchical scheduling, feedback scheduling and implementation of hierarchical scheduling.

9.2.1 Hierarchical scheduling

Reservation-based scheduling has been developed for scheduling component-based real-time systems. Hierarchical scheduling through CPU reservation emerged in 90's [1], where the idea was to partition the CPU into a number of partitions (CPU reservations) and assign a partition to a component. Processor partitioning provides temporal isolation to components, consequently, the timing behavior of each component can be studied independently. The partitioning also paves the way for adding and removing components without jeopardizing the timing requirements of the other components in the system.

Researchers have proposed several modeling techniques for capturing the timing behavior of the processor partitions (e.g., [2]). Using such models, then, the schedulability of the components can be studied. For instance, Zhang and Burns [5] presented a schedulability analysis for hierarchical scheduling with fixed priority at the global level and local EDF. In [3] Shin and Lee presented the periodic resource model in which the inner components of each partition is guaranteed to receive B units of the CPU time each P time units. We use the periodic resource model in our adaptive scheme. When using the CPU partitioning approach, the common assumption is that the CPU demand of the real-time tasks (WCET) are known a priori. Given this demand, a sufficient partition size can be calculated such that the timing requirement of real-time tasks are not violated. However, we assume that for soft real-time tasks the task demands and therefore the sufficient partition sizes are unknown. Thus, our framework is different from all conventional static hierarchical frameworks in the sense that the partition sizes are (i) not predefined at design time, but (ii) they are dynamic at run-time.

9.2.2 Feedback scheduling

Since Stankovic *et al.* introduced the idea of closed-loop real-time scheduling [6], there has been a growing interest in adopting feedback control techniques in the context of real-time scheduling. In the following we review a few of early publications in this area. The actual execution times of tasks may be different than their estimations. Therefore, a system can benefit from monitoring real execution times and performing admission control based on real information rather than estimations. To this end, Lu *et al.* [7] presented the Feedback Control EDF (FC-EDF) in which there is a PID controller on top of the EDF scheduler. The controller monitors the deadline miss ratio and based on that, it adjusts the tasks requested CPU utilization values. This adjustment affects the available CPU utilization and therefore the admission control. In the context of controlling physical plants, it is desirable to increase the quality-of-control. Cervin *et al.* presented a feedback-feedforward scheme in which by adapting the sampling period of tasks, the quality-of-control is regulated [8].

Multimedia tasks are in particular interesting from the timing behavior perspective, because they demonstrate highly dynamic execution times. In order to ensure that the multimedia tasks receive enough CPU bandwidth, resource partitioning is used for scheduling multimedia tasks [9]. Due to the dynamic nature of multimedia tasks, adaptive reservation techniques are devised for dealing with this kind of tasks. Abeni *et al.* proposed using a PI controller on top of Constant Bandwidth Servers (CBS) which adapts the CBS bandwidth such that it tracks the current workload of tasks attached to the CBS. Utilizing adaptive CBS (with a new control scheme), Palopoli *et al.* presented the AQuoSA framework [10]. In the context of the ACTORS project [11], a cascade controller is used on top of the hard CBS scheduling algorithm for adapting the CBS bandwidth. Our framework is different from both AQuoSA and ACTORS in the following. We consider component-based systems in which one component may be composed of multiple tasks/subcomponents, whereas the above frameworks do not support subcomponents and multiple tasks.

Finally, we studied the problem of budget adaptation using PI controllers [12] and Fuzzy controllers [13]. In our aforementioned previous work we have investigated two-level hierarchical scheduling, however, in this paper we present a new budget adaptation scheme which supports any arbitrary level of hierarchy. The new scheme also takes the existence of hard real-time systems into account and we show that adapting the soft real-time components does not harm the hard real-time components. Moreover, in the above papers we performed simulation-based evaluations whereas in this

paper we evaluate our framework by implementing our multi-level adaptive hierarchical scheduling framework in the Linux kernel and by running real tasks.

9.2.3 Implementation

Implementation of real-time schedulers have been widely studied (e.g., RTLinux [14] and RTAI [15]). However, in this paper we only review a part of the literature that focuses either on hierarchical scheduling or on adaptive scheduling which are closely related to our Linux implementation contribution.

Hierarchical scheduling has been implemented in many different platforms. Behnam *et al.* [16] implemented hierarchical scheduling on top of the VxWorks operating system. Inam *et al.* implemented hierarchical scheduling on top of the FreeRTOS operating system [17]. ExSched [18] is a platform independent real-time scheduler which has a hierarchical scheduling plug-in. Hierarchical scheduling is also implemented in μC/OS-II [19]. The above implementations do not support run-time adaptations.

Hierarchical scheduling is also used for virtualization purposes. Recursive virtual machines are proposed in [20] where each virtual machine can directly access the microkernel. Yang *et al.* presented a two-level hierarchical scheduler using the L4/Fiasco hypervisor [21]. In the Xen hypervisor, Lee *et al.* developed a virtualization platform [22]. A virtual CPU scheduling framework in the Quest operating system is developed by Danish *et al.* [23]. In [24], the CPU reservations are used for scheduling virtual machines. The VirtualBox and the KVM hypervisor are scheduled using CPU reservation techniques in [25]. The difference of our work with this line of work is that we support on-line adaptability and multilevel hierarchies of components.

HLS [26] is a multi-level hierarchal scheduler implemented in Windows 2000 which targets composing soft real-time systems. In [27], Parmer and West presented a hierarchical scheme for managing CPU, memory and I/O. These frameworks are not adaptive in the sense that the resource demands are not monitored and hence the resource reservations are fixed during run-time.

9.3 Framework

In this section, we present the background information required for understanding our framework as well as the structure of our adaptive framework and our

adaptation mechanism.

9.3.1 Component model

A component ($\mathfrak{C}_j$) consists of m_j real-time tasks (τ_i^j) and n_j subcomponents ($\mathfrak{C}_\kappa^j$), where i and κ are the index of task and subcomponent respectively, and j indicates their parent index. We use the term inner-elements of $\mathfrak{C}_j$, when referring to both tasks and subcomponents of $\mathfrak{C}_j$. We use idling periodic servers for providing processor time to the components. Since each component is assigned to a single periodic server, we use the terms component and server interchangeably in the rest of the paper. A periodic server S_j, assigned to $\mathfrak{C}_j$, receives B_j units of the processor time every P_j time units. When the inner-elements of $\mathfrak{C}_j$ are not active while S_j is active, the server budget is idled. The relative importance of $\mathfrak{C}_j$ with respect to the other components under the same parent is denoted by ζ_j. This value is used when the system is overloaded. In such a condition, some components have to suffer from low processor time provisioning. We use the importance value for making decisions on which component that has to be sacrificed. We consider open systems in which tasks/subcomponents may join/leave the component during run-time. Components can either be Hard Real-Time (HRT) or Soft Real-Time (SRT). The HRT components receive a fixed amount of the processor time during their life-time. In other words, the server assigned to a HRT component has a fixed budget and a fixed period. The amount of required budget for the HRT components is calculated using the analysis provided in [3]. The SRT components, however, may receive different budgets from period to period. The amount of processor time assigned to the SRT components depends on the current processor needs of their inner-elements.

9.3.2 Task model

In this paper we assume a periodic task model. A task τ_i^j is released every T_i^j time units. The priority, deadline and the worst-case execution time of tasks are denoted using pr_i^j, D_i^j and C_i^j, respectively. One instance of the task execution is called a "job". We assume that only the HRT tasks have a known worst-case execution time, while the execution time of the SRT tasks is unknown a priori. Therefore, we provide sufficient processor time to the HRT tasks. The SRT tasks, however, may occasionally miss their deadlines due to insufficient processor provisioning. In such a case, the remaining execution of the task will be scheduled for execution after the deadline. Note that execution until

completion is necessary in a group of real-time tasks such as video decoders. In such tasks, jobs often rely on the result of the previous job's computations.

9.3.3 System model

We assume a system comprised of $n+m$ components of which m components are HRT and the rest are SRT. Components may be composed of subcomponents hierarchically. The scheduling is also performed hierarchically. At the global level the CPU time is distributed among the global-level components. Each component in turn is responsible to serve the subcomponents/tasks that are located underneath them in the hierarchy. We provide fixed processor portions to the HRT components. The processor portion of the SRT components, on the other hand, are adjusted using *budget controllers* based on the current demand of the components. Figure 9.1 illustrates our system model. The set of SRT, HRT and all components that are located at the first level of the hierarchy are denoted using $\mathfrak{C}^{all}$, $\mathfrak{C}^{srt}$ and $\mathfrak{C}^{hrt}$ respectively.

Considering the system hierarchy, let us present two definitions that are used in the later sections of the paper for explaining the implementation of the scheduler.

Definition 1. *S_i^j is an **ancestor** of S_κ^l if either $i = l$ or by upward traversing the parent of S_l we reach S_i^j. For instance, S_1^0 is an ancestor of S_5^3 in Figure 9.2.*

Definition 2. *S_i^j **outranks** S_κ^l if and only if an ancestor of S_l is S_j. For instance, S_2^0 outranks S_3^1 in Figure 9.2 because S_0 is an ancestor of S_3^1.*

9.3.4 Adaptation model

In the following we explain our adaptation model that is used for adjusting the processor portions provisioned to the SRT components. We assume that the component periods are selected by the designers. For instance, in the context of video decoding, the period may be selected based on the number of frames required to be delivered per each second. We, then, adapt the budgets at run-time. Therefore, the component budgets are time-variant. The budgets are adapted periodically every P_j^{Ctrl} time units. We assume that the adaptation periods are proportional to the period of the servers of the components: $P_j^{Ctrl} = \mu \times P_j$, where $\mu > 1$ is an integer number. Each activation of the budget adaptation is referred as a "control event", and it is represented using k, where $k \geq 1$. The first control event happens at time $t = P_j^{Ctrl}$. The result of each control event is a new budget for the component. Therefore, the budget is a function of the

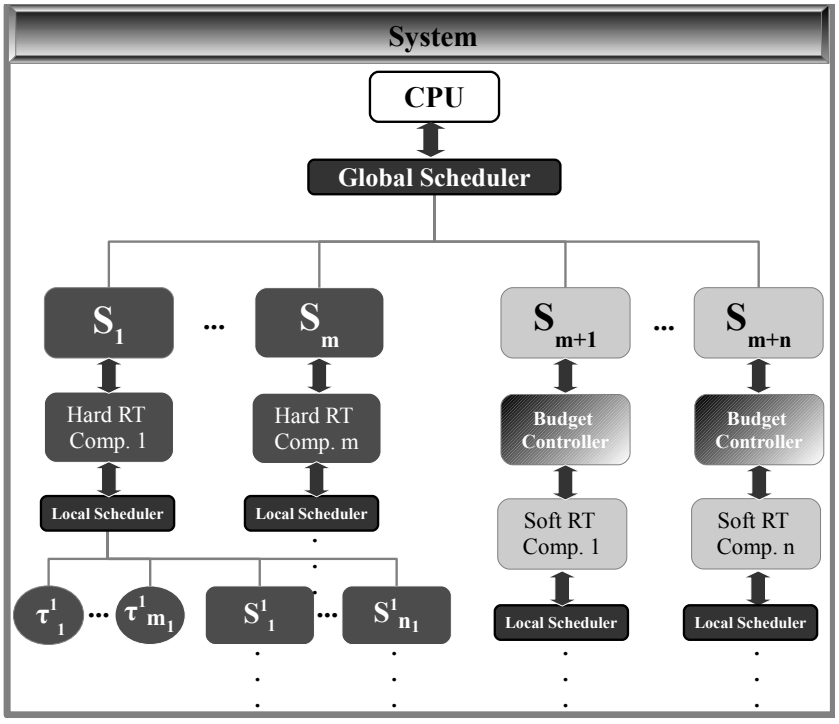


Figure 9.1: Visualization of the system model.

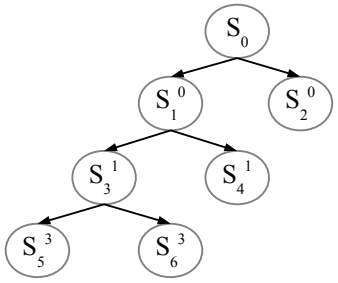


Figure 9.2: Example tree structure (S_0 represents the root scheduler).

control event $B_j(k)$. The component bandwidth ($\alpha_j(k)$) is defined as follows: $\alpha_j(k) = B_j(k)/P_j$. Let us define $\Psi_j^{k_j}$ as the k^{th} "control period" of $\mathfrak{C}_j$ which

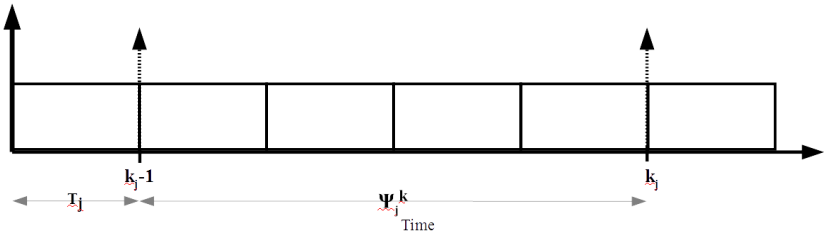


Figure 9.3: Visualization of the control period Ψ_j^k and the control event k_j.

represents the following time window:

$$\Psi_j^{k_j} = \Big((k_j - 1)P_j^{Ctrl}, k_j P_j^{Ctrl}\Big].$$

The first control period of server S_j (Ψ_j^1) is the following time interval: $\Psi_j^{k_j} = (0, P_j^{Ctrl}]$. Figure 9.3 visualizes the control event and the control period concepts. In the figure, the controller period is assumed to be four times the server period, i.e. $\mu = 4$.

Although components may have different control events as well as control periods, in the rest of the paper we drop index j when referring to k_j and Ψ_j^k for keeping the equations simple. At each control event, our goal is to assign a sufficient budget to the component under adaptation. Let us assume that we know how much budget is sufficient for serving the component's inner-elements in the next control period. We use $b_j(k)$ to denote the amount of the sufficient budget for Ψ^{k+1}. Additionally, we need to compensate for the amount of backlog work that is pushed from Ψ^k to Ψ^{k+1}. This amount is denoted using $\varpi_j(k)$. Basically, the backlog is the part of component load that has missed its deadline and it has postponed to be executed after its deadline point. Therefore, the total budget assigned for the next control period is equal to:

$$B_j(k) = \lambda \times \Big(b_j(k) + \varpi_j(k)\Big), \tag{9.1}$$

where λ is a coefficient that scales down the total budget in a control period to the budget of each server period. If the control period is equal to the server period ($P_j^{Ctrl} = P_j$) then $\lambda = 1$. However, in order to decrease the control overhead we may assign a control period that is larger than the server period. Therefore, in the general case $\lambda = \mu^{-1}$.

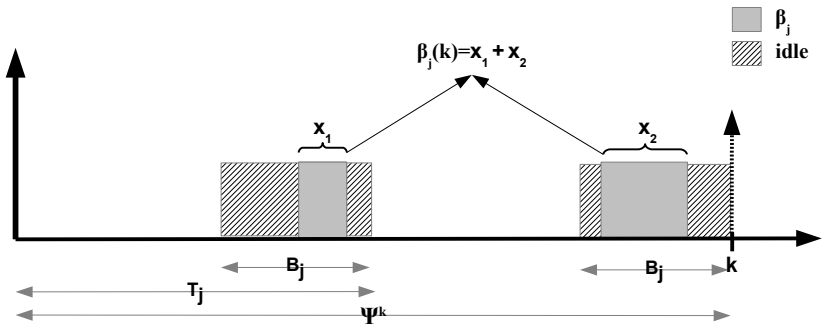
9.3.5 Control parameters

In order to make sense of the state of components, with respect to their processor usage, we need to monitor some scheduling parameters referred to as "control parameters". Recall that the periodic servers idle their budgets when the servers are active and the inner-elements of the components are not using the processor capacity. To this end, we choose to monitor the consumed server budgets. The amount of server budget used by the inner-elements of the component is referred as the actual required budget β_j. When a component wastes some part of its budget we have $\beta_j(k) < \mu \times B_j(k)$. In such a case the budget controller may decide to assign a smaller budget for the next control period of this component. Figure 9.4a visualizes this parameter ($\beta_j(k)$). In this figure we assumed $\mu = 2$, therefore at time k the controller will be triggered and it will observe that $\beta_j(k) = x_1 + x_2$.

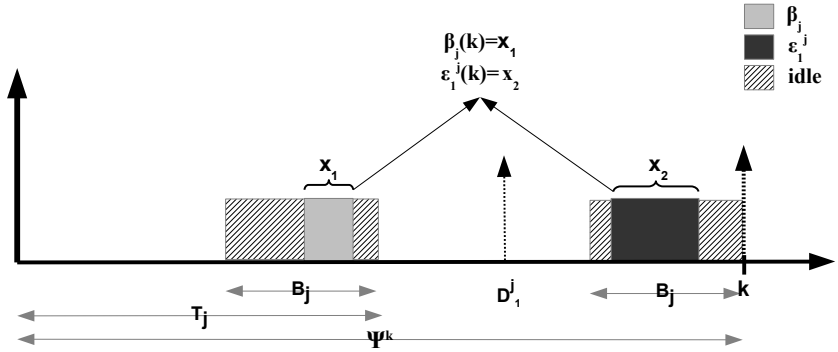
$\beta_j(k)$ can reveal the budget excess state. We also need to detect the budget deficiency problem. The budget deficiency may occur due to two different reasons: (i) the inner tasks of the component may be suffering, i.e., tasks may be missing their deadlines (ii) the inner subcomponents may be suffering, i.e., the subcomponents may not receive their assigned budgets at each period. We use two different metrics to measure the above deficiency sources. We use the execution part of tasks that is scheduled after its deadline point to detect situation (i). Let ϵ_i^j denote the amount of the execution of τ_i^j after its deadline. ϵ_i^j can be translated as the amount of budget deficiency of its parent. For instance, assume that in Figure 9.4b, τ_1^j is the only task in $\mathfrak{C}_j$ and that it misses its deadline at D_1^j, therefore the amount of consumed budget after the deadline $\epsilon_1^j(k)$ is equal to x_2. In order to detect the budget deficiency due to suffering subcomponents (situation (ii) as described above), we can monitor the amount of assigned budget to the subcomponents. If the subcomponents do not receive $B_\kappa^j(k)$ in Ψ^k, i.e., the sum of used budget and idled budget in Ψ^k is less than $B_\kappa^j(k)$, then we can conclude that the subcomponent is suffering from a budget deficiency. We represent the amount of unallocated budget to $\mathfrak{C}_\kappa^j$ using δ_κ^j. Note that δ_κ^j is the amount of budget that the parent was suppose to provide to its child, however due to the parent's budget deficiency it is not provided.

Backlog workload ($\varpi_j(k)$)

The backlog workload, for a component, is produced when either a task misses its deadline or a subcomponent receives a budget less than its assigned budget. In the former case, the task executes ϵ_i^j time units after its deadline. Similarly,



(a) Actual required budget $\beta_j(k)$.



(b) Execution after deadline miss $\epsilon_i^j(k)$.

Figure 9.4: Visualization of the control parameters.

when a subcomponent is suffering from a budget deficiency its inner workload will be postponed to the next server period. If $\mathfrak{C}_\kappa^j$ receives δ_κ^j less budget than its assigned budget, then δ_κ^j workload is postponed to the next server instance. These workload backlogs result in pushing forward the workloads of the next instances. In order to stop this domino effect, we choose to compensate for the backlog workloads at each control event. Therefore, we can derive the amount of backlog workload of $\mathfrak{C}_j$ from Ψ^k that should be compensated in Ψ^{k+1}:

$$\varpi_j(k) = \sum_{i \in S_j} \epsilon_i^j(k) + \sum_{\kappa \in S_j} \delta_\kappa^j(k). \tag{9.2}$$

Using the unassigned budgets for calculating the backlog of the subcomponents allows our approach to be easily applicable for any arbitrary number of levels of hierarchy in the system.

When $\varpi_j(k) > 0$ **and** $\beta_j(k) < \mu \times B(k-1)$

Components' inner-elements may be suffering from a budget deficiency while the component is idling some budget in the same control period. The following two reasons may give rise to the above situation: (i) when the server period is not aligned with the period of the inner-elements; (ii) when the component workload is first decreased and then increased. In either of the above situations the component's budget has to be increased for the next control period. Therefore, at each control event, we first calculate the backlog workload using Equation 9.2. Afterwards, if $\varpi_j(k) > 0$, then the actual required budget is overwritten: $\beta_j(k) = \mu \times B_j(k) + \varpi_j(k)$. The above action means that in the case that backlog exists, the required budget was $\varpi_j(k)$ units more than the assigned budget. Since we use the series of actual required budgets for predicting the future workloads, overwriting $\beta_j(k)$ gives a signal to the controller to increase the budget for the next control periods.

9.3.6 Estimating the future workload

Our budget adaptation mechanism, presented in Equation 9.1, is based on the assumption that the workload of the next control period is known a priori. However, before running the components we do not have information about its future workload. To this end, inspired by the AQuoSA framework [10], we use a workload predictor in our framework. The workload predictor estimates the workload of the next control period based on the observed previous workloads. We use the past observed workloads for estimating the next workload of $\mathfrak{C}_j$. We model the CPU demand of the tasks using the Autoregressive (AR) model, therefore considering h previous $\beta_j(k)$ we have:

$$b_j(k) = \sum_{k-h}^{k-1} w_k \beta_j(k) + e_k, \tag{9.3}$$

where w_k is the weight of observation k and e_k is a Gaussian white noise.

9.3.7 Dealing with overload situations

In some control periods, the system may become overloaded; that is, the sum of the suggested component utilizations may become more than the schedulability threshold provided by the parent component. For instance, assume we use EDF at the root level, the schedulability threshold is one. If the sum of the utilization of the components at the root level, based on the new budget suggestions becomes more than one, then the system becomes overloaded. In the overload situations, the schedulability of the HRT components will not be guaranteed anymore. To this end, we detect overload situations, and we take measures to prevent the overload situations.

We use the following procedure to detect and to prevent the overloads. We first measure the available budget for the component $av^j_\kappa(k)$ by excluding the bandwidth of the other components from the bandwidth of its parent $\alpha_j(k)$:

$$av^j_\kappa(k) = \alpha_j(k) - \sum_{q \in \mathfrak{C}_j \wedge q \neq \kappa} \alpha_q(k).$$

Thereafter, the maximum possible budget is calculated by only excluding the bandwidth of the higher importance components from the parent's bandwidth:

$$ma^j_\kappa(k) = \alpha_j(k) - \sum_{q \in \mathfrak{C}_j \wedge \zeta_q > \zeta_\kappa} \alpha_q(k).$$

Finally, the component's new budget $new^j_\kappa(k)$ is derived using Equation 9.1. Based on the values of $new^j_\kappa(k)$, $av^j_\kappa(k)$ and $ma^j_\kappa(k)$ the following three cases can happen:

1. $new^j_\kappa(k) \leq av^j_\kappa(k)$: in this case we have enough bandwidth, therefore, the controller assigns the new budget to $\mathfrak{C}_\kappa$: $B^j_\kappa(k) = new^j_\kappa(k)$.

2. $av^j_\kappa(k) < new^j_\kappa(k) \leq ma^j_\kappa(k)$: in this case we do not have enough bandwidth, however, we can make enough room for the component by stealing bandwidth from the lower importance components. Therefore, the controller assigns the new budget to the component ($B^j_\kappa(k) = new^j_\kappa(k)$) but it needs to compensate $\Delta B(k) = B^j_\kappa(k) - av^j_\kappa(k)$ by taking the bandwidth from lower importance components. The budget stealing process is done in the reverse order of the component importances, i.e., we start from the lowest importance component, if it does not have enough bandwidth for compensating the stolen bandwidth, then we take all of its bandwidth

and we move to the second lowest importance component to compensate for the remaining stolen bandwidth. This process continues until $\Delta B(k)$ is completely compensated.

3. $new_\kappa^j(k) > ma_\kappa^j(k)$: in this case, the suggested budget cannot be assigned, however, we can assign the maximum possible budget and shut down all lower importance components. The controller assigns $ma_\kappa^j(k)$ to the component ($B_\kappa^j(k) = ma_\kappa^j(k)$) and similar to case (2) it compensates $\Delta B(k)$ by stealing budget from the lower importance components.

In the following we provide an example for further elaborating on our strategy in dealing with overload situations.

Example 1. *Assume a system composed of three SRT components* $\mathfrak{C}_1$, $\mathfrak{C}_2$ *and* $\mathfrak{C}_3$ *where* $\zeta_1 > \zeta_2 > \zeta_3$, $P_1 = P_2 = P_3 = 10$, $B_1(k-1) = 5$, $B_2(k-1) = 3$ *and* $B_3(k-1) = 2$. *At control event* k *the controller decides to increase the budget of* $\mathfrak{C}_2$ *to four. Therefore, we have:* $new_2(k) = 4$, $av_2(k) = 3$ *and* $ma_2(k) = 5$. *In this situation, the controller assigns* $B_2(k) = new_2(k) = 4$ *and decreases* $\Delta B(k) = 1$ *from* $B_3(k)$, *hence, we will have* $B_1(k) = 5$, $B_2(k) = 4$ *and* $B_3(k) = 1$.

9.3.8 Mode change

We provided timing isolation between the HRT and SRT components by (i) using a reservation-based scheduling technique namely periodic servers; (ii) preventing overload situations using the above technique. In the following we point out a scenario in which the timing isolation may be violated despite using the above two strategies. The timing isolation violation may jeopardize the schedulability of the HRT components.

Example 2. *Assume a system composed of the following three components. One HRT component (*$\mathfrak{C}_1$*) and two SRT components (*$\mathfrak{C}_2$ *and* $\mathfrak{C}_3$*) with the same period* $P_1 = P_2 = P_3 = 10$. *The schedule of this system is depicted in Figure 9.5. The initial budgets are as follows:* $B_1 = 5$, $B_2 = 3$ *and* $B_3 = 2$. *An adaptation takes place at time* t_M *and the controller decides to change the budgets as follows:* $B_2 = 2$ *and* $B_3 = 3$. *As a result of this adaptation,* B_1 *which is an HRT component does not receive its five budget units in time. This example shows that the timing isolation may be violated if adaptation is not done carefully at right moments.*

The above problem is similar to that one of the multi-mode real-time systems in which a system is schedulable in two modes while it is unschedulable

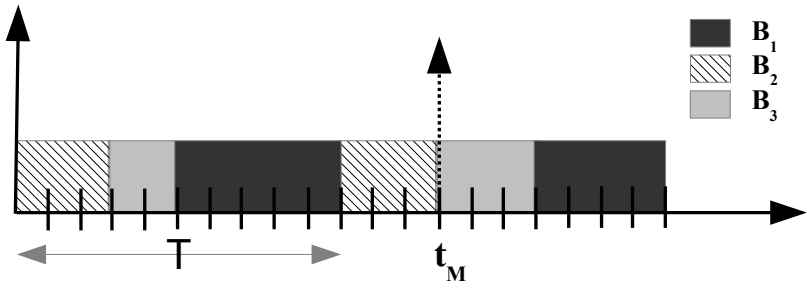


Figure 9.5: Mode change hazard.

during the transition. In the following we propose two solutions for solving this problem.

Using only one SRT component at the root level

A simple way to avoid this problem is to have only one SRT node at the root level ($\mathfrak{C}_s$) and make sure that $B_s \leq U' \times P_s$ where U' is the maximum processor utilization that is available for the SRT components. Assuming EDF at the root level we have:

$$U' = 1 - \sum_{j \in \mathfrak{C}^{hrt}} \alpha_j,$$

where $\mathfrak{C}^{hrt}$ is the set of all HRT components at the root level of the hierarchy.

A mode change protocol

In the multi-mode real-time literature [28] two types of protocols are proposed for a safe mode change in which the schedulability of the transition is guaranteed (i) synchronous protocols (ii) asynchronous protocols. The synchronous protocols do not allow release of the new-mode tasks until the old-mode tasks have finished their executions. This type of protocols is not desirable for our system because our aim is to increase the system performance, while the release delays will introduce some utilization loss. The asynchronous protocols, on the other hand, only provide a test for the schedulability of the transition phase. We propose a protocol which introduces a delay for increasing the budgets and we prove that it can guarantee the schedulability of the transition phase.

For a safe mode change the first condition is to have the system schedulable both in the old-mode and in the new-mode. Assuming that we use EDF as our global scheduler, we must have:

$$\forall\, k \sum_{j \in \mathfrak{C}^{all}} \frac{B_j(k)}{P_j} \leq 1.$$

However, since the utilization of the HRT components is fixed we can exclude it and only check the condition for the utilization of the SRT components. Therefore:

Condition 1.

$$\forall\, k \sum_{j \in \mathfrak{C}^{srt}} \frac{B_j(k)}{P_j} \leq U'.$$

Let us assume that the mode change happens at t_M, then we should show that in all time windows $(t_0, t_1]$ where $t_1 - t_0 = L$ and $t_0 < t_M < t_1$, the following condition holds:

Condition 2.

$$\texttt{dbf}(\mathfrak{C}^{srt}, L) \leq U' \times L,$$

where `dbf` returns the maximum processor demand of its input component set during its input time window. Given that we know the budgets in each mode, we can find an upper bound for `dbf`:

$$\texttt{dbf}(\mathfrak{C}^{srt}, L) \leq L \times \sum_{j \in \mathfrak{C}^{srt} \,\wedge\, k \in (t_0, t_1]} \frac{\max\Big(B_j(k)\Big)}{P_j}.$$

Consequently, it is sufficient to show:

$$\sum_{j \in \mathfrak{C}^{srt} \,\wedge\, k \in (t_0, t_1]} \frac{\max\Big(B_j(k)\Big)}{P_j} \leq U'.$$

Since we use periodic servers it is safe to evaluate time windows with the following length range:

$$0 \leq L \leq LCM(\mathfrak{C}^{srt}),$$

where LCM returns the least common multiple of the periods of its input component set. The LCM might be a large number, therefore similar to [29] we can

find a smaller range for L. However, this problem is out of the scope of this paper. According to Condition 2 the maximum processor demand in the transition phase depends on the maximum of the budget in the two modes. In the following we introduce our mode change protocol. Since we avoid overload situations using the mechanism explained earlier in this section, Condition 1 is always fulfilled. Therefore, in designing the mode change protocol we should address Condition 2. Note that the above conditions are barely used for explaining the rational behind our mode change protocol and they are not meant to be checked at run-time.

The Decrease, Wait, Increase (DWI) protocol: At each mode change there are two types of changes: (i) the budget of some components should be increased ($\mathfrak{C}^{inc}$) (ii) the budget of some other components should be decreased ($\mathfrak{C}^{dec}$). The budget decreases are performed immediately. Afterwards, we wait for the LCM of $\mathfrak{C}^{srt}$ periods. Finally, the budget increases are performed. Note that in contrast to the synchronous mode change protocols, the DWI protocol does not delay the release of the servers and it only delays the budget increases.

Lemma 1. *The DWI protocol fulfills Condition 2.*

Proof. We prove the lemma by contradiction. Assume that the system is changing its mode from mode one with budget B_j to mode two with budget B'_j, and also assume that Condition 2 does not hold, hence:

$$\exists\, \mathcal{W} \;\Big|\; \mathcal{W} \leq LCM(\mathfrak{C}^{srt}) \wedge \sum_{j \in \mathfrak{C}^{srt} \wedge k \in \mathcal{W}} \frac{max\Big(B_j(k)\Big)}{P_j} > U'.$$

Given that Condition 1 holds in both modes:

$$\exists\, \mathcal{W}, i, j, k \;\Big|\; \mathcal{W} \leq LCM(\mathfrak{C}^{srt}) \wedge\; B_j^{dec}(k) = B_j \;\wedge\; B_i^{inc}(k) = B'_i,$$

however, according to the DWI protocol there is at least $LCM(\mathfrak{C}^{srt})$ time units distance between B_j and B'_i and they can not happen in the same $\mathcal{W}$. Hence, Lemma 1 is proved. □

9.4 Implementation

We have implemented our framework in the Linux kernel. In this section we present the implementation details of our Linux kernel loadable module, called

Adaptive **Hier**archical **Sched**uling (`AdHierSched`) module[1]. Our aim was to implement the framework without modifying the Linux kernel. Therefore, we used a similar idea to [18] in which the kernel loadable modules are used for performing real-time scheduling. Basically, the module plays a middleware role between real-time tasks and the Linux kernel. The module has full control on releasing, running and stopping the real-time tasks. When a task has to run, `AdHierSched` inserts it into the Linux run queue and changes its state to running. However, when it wants to stop a task, it removes the task from the Linux run queue and the task goes to the sleep state. Consequently, we keep at most one real-time task (priority 0 to 99) in the Linux run queue at any point in time. Regardless of the selected Linux real-time scheduling class, the `schedule()` system call will always select the single real-time task that is in the Linux run queue if it exists. Otherwise, background tasks (i.e., non-real-time tasks) get a chance to execute. Figure 9.6 puts the role of the `AdHierSched` module into perspective by illustrating its relation with the Linux run queue.

We use the low-resolution timers available in `kernel/timer.c` for managing the time triggered scheduling events. We use one timer per task for handling the task releases. We also use two timers per server for server release and budget depletion events. `AdHierSched` does not have a release queue and instead it delegates the job of the release queue to the Linux timer list. Since the Linux timer list is implemented using the red-black trees, when the number of timers increases, retrieving and inserting them are still efficient ($O(logn)$). The timers are inserted using the `setup_timer_on_stack` and `mod_timer` system calls, and removed using the `del_timer` system call. We use the `jiffies` variable available in the kernel which return the current time for converting the relative scheduling parameters to absolute parameters.

The `AdHierSched` module uses a task descriptor as well as a server descriptor for storing parameters corresponding to the tasks and servers. The task descriptor is presented in Code Snippet 2. The `period_timer` member (line 17) is used for periodically release of the tasks. We set the `period_timer` to the next release of the task when the task finishes its current job. Each `AdHierSched` task points to a Linux task (line 16). Tasks may be attached to a periodic server (line 18). We use two members for storing the adaptation related parameters: `dl_miss` in line 7 and `dl_miss_amount` in line 15. The `timestamp` member is used for measuring the duration of the scheduling events such as the duration that tasks are assigned to the CPU.

[1] The source code is available at:
`http://www.idt.mdh.se/~adhiersched`.

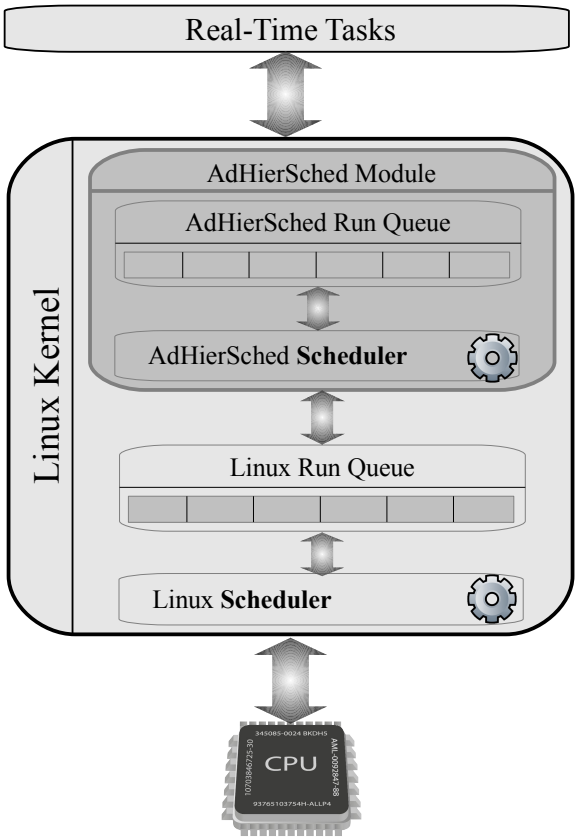


Figure 9.6: `AdHierSched` module.

The server descriptor is presented in Code Snippet 3. We store a pointer variable pointing to the scheduling elements, i.e. tasks and servers, that are inside the server (`children` member in line 3). The `control_period` field in line 7 stores the period of the budget adaptation (P^{Ctrl}). We store the remaining budget in `current_budget`. The fields from line 14 to line 16 are used for the budget adaptation purpose. The servers have their own ready queues (line 18). At each point in time, the ready queue contains the child tasks and servers that are ready to run. The `period_timer` is used for periodical release of the servers. The `budget_timer` is used for handling the budget

Code Snippet 2: Task descriptor.

```
1: struct Task {
2: struct list_head head;
3: int id;
4: int priority;
5: int state;
6: int cnt; /* job number */
7: int dl_miss; /* number of deadline misses */
8: int missing_dl_flag;
9: unsigned long period;
10: unsigned long release_time;
11: unsigned long exec_time;
12: unsigned long relative_deadline;
13: unsigned long abs_deadline;
14: unsigned long timestamp;
15: unsigned long dl_miss_amount;
16: struct task_struct *linux_task;
17: struct timer_list period_timer;
18: struct Server *parent;};
```

depletion events.

We have implemented two types of timer handlers corresponding to the release events and the budget depletion event. Each type is implemented in a separate function. We have defined a generic type called "scheduling element" which covers both tasks and servers. The element is the building block of our ready queue. The ready queues store the scheduling elements in the order of their priority. The ready queue is implemented as a linked list through the `list_head` structure available in the Linux kernel. We have implemented two functions for inserting/deleting an element to/from queue:

- `insert_queue(queue, element)`
- `delete_queue(element)`

When the scheduling policy is fixed priority, the `insert_queue` function inserts the new elements based on their priorities. On the other hand, when the scheduling policy is EDF, the insertion to the ready queue is based on the `abs_deadline` of the scheduling elements. Note that we use multiple ready

Code Snippet 3: Server descriptor.

```
1: struct Server {
2: struct list_head head;
3: Children children;
4: int id;
5: int priority;
6: int cnt; /* number of jobs */
7: int control_period;
8: int importance; /* ζ*/
9: unsigned long budget;
10: unsigned long period;
11: unsigned long relative_deadline;
12: unsigned long abs_deadline;
13: unsigned long current_budget;
14: unsigned long consumed_budget;
15: unsigned long extra_req_budget;
16: unsigned long total_budget;
17: unsigned long timestamp;
18: struct Queue *ready_queue;
19: struct timer_list period_timer;
20: struct timer_list budget_timer;
21: struct Server *parent; };
```

queues (one queue per server). Let q_i^j represent the total number of elements that belong to $\mathfrak{C}_i^j$. The complexity of the insertion to the queue is $O(q_i^j)$.

At each scheduling event at most two insertions are required. If the newly released element can preempt the already running element, then we need to stop the running task and the active server. Both the task and the server are inserted into their corresponding ready queues (two insertions). On the other hand, if the released element cannot preempt the running element, then we only need to insert the newly released element into its corresponding ready queue. Retrieving elements from the queues is done with constant time complexity. Let d^{max} represent the maximum depth of the system model, i.e. the hierarchy depth. Retrieving the running task at each scheduling event is done with complexity of $O(d^{max})$. Assuming that q^{max} represents the maximum number of elements in all components, then the complexity of the entire scheduling event is $O(q^{max}d^{max})$.

9.4.1 Communication between tasks and `AdHierSched`

We use a device file as a communication medium between the tasks and `AdHierSched`. A number of API functions are provided by `AdHierSched`. The API functions use the `ioctl()` system call for the communication purpose. When the message is delivered to the `AdHierSched` module, it relays the message to the message's corresponding function. Table 9.1 presents the list of provided API functions. In the following we explain some of the API functions. We believe the names of the rest of the functions are self explanatory. All of the defined servers and tasks are released when `AdHierSched` receives a `run()` message. In doing so, all scheduling elements are released synchronously. The `stop()` function first stops inserting new timers to the timer list, i.e. it stops the release events. Secondly, it calls the `wake_up_process()` system call for all of the tasks that are still running. In other words, when the `stop()` function is called, the `AdHierSched` module no longer operates and Linux takes the complete responsibility of scheduling the real-time tasks. Jobs need to report their execution end to `AdHierSched`. Therefore, at the end of job executions the `task_finish_job(task_id)` function should be called. This call indeed changes the task status to sleep until the next release of the task. Note that it is possible to add/remove tasks and servers through the API functions while the module is running.

run()
stop()
create_task()
detach_task(task_id)
release_task(task_id)
task_finish_job(task_id)
detach_server(server_id)
release_server(server_id)
attach_task_to_mod(task_id)
create_server(queue_type, server_type)
attach_server_to_server(server_id, server_id2)
attach_task_to_server(server_id, task_id, server_type)
set_task_param(task_id, period, deadline, exec_time, priority)
set_server_param(server_id, period, deadline, budget, priority, server_type)

Table 9.1: List of provided API functions by `AdHierSched` library.

9.4.2 Configuration and run

The API functions make it possible to configure the system such that the component structures as well as the system structure is implemented. In other words, we can create components, attach tasks to the components and attach subcomponents to the parent components. Once the system design is done, we need to attach Linux tasks to `AdHierSched` using the `attach_task_to_mod(task_id)` API function. A sample task structure is presented in Code Snippet 4.

Code Snippet 4: Sample task structure.

```
1: int main(int argc, char* argv[]){
2: task_id = atoi(argv[1]);
3: attach_task_to_mod(task_id);
4: while i < job_no do
5:    /* periodic job */
6:    task_finish_job(task_id);
7: end while
8: detach_task(task_id);
9: return 0; }
```

As mentioned earlier, we call the `run()` function for starting the module. When `AdHierSched` receives a `run()` call, it releases all servers and tasks and then tries to run them. Among all released scheduling elements at the root level of the hierarchy, the one that has the highest priority or shortest deadline (depending on the global level scheduling policy) will be assigned to the CPU. If a component is assigned to the CPU, it will try to run an element from its local ready queue. If from the component's ready queue a subcomponent receives the CPU, the local ready queue running operation continues until the scheduler decides to run a task. As soon as server S_i^j becomes active, we insert its corresponding budget depletion timer (`budget_timer`) to be invoked at time t_{dep}, where:

$$t_{dep} = \texttt{jiffies} + B_i^j(k),$$

where $B_i^j(k)$ is the current budget of $\mathfrak{C}_i^j$. When the `jiffies` is equal to t_{dep}, the budget depletion timer handler is invoked. The handler deactivates its corresponding server (S_i^j) and all of its child servers. If S_i^j is an ancestor (see Definition 1) of the active server, we also stop the active server. After deactivating the servers, we need to check the running task to see if its server is

deactivated or not. In case that its parent server is deactivated, the running task is also stopped. Finally, the timer handler runs the first element that is in the ready queue of S_j (the parent of the server whose budget is depleted). When a server is stopped (either because of its parent budget depletion or because of a preemption), its remaining budget is updated. We have the following scheduling events in the system.

- task and server release
- server budget depletion
- task finishing its job
- task and servers leaving the system

Since `AdHierSched` implements hierarchical scheduling, we use a ready queue per component. Additionally, we have a global ready queue in which the root level elements are placed. Each scheduling element belongs to a ready queue. When an element causes a scheduling event, the event takes place at its corresponding ready queue. At task release events, we first compare the active ready queue with the ready queue of the released task. If the active ready queue is different than the one that the task belongs to, then the released task waits until its parent component is activated. When a server is released, it should wait unless one of the following conditions hold in which the released server is allowed to preempt the active server or the running task.

- The server's parent is active and the released server can preempt (due to a higher priority/earlier deadline) the running/active scheduling element.
- The released server outranks (see Definition 2) the active server.

9.4.3 Budget adaptation

We have implemented a function for performing the budget adaptations. This function is called at certain server release events (depending on the control period P^{Ctrl}). When calling the budget adapter function, the pointer to the caller server structure is passed to the function. The control variable $\beta_j(k)$ is stored in the `consumed_budget` field. On the other hand, the `extra_req_budget` fields stores accumulation of both δ_κ^j and ϵ_i^j variables.

9.5 Evaluations

In this section, we present our evaluations. We used an Intel Core i5-3570 processor clocked at 3.40 GHz in which only CPU 0 is active. Our hardware is equipped with 8 GB of memory. In addition, Ubuntu 12.04.4 with Linux kernel version 3.13.7 is used in the evaluations. The scheduler resolution (system tick) is set to one millisecond. The weight values of the workload predictor (Equation 9.3) are all set to $1/h$, where h is the number of observed actual required budgets. We set $e_k = 1/2 \times std(\beta_j(k), h)$ where std returns the standard deviation of its h previous $\beta_j(k)$. Since the controller has low overhead (see Section 9.5.5) we set $\mu = 1$. We first design a sample component. We, then, use this component to perform the evaluations. We have evaluated our framework in different settings. First, we studied the performance under different history length settings (h). We, then, studied the performance under different server period settings. Afterwards, we studied the scalability by varying the number of components. Finally, we created a three-level hierarchical system, and we evaluated the effect of hierarchy depth assuming both adaptive and fixed budget allocations. The experiment duration was $100s$ in all settings. We use the following three evaluation metrics in this section. (i) The task deadline miss ratio, that is, the number of jobs which have missed their deadline divided by the total number of released jobs. (ii) Control overhead, which is, the accumulative time spent executing the budget controller function. (iii) Overall overhead, that is, the accumulative time spent on running the hierarchical scheduler as well as the budget controller function.

Component 1 (Vision component)**.** *We consider a SRT component $\mathfrak{C}_1$ consisting of three tasks: τ_1^1, τ_2^1 and τ_3^1. τ_1^1 and τ_2^1 are video decoder tasks while τ_3^1 is a static task. This component could, for instance, represent a vision component of a robot which analyses two video streams each related to a different camera (τ_1^1 and τ_2^1). This component also has a task (τ_3^1) which performs a fixed number of instructions on the result of decoded frames at each period. We assume that the tasks within the component are scheduled using the EDF scheduling policy. The execution time of τ_3^1 was $5ms$. The execution time distribution of τ_1^1 and τ_2^1 is presented in Figure 9.7. We assume $T_1^1 = 80ms$, $T_2^1 = 40ms$ and $T_3^1 = 100ms$.*

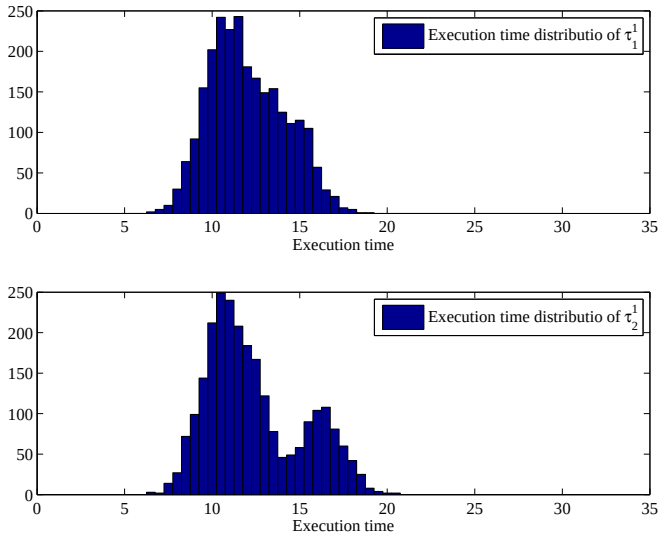


Figure 9.7: The distribution of the execution times of the decoder tasks of the vision component.

9.5.1 One component

We used Component 1 for evaluating the budget adapter module. We created a periodic server with period equal to $50ms$, and we let the budget controller adapt the budget during run-time. We have performed the evaluations using different history lengths (h). Figure 9.8 shows the budget adaptations for different values of h. The figure shows that when using small values of h, the budget adapter responds to transient demand changes faster than the adapters that use large values of h. In particular, when we have a drop in the workload in the beginning of the experiment (after the first second), it is easy to observe that lower values of h provide faster responses. However, faster responses to the transient workload changes does not necessarily correspond to lower deadline miss ratio. We show the deadline miss ratio as well as the control overhead corresponding to different values of h in Figure 9.9. This figure shows that there is a significant reduction in the deadline miss ratio when increasing h from 5 to 10, while further increments do not have a significant effect on the deadline

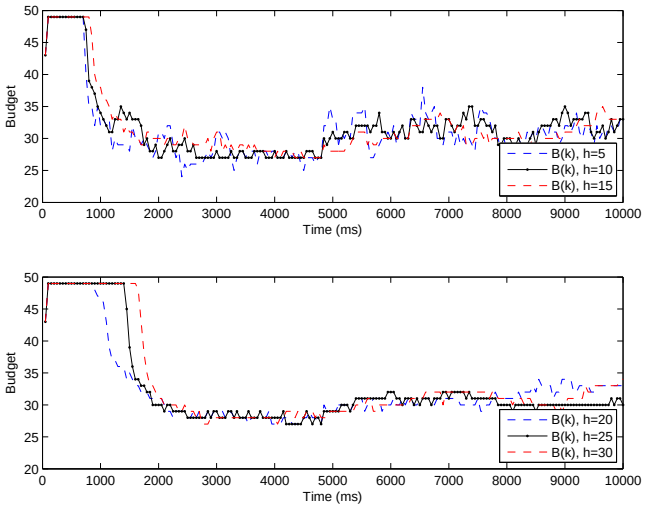


Figure 9.8: The evolution of the assigned budget corresponding to different values of h.

miss ratio. This may be due to the fact than the estimator with $h = 5$ is too sensitive to transient changes in the load. As a result, when the load is reduced transiently, the estimator decreases the budget. Thereafter, when the load is increased again, the controller will need a number of sampling periods to increase the budget to a sufficient amount which results in a number of deadline violations. On the other hand, the control overhead is linearly increased with h. Therefore, considering both the deadline miss ratio and the overhead, we select $h = 10$ for the rest of the experiments.

For a group of soft real-time tasks, missing the deadline point may be acceptable given that the tasks finish their job execution close enough to their deadline points. We use the notion of job tardiness (Θ_i) [30] which shows the distance of a job deadline point and the end of its execution. If a task τ_i misses its deadline, then we have $\Theta_i < 0$, while $\Theta_i \geq 0$ means that the task has finished its execution in time. Figure 9.10 shows the tardiness of the tasks within the vision component related to the above experiment. Let $\min(\Theta_i^j)$ represent a function which returns the minimum observed value of Θ_i^j. We had $\min(\Theta_1^1) = -89ms$, $\min(\Theta_2^1) = -75ms$, and $\min(\Theta_3^1) = -105ms$. There-

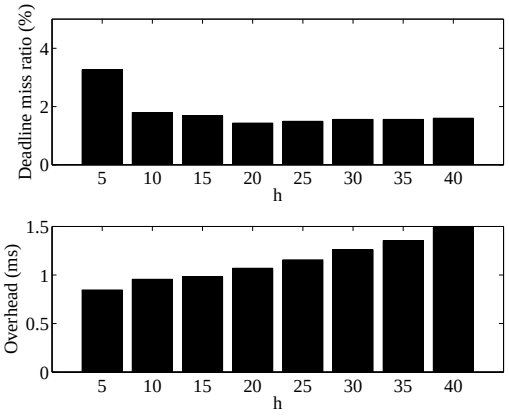


Figure 9.9: The deadline miss ratio and the control overhead for different values of h.

fore, given the period of the video decoder tasks ($T_1^1 = 80ms$, $T_2^1 = 40ms$), both tasks were at most two frames behind their schedule.

9.5.2 Varying the server period

In a new set of experiments, we studied the effect of varying the server period on the system performance. Figure 9.11 illustrates the effect of increasing server period on the following three variables: (i) deadline miss ratio; (ii) the overhead imposed to the system by the budget controller, (iii) the overall overhead including the scheduling overhead and the control overhead. Our conclusion from this experiment is as follows. The deadline miss ratio is sensitive to the server period, and the controller cannot compensate for poor period assignments. Besides, although a short server period imposes slightly higher overhead than the large server periods, the deadline miss ratio rises significantly with the increase of the period. Therefore, the system designers should carefully study the performance of the components under different server periods.

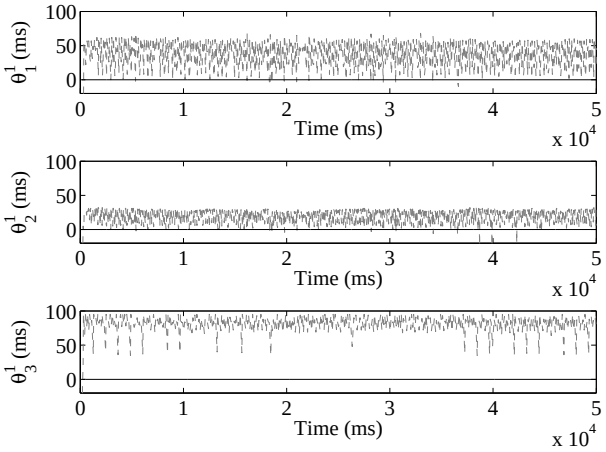


Figure 9.10: Tardiness of the three tasks inside the vision component assuming $h = 10$.

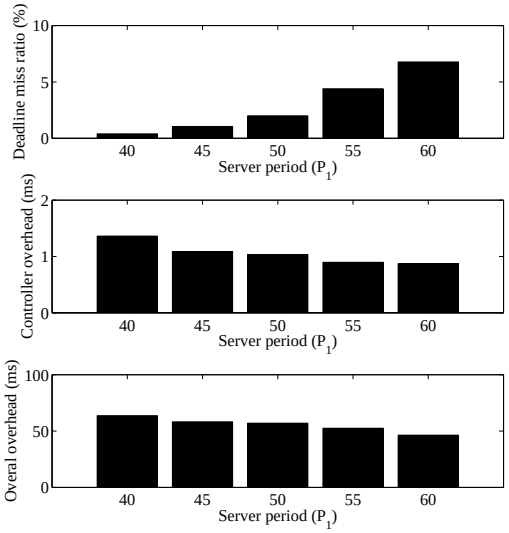


Figure 9.11: Deadline miss ratio, budget controller overhead and overall overhead against server period variations.

9.5.3 Higher number of components

We conducted another set of experiments to study the scalability of our framework. In these experiments we varied the number of components from one to four. In other words, we conducted four experiments. In the first experiment we used one component, in the second experiment we used two components, and similarly in the third and forth experiments we used three and four components. We used the same component as the previous experiments. However, we scaled down the execution times in order to be able to integrate four components on the processor. We assigned the following periods to the server of the components: $P_1 = 35ms, P_2 = 40ms, P_3 = 45ms, P_4 = 50ms$. We assumed the following importance order for the components: $\zeta_1 > \zeta_2 > \zeta_3 > \zeta_4$.

Figure 9.12 shows the deadline miss ratio experienced by the components in the above four experiments. In all of the experiments $\mathfrak{C}_1$ experienced the lowest deadline miss ratio. This is because $\mathfrak{C}_1$ has the highest importance among all other components. Therefore, in overload situations, this component is favored over the other components. Figure 9.13 shows the control overhead, the overall overhead, and the number of overload situations observed in the above experiments. The overall overhead reflects the accumulative value of control overhead and the scheduling overhead imposed by using our framework. Note that when we had four components, the control overhead is increased because of the following two reasons: (i) the number of invocations of the budget controller is increased due to the additional component; (ii) the overload controller is executed several times. Moreover, since the system is overloaded, the lowest importance component experienced the most deadline violations among all components.

9.5.4 Three-level hierarchical system

We considered a system composed of four components. The structure of this system and its specification is presented in Figure 9.14 and Table 9.2 respectively. Note that we use one server per each component i.e., S_i^j contains $\mathfrak{C}_i^j$. The period and initial budget values reported in the table are all in milliseconds. We assumed the following importance order for the components $\zeta_3^2 > \zeta_4^2$. Therefore, in overload situations the controller will steal bandwidth from $\mathfrak{C}_4^2$ and give it to $\mathfrak{C}_3^2$.

In this experiment we created an overload situation to evaluate our framework in such a situation. First we ran the system with enabled budget controller. In this setting $\mathfrak{C}_3^2$ experienced 2.65 $\%$ deadline miss ratio, while $\mathfrak{C}_4^2$

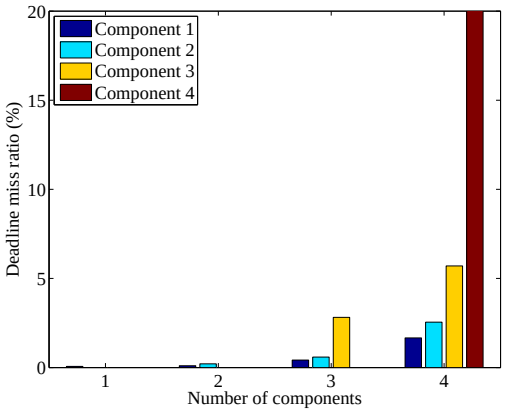


Figure 9.12: The deadline miss ratio of corresponding to the four experiments with one to four components.

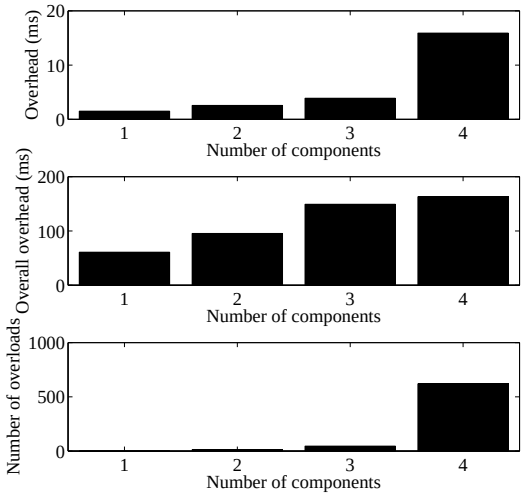


Figure 9.13: The control overhead, overall overhead, and the number of overload situations corresponding to the four experiments.

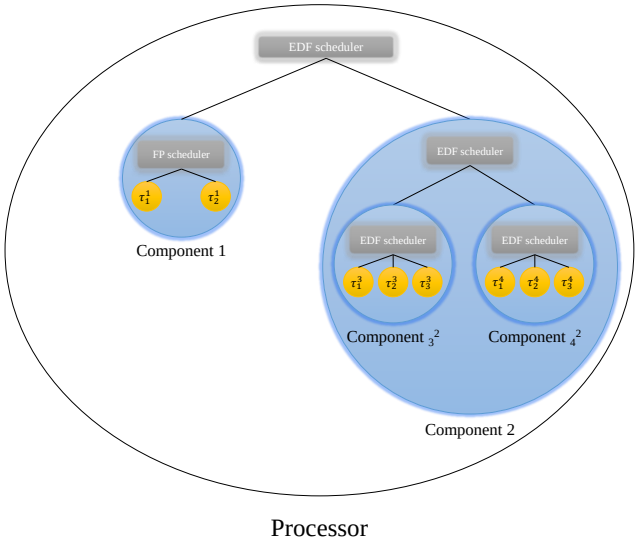


Figure 9.14: The structure of the assumed system.

	Hard-Soft	Type	$P_j - T_i^j$	$B_j(0)$
S_1	HRT server	-	100	39
τ_1^1	HRT task	static	200	31
τ_2^1	HRT task	static	400	31
S_2	SRT server	-	10	6
S_3	SRT server	-	35	8
τ_1^3	SRT task	decoder	80	-
τ_2^3	SRT task	decoder	40	-
τ_3^3	SRT task	static	100	-
S_4	SRT server	-	40	10
τ_1^4	SRT task	decoder	35	-
τ_2^4	SRT task	decoder	40	-
τ_3^4	SRT task	static	100	-

Table 9.2: Specification of servers and tasks used in the evaluation of the three-level hierarchical example.

experienced 53.92 % deadline miss ratio.

In a new experiment, we used the average assigned budgets by the controller in the above setting, and we assigned fixed budgets to the components. The result was 1.71 % deadline miss ratio for $\mathfrak{C}_3$ while the tasks within $\mathfrak{C}_4$ missed all of their deadlines. This is due to the fact that in the beginning of the experiment the jobs miss their deadlines and a backlog was created. Since the backlog was never compensated, the workload kept accumulating and all jobs finished after their deadlines. This experiment reveals that assigning fixed budgets may significantly degrade the performance of the system.

9.5.5 Overhead

The maximum observed control overhead throughout our experiments corresponds to the cases where we had four components (Subsection 9.5.3 and 9.5.4). In the case of two-level components approximately 0.01 % of the experiment time was spent on executing the budget controller function. Considering the overall overhead, 0.16 % of the experiment time was spent on executing the adaptive hierarchical scheduling framework code. Note that our measurements excluded the Linux scheduler overhead that is responsible to assign the real-time tasks to the CPU. Therefore, we did not include the context switch related overhead. Basically, we measured the overhead that `AdHierSched` adds to the scheduler. We measured the overhead using time stamps that monitored the execution length of the timer handlers and the `task_finish_job(task_id)` API function.

The scheduling overhead (excluding the budget controller) observed in the four component three-level hierarchical experiment was 0.03 % more than the scheduling overhead of the four component two-level experiment. This observation suggests that increasing the hierarchy depth increases the scheduling overhead, however, the overhead increase is insignificant.

9.6 Conclusion

In this paper we presented a multi-level adaptive hierarchical scheduling framework for scheduling real-time systems developed using the component-based software development paradigm. In our framework, we assign adaptive CPU partition sizes to soft real-time components based on feedbacks from their workload status. We showed that by imposing a negligible overhead (less than 0.2 % of the CPU time in a system consisting of four components) we are able

to serve real-time components such that they reach an acceptable deadline miss ratio. In addition, we presented the implementation details of our Linux kernel loadable module, called `AdHierSched`, which implements our adaptive framework in the Linux kernel.

Although we are not currently considering I/O operations, we would like to investigate the implications of modeling them in our adaptive framework. For instance, we can model the I/O requests as critical sections and we can use available semaphore based protocols such as SIRAP [31] and HSRP [32]. We also want to study the use of more sophisticated workload estimator modules in our framework to examine whether we could better serve the components by more accurate workload estimations.

References

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

[2] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, May 2001.

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

[4] N. Khalilzad, M. Behnam, and T. Nolte. Multi-level adaptive hierarchical scheduling framework for composing real-time systems. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages 320–329, August 2013.

[5] F. Zhang and A. Burns. Analysis of hierarchical EDF pre-emptive scheduling. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 423–434, December 2007.

[6] J. A. Stankovic, C. Lu, S.H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS'99)*, pages 11–20, June 1999.

[7] C. Lu, J. A. Stankovic, G. Tao, and S.H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 56–67, December 1999.

[8] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzen. Feedbackfeedforward scheduling of control tasks. *Real-Time Systems*, pages 25–53, 2002.

[9] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 4–13, December 1998.

[10] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, January 2009.

[11] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino. Resource management on multicore systems: The ACTORS approach. *Micro, IEEE*, 31(3):72–81, May-June 2011.

[12] N. Khalilzad, T. Nolte, M. Behnam, and M. Åsberg. Towards adaptive hierarchical scheduling of real-time systems. In *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11)*, pages 1–8, September 2011.

[13] N. Khalilzad, M. Behnam, G. Spampinato, and T. Nolte. Bandwidth adaptation in hierarchical scheduling using fuzzy controllers. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 148–157, June 2012.

[14] M. Barabanov and V. Yodaiken. Real-time linux. *Linux journal*, 23, March 1996.

[15] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 2000(72es), April 2000.

[16] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril. Towards hierarchical scheduling on top of VxWorks. In *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008.

[17] R. Inam, J. Maki-Turja, M. Sjodin, M. Ashjaei, and S. Afshar. Support for hierarchical scheduling in FreeRTOS. In *Proceedings of the 16th IEEE International Conference on Emerging Technologies Factory Automation (ETFA'11)*, pages 1–10, September 2011.

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

[19] M. 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 *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10)*, pages 71–81, July 2010.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX symposium on Operating systems design and implementation (OSDI'96)*, pages 137–151, 1996.

[21] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin. Implementation of compositional scheduling framework on virtualization. *SIGBED Rev*, 8:30–37, 2011.

[22] J. Lee, S. Xi, S. Chen, L. T. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. Realizing compositional scheduling through virtualization. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*, pages 13–22, April 2012.

[23] M. Danish, Y. Li, and R. West. Virtual-CPU scheduling in the quest operating system. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, pages 169–179, April 2011.

[24] T. Cucinotta, G. Anastasi, and L. Abeni. Respecting temporal constraints in virtualised services. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, volume 2, pages 73–78, July 2009.

[25] M. Åsberg, N. Forsberg, T. Nolte, and S. Kato. Towards real-time scheduling of virtual machines without kernel modifications. In *Proceedings of the 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress (WiP) session*, pages 1–4, September 2011.

[26] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 3–14, December 2001.

[27] G. Parmer and R. West. HIRES: A system for predictable hierarchical resource management. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, pages 180–190, April 2011.

[28] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[29] B. Andersson. Uniprocessor EDF scheduling with mode change. Technical report, Polytechnic Institute of Porto (ISEP-IPP), January 2008.

[30] A. Srinivasan and J. H. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 51–59, July 2003.

[31] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*, pages 279–288, 2007.

[32] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 257–270, December 2006.

Chapter 10

Paper C: A Feedback Scheduling Framework for Component-Based Soft Real-Time Systems

Nima Khalilzad, Fanxin Kong, Xue Liu, Moris Behnam and Thomas Nolte.
In Proceedings of the 21th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15), April, 2015.

Abstract

Component-based software systems with real-time requirements are often scheduled using processor reservation techniques. Such techniques have mainly evolved around hard real-time systems in which worst-case resource demands are considered for the reservations. In soft real-time systems, reserving the processors based on the worst-case demands results in unnecessary over-allocations.

In this paper, targeting soft real-time systems running on multiprocessor platforms, we focus on components for which processor demand varies during run-time. We propose a feedback scheduling framework where processor reservations are used for scheduling components. The reservation bandwidths as well as the reservation periods are adapted using MIMO LQR controllers. We provide an allocation mechanism for distributing components over processors. The proposed framework is implemented in the TrueTime simulation tool for system identification. We use a case study to investigate the performance of our framework in the simulation tool. Finally, the framework is implemented in the Linux kernel for practical evaluations. The evaluation results suggest that the framework can efficiently adapt the reservation parameters during run-time by imposing negligible overhead.

10.1 Introduction

Multiprocessors are becoming increasingly more widespread computing platforms. Thanks to the computational capacity of the multiprocessors, previously segregated software systems can now be integrated on a shared hardware platform. Component-Based Software Engineering (CBSE) provides a modular approach for designing and developing complex software systems. CBSE provides means and techniques for integration of independently developed software components.

When it comes to real-time systems, timing constraints of software components have to be considered at the integration phase. We consider component models in which a real-time software component corresponds to a set of real-time tasks. A component also has an intra-component scheduler which coordinates task executions. Processor reservation and hierarchical scheduling techniques are often used to provide timing guarantees to the components in component-based systems (e.g., [1, 2]). Therefore, from the real-time scheduling perspective, the problem of component integration is reduced to creating adequate processor reservations for hosting the components.

Real-time tasks can either have hard deadlines where deadline misses are absolutely unacceptable or soft deadlines where occasional deadline misses can be tolerated. A hard real-time component is a component with hard real-time tasks. The size of processor reservations assigned to the hard real-time components is derived from the Worst-Case Execution Time (WCET) of the component's inner tasks. For instance in [3] and [4], targeting multiprocessor platforms, the authors provided analysis frameworks in which the reservation properties are extracted from intra-component schedulers and task parameters. Such analyses result in pessimistic allocations. The over-allocation is due to two reasons. Firstly, WCET is unlikely to happen in reality. Secondly, the analysis that derives the processor reservation sizes based on the WCET of tasks is pessimistic. Soft real-time components are software components consisting of soft real-time tasks. When integrating soft real-time components, pessimistic allocations are not justifiable. This is because pessimistic allocations do not permit an efficient processor utilization. In addition, in a group of soft real-time tasks the processor demand is subjected to large variations during run-time. For instance, the execution time of video decoder tasks can significantly vary depending on the content of the video frames. As a result, the processor demand of a real-time component consisting of such dynamic tasks may change significantly during run-time. Therefore, assigning a fixed-size processor reservation (for instance based on the average processor demands)

results in an unacceptable number of timing violations.

Adaptive reservation techniques are widely used in single-processor platforms for scheduling soft real-time tasks with dynamic execution times (e.g., [5, 6]). In this paper, however, we focus on soft real-time components integrated on multiprocessor platforms. In our model, the components may be spread over multiple processors. As a result, the component's inner tasks are scheduled using a multiprocessor global scheduling algorithm. We propose a feedback scheduling framework which is built upon adaptive reservations. In our framework, the component demand is monitored during run-time. The processors reservations hosting the component are, then, adjusted according to the current demand. The processor allocations are also reconfigured to cope with the current state of the components. More specifically, in this paper, we present the following contributions: (i) a feedback scheduling scheme that uses Multiple Input Multiple Output (MIMO) controllers to regulate both period and budget of the periodic servers simultaneously (ii) an approximate model of the reservation dynamics through system identification (iii) a component allocation heuristic that maps software components to the processors and evaluating it against the optimal solution (iv) optimal compression algorithms that provide compressed bandwidths in overload situations (v) simulation-based evaluation of our MIMO controllers in TrueTime (vi) implementation and evaluation of our framework in the Linux kernel.

10.2 Preliminaries

System model. We assume a multiprocessor platform consisting of M identical processors. n components are running on the multiprocessor platform. We consider an open system in which components are allowed to join and/or leave the platform. As a result, n varies in run-time. The set of components which are active in the system at any given time t is denoted using $\Gamma(t)$.

We target a component-based software development model in which the following two roles are defined: (i) component developer (ii) system integrator. The component developer is responsible for developing real-time tasks and selecting an appropriate scheduling policy for them. Then, the component requirements are abstracted using a number of interface parameters. When it comes to components with hard real-time requirements, a component interface represents the minimum amount of resource needed for guaranteeing the schedulability of the component. Such an interface is calculated using the WCET of the component's inner tasks. In our framework, however, we

are targeting soft real-time components with dynamic workloads. Therefore, a component interface expresses an interval in which the processor demand of the component may vary during run-time. Basically, instead of the worst-case resource demands, the aggregate behavior of all tasks with respect to the processor requirement is expressed in the component interface. The system integrator, on the other hand, receives a number of components and he/she is responsible for integrating the components such that the requirements specified in the interface parameters are respected. The integrators' responsibility involves (i) identifying an approximate model of the component's resource requirements within its operating region (ii) designing controllers that adapt the resource provisions to the components during run-time. In this paper, we focus on the component integration.

Component model. Component $\mathfrak{C}_j$ consists of a number of real-time tasks, where $j \in [1, n]$ is the index of the particular component. The components also have an intra-component scheduler which is responsible for scheduling component's inner task. The relative importance of components with respect to the other components that are composed together on one platform is represented by ζ_j. The importance value is used when the system is overloaded. In such a situation, the components that can better contribute to the overall value of the system are preferred to the ones that have less impact on the total system value. Components can be assigned to one or more processors. We use period and bandwidth for specifying processor requirements of components. The bandwidth indicates the processor portion that a component requires, while the period indicates the granularity of the CPU provisioning. The component developers specify the operating range of their components, that is, a processor demand interval that the component will operate at run-time. The bandwidth requirement is denoted using $\bar{\alpha_j}$ and σ_{α_j}, where $\bar{\alpha_j}$ is the operating bandwidth and σ_{α_j} indicates the amount of deviation from the operating bandwidth. Similarly, the period requirement is specified using an operating period $\bar{T_j}$ and its deviation σ_{T_j}. The component interface $< \bar{\alpha_j},\ \bar{T_j},\ \sigma_{\alpha_j},\ \sigma_{T_j},\ \zeta_j >$ denotes that the component will require a bandwidth between $\bar{\alpha_j} - \sigma_{\alpha_j}/2$ and $\bar{\alpha_j} + \sigma_{\alpha_j}/2$. Similarly, the period may be changed from $\bar{T_j} - \sigma_{T_j}/2$ to $\bar{T_j} + \sigma_{T_j}/2$. The system integrator develops a model in the operating range of the component that is used for adaptation purposes. Note that $\bar{\alpha_j}$ and $\bar{T_j}$ do not need to be exact values, rather they are estimations of the component's processor requirements. We will adapt the resource provisioning to the components during run-time to compensate for the resource requirement estimation errors.

Task model. Our scheme supports periodic/sporadic task models τ_i<p_i, $c_i(l)$, D_i>, where i is the index of the particular task, p_i is the task period or the

minimum inter-arrival time, $c_i(l)$ is the execution cost of the l^{th} instance of the task and D_i is the task deadline. Each instance of task execution is called a job. Note that the execution cost of tasks is time-varying and may be different from job to job. Throughout the paper and for simplicity we use an implicit deadline periodic task model, i.e. $p_i = D_i$. We do not assume any predefined execution cost $c_i(l)$ for tasks, however, we assume a task is not allowed to run in parallel, hence $\forall t\ c_i(l) \leq p_i$. The jobs of a task are executed sequentially, i.e., each job of a task is only allowed to run if all of the previous jobs of the same task have finished their executions. When tasks miss their deadlines, they continue their execution until the end. The goal of our framework is to provide a predictable Quality of Service (QoS) to the tasks, while efficiently utilizing the processor capacity. We use the number of deadline violations as a metric for measuring the QoS.

Virtual clusters and virtual processors. The computation capacity of the multiprocessor platform becomes available to the components through Virtual Clusters (VC). A particular VC i, denoted by Π_i, is a set of Virtual Processors (VP) $\Pi_i = \{\pi_{i,1}, \pi_{i,2}, \cdots\}$, where $\pi_{i,j}$ is the j^{th} VP of Π_i. A VP is created by partitioning a single physical processor in time. We use idling periodic servers compatible with the periodic resource model [7] for partitioning a single physical processor. When the server is active while there is no ready task to run, the idling servers idle their budget. The deadline of servers implementing the VPs is assumed to be equal to their corresponding periods. $\pi_{i,j}$ receives $q_{i,j}$ units of the physical processor time every T_i time units, where $q_{i,j} \leq T_i$. The periods of all VPs belonging to Π_i is equal to T_i. The bandwidth of a VP is defined as $\rho_{i,j} = q_{i,j}/T_i$. We assume Π_i can have at most one VP on any given physical processor. Π_i receives B_i time units every T_i units, where $B_i = \sum_{j \in [1 \ldots M]} q_{i,j}$. In this summation, we assume $q_{i,j} = 0$ for the case where the VC has less than M VPs and $\pi_{i,j}$ does not exist. The bandwidth of Π_i is defined as the following: $\alpha_i = B_i/T_i$. We have n VCs hosting n components at each point in time, i.e., the number of VCs in the system is equal to the number of components n.

Multiple VPs that belong to distinct VCs may share a physical processor. We use the partitioned EDF scheduling algorithm for scheduling the VPs. For scheduling the tasks within the components, however, we use a global multiprocessor scheduler. In other words, when a VC is spread over multiple processors, tasks within the VC may migrate from a processor to another processor. The intra-cluster scheduler (task scheduler) can be either global fixed-priority or global EDF. Considering the two levels of scheduling, our scheme can be seen as a two-level hierarchical scheduling framework.

Operational modes. We consider the following two mutually exclusive operating modes for the system: *normal mode* and *overload mode*. In the normal mode the components can receive their desired processor bandwidths because the total required processor is less than the available processor time, i.e., $\sum_{i\in\Gamma} \alpha_i \leq M$. In the normal mode we use a number of independent MIMO controllers to regulate the bandwidths and the periods of the VCs. In the overload mode, however, the total required bandwidth is larger than the available processor capacity. In this mode the system will suffer, i.e., real-time tasks will inevitably miss their deadlines. Our goal, in the overload mode, is to distribute the total bandwidth among components in such a way that the overall value of the system is maximized. Therefore, utilizing the importance value of the components (ζ_i for each component $\mathfrak{C}_i$), we use a centralized controller to distribute the total bandwidth among components. If a system operates in the overload mode most of the time, then the system is poorly designed and the integrator should remove some of the components to reduce the load. We assume that the overload mode happens transiently, and the system mostly operates in the normal mode.

Overview of the framework. Figure 10.1 depicts the architecture of our adaptive framework. The framework is comprised of two types of elements: (1) cluster controllers (2) a resource manager. The cluster controllers monitor the state of the VCs and adapt their bandwidths and periods to deal with components' dynamic resource requirements. The cluster controllers are designed using control theory. Section 10.3 describes the cluster controllers in detail. The resource manager, on the other hand, is responsible for allocating components on the processors. The resource manager receives n VCs and it allocates each VC on a number of VPc. The resource manager adds, removes and adjusts VPs dynamically to respond to the needs of the VCs. Section 10.4 addresses the design of the resource manager.

10.3 Modeling and design of cluster controllers

In this section we focus on adapting the parameters of a single VC serving a component. Therefore, for simplicity, we drop index i when referring to parameters associated with Π_i. Throughout this section we assume that the system is in the normal mode. The cluster dynamics are sampled and adapted periodically. The sampling time is denoted using k. The time distance of two consecutive samples is referred as a *sampling interval* and its length is denoted using Ψ.

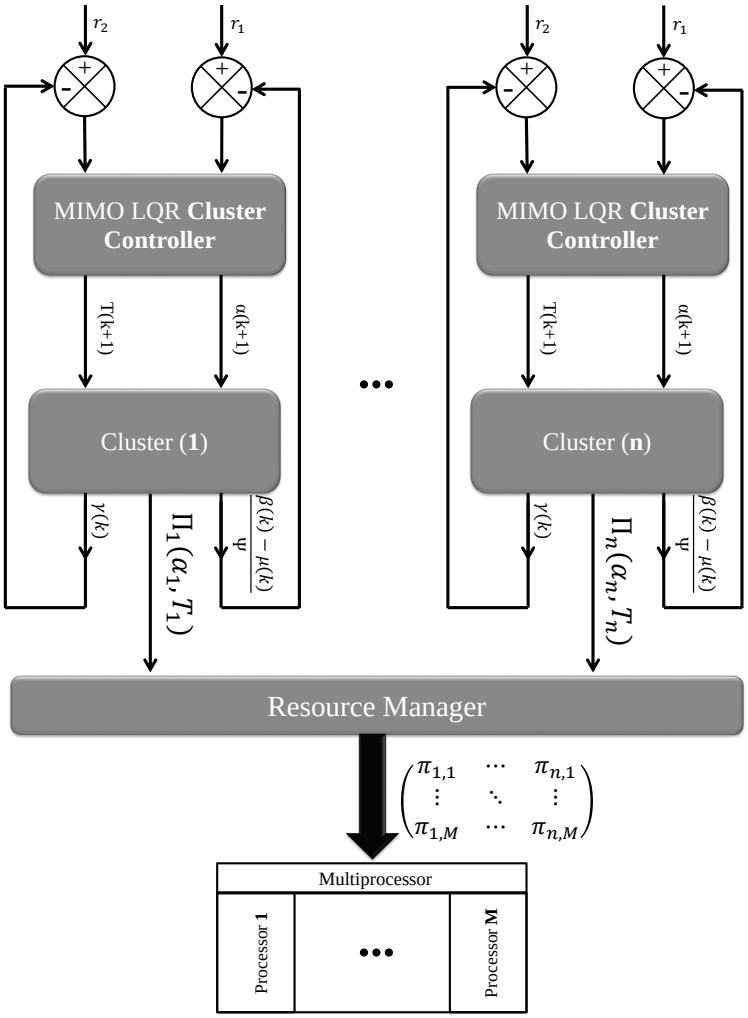


Figure 10.1: The architecture of our adaptive component-based scheduling framework.

In control theory, control inputs are variables that are used for manipulating the plant. We consider the VCs as our plant. Our objective is to make sure that the VCs provide sufficient processor capacities to the components at each point in time. Therefore, we choose T and α as our control inputs. We use the parameters expressed in the component's interface as the operating points and we take the distance from the operating points as our control inputs. Therefore we have:

$$\mathbf{u}(k) = \begin{bmatrix} \alpha(k) - \bar{\alpha} \\ T(k) - \bar{T} \end{bmatrix},$$

where $\mathbf{u}(k)$ is the control input at sampling time k. We construct our model around the operating points of the system. The reason behind using the operating points is that, the plant's behavior can be approximated in the vicinity of these points using a linear model.

10.3.1 Why should the cluster periods be adapted?

At first glance it might appear that changing the VC bandwidths through adapting their budgets might be sufficient. However, there are a number of good reasons for adapting the VC periods as well. Let's first discuss the problems associated with two extremes of period assignment, i.e., extremely short periods and extremely large periods. As the VC period decreases, the number of preemptions in a given time interval increases. Therefore, considering the overhead penalty associated with preemptions, it is desirable to assign periods as large as possible. Extremely large periods, on the other hand, impose insignificant overhead. However, when tasks are faster than their VPs, the VP bandwidths have to be significantly larger than the task set's processor utilization, because the budget provisioning may not be aligned with the task executions. We refer to this problem as the *alignment problem* which is illustrated in Figure 10.2. The importance of the granularity of a resource partition is also studied in [8]. In addition, in case of sporadic task models, the tasks may occasionally use their minimum inter-arrival times. Hence, assigning the VC period based on the minimum inter-arrival times will impose unnecessary overhead to the system.

Measurable variables. For controlling the VC parameters, we need a number of variables that can describe the dynamics of the VCs. We should choose parameters that (i) can be easily measured (ii) be an indication of the workload and task frequencies. In fact, we consider the changes in workload as a disturbance and our control objective is to compensate for it.

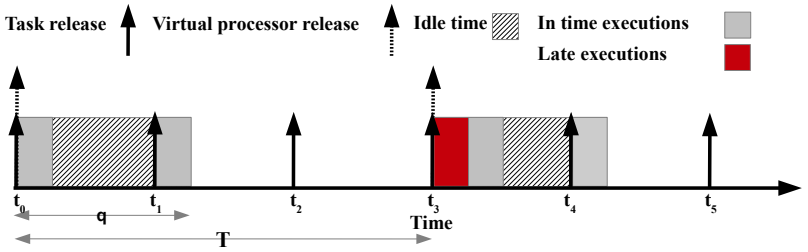
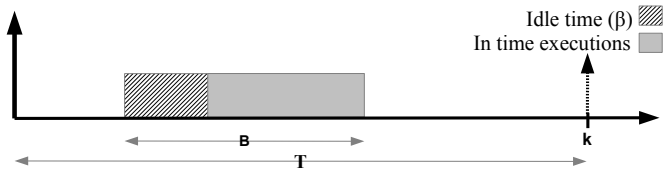


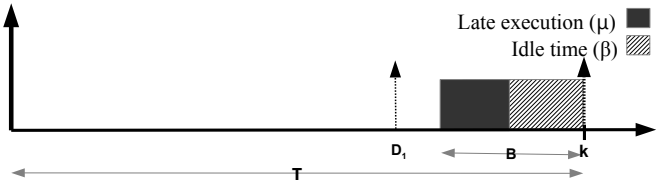
Figure 10.2: Alignment problem: VP's period is not aligned with the task period. Task is released at time t_2 while the VP is inactive. The task has to wait until the next VP release which is after task's deadline. Therefore, the task misses its deadline while the VP budget is idled.

The cluster Π is assigned $B(k)$ time units every $T(k)$ time units. It idles $\beta(k)$ time units of its budget due to unavailability of workload and utilizes the rest of its budget ($B(k) - \beta(k)$). Tasks inside the VC either finish their executions before their deadlines or after them. The part of task's execution time executed after task's deadline is called a *late execution*. The part of the VC budget consumed by the late executions is denoted by $\mu(k)$. At any sampling time k, the cluster controller can measure $\beta(k)$ and $\mu(k)$. Note that $\beta(k)$ and $\mu(k)$ are respectively the aggregate values of the idled budget and the late executions happened in a sampling interval. These parameters are illustrated in Figure 10.3a and Figure 10.3b. The number of jobs that have missed their deadlines is another variable that can be monitored by the cluster controller. The number of deadline misses happened between sampling times $k-1$ and k is denoted by $\gamma(k)$.

State variables. We intend to use a linear model for modeling the dynamics between the inputs and the state variables. Thus, we are interested in variables that their changes, with respect to the changes of the inputs, are as close as possible to linear. Since both $\beta(k)$ and $\mu(k)$ are saturated at zero we use the following linear combination of them as our first state variable: $x_1(k) = (\beta(k) - \mu(k))/\Psi$. Note that this state variable is normalized by the sampling length Ψ. The processor resource over-allocation ($x_1(k) > 0$) and under-allocation ($x_1(k) < 0$) to the components is revealed by $x_1(k)$. However, when the idle time is equal to the late execution time ($\beta(k) = \mu(k)$), or when the late execution time is significantly smaller than the idle time ($\mu(k) \ll \beta(k)$), components may suffer from deadline misses while $x_1(k)$



(a) Idle time (β).



(b) Late execution (μ).

Figure 10.3: Visualization of the measurable variables.

is not revealing the state of the VCs. To address this problem, we choose to further monitor the number of deadline misses happened in a sampling interval. Thus, the second state variable is: $x_2(k) = \gamma(k)$. This variable can further express the state of VCs when $x_1(k)$ is not expressive. In summary, we use the following state variables:

$$\mathbf{x}(k) = \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} = \begin{bmatrix} \frac{\beta(k)-\mu(k)}{\Psi} \\ \gamma(k) \end{bmatrix}.$$

Suppose that $\alpha^*(k)$ and $T^*(k)$ are a bandwidth and a period in which $x_1(k) = r_1$ and $x_2(k) = r_2$, where r_1 and r_2 are desired values of $x_1(k)$ and $x_2(k)$ respectively. Assuming that $T(k) = T^*(k)$ if $\alpha(k) > \alpha^*(k)$ the VC will waste some of its budget, hence $x_1(k) > 0$. If $\alpha(k) < \alpha^*(k)$ the VC will suffer from a budget deficiency and $x_1(k) < 0, x_2(k) > 0$. Assuming that $\alpha(k) = \alpha^*(k)$, if $T(k) > T^*(k)$ the VC will suffer from the alignment problem, therefore $x_1(k), x_2(k) > 0$. If $T(k) < T^*(k)$, the VC will impose some overhead due to short periods and $x_2(k) < 0$. Note that in order to detect this case, i.e., $T(k) < T^*(k)$ we have to set r_2 to a small number greater than zero. As discussed above, the designed state variables reveal the internal states of the VCs.

10.3.2 Modeling the cluster dynamics

We are interested in deriving a model which captures the relation between the control inputs and the state variables. Throughout our experiments we observed that this relation is not linear due to (i) queuing effects of **u** on **x** (ii) saturation of **x**. The queuing effect is due to task scheduling. For instance, increasing the VC bandwidth does not necessarily reduce the number of deadline misses. This is because some tasks may have backlogs from the previous sampling interval. Therefore, increasing the bandwidth will allow them to execute in the next sampling interval. Saturation of **x** happens due to the nature of our system. For example, at most all of jobs of all tasks within the VC can miss their deadlines. Therefore, in such a condition that all jobs miss their deadlines, decreasing the bandwidth will not have any influence in the number of deadline misses. Despite the non-linearity nature of our system, linear models often work well for nonlinear systems specially when the purpose is to regulate the system output based on a number of control inputs [9]. We use the so called "black box" approach for modeling the relation between **u** and **x**. We employ the Auto-Regressive with eXogenous variables (ARX) model to describe the relation between the state variables and the inputs. Therefore, the state space system model is as follows:

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k), \end{aligned} \tag{10.1}$$

where **A**, **B** and **C** are 2×2 matrices, $\mathbf{u}(k)$ is the control input, $\mathbf{y}(k)$ is the system output, and for simplicity we assume $\mathbf{C} = \mathbf{I}$ (**I** is the identity matrix). Matrix **A** indicates the dependency between the next value of outputs to their previous values. Matrix **B**, however, expresses the functional dependency between the control inputs and the system outputs.

10.3.3 System identification

We use system identification for identifying matrices **A** and **B** of the model presented in Eq. 10.1. System identification uses statistical tools to estimate the model parameters. The identification processor is as follows. First the components are executed on the target hardware platform. The control inputs (**u**) are modified throughout the execution of the components and the system outputs (**y**) are noted. Finally, parameter estimation is performed given **u** and **y**. Similar to WCET analysis which is hardware dependent, the identified parameters may depend on the characteristics of the target hardware platform.

The system outputs $\mathbf{y}(k)$ can be highly variable due to stochastics of workload needed to be processed during a sampling interval. This effect can make it difficult to model the relation between the control inputs $\mathbf{u}(k)$ and the outputs $\mathbf{y}(k)$. The amount of workload submitted during each sampling interval depends on (i) number of job releases (ii) execution time of each job (iii) amount of backlog, i.e., job executions that are released in the previous sampling interval but not completed in the same sampling interval. The effect of the number of job releases can be counteracted by choosing an appropriate sampling length. For instance, if a component consists of periodic tasks, we can use the least common multiple of tasks to counteracted for the problem of number of releases. The sampling length should be large enough to accommodate multiple job releases. However, sampling infrequently may result in slow responses to changes. On the other hand, sampling too frequently may impose considerable overhead to the system. Therefore, choosing an appropriate sampling length is of paramount importance that requires careful study and investigations.

We use the Root Mean Square Error (RMSE) for evaluating our identified model parameters[1]. RMSE is scale dependent, therefore we use it for comparing different models representing same data. When comparing two models, the one that has smaller RMSE is better. We also evaluate the variability explained by the model using R^2. In general, models that their $R^2 > 0.8$ are considered to be an acceptable fit to the system [9]. Models that their R^2 is closer to one better explain the identified system. We use sine functions for changing inputs to excite the system and observe the outputs. First we excite the system by only altering α. Then we alter T while keeping α unchanged. Finally, we use all samples for system identification. The model parameters (**A** and **B**) are estimated using least squares. Note that since the periodic servers provide timing isolation, system identification for each component can be done independently. The identified system model will still be applicable after integration with other components because the processor provision to the component will not be affected by other components in the normal mode. Recall that in this section we assume that the system is in the normal mode.

10.3.4 Controller design

Linear-Quadratic Regulators (LQR) let us to trade-off between control speed and over reaction. In contrast to the well-know PID controllers in which the

[1] The model evaluation metrics used in this paper (i.e., RMSE and R^2) are explained in Chapter 2.4.4 of [9].

gain values are directly quantified by designers, the LQR controllers allow designers to focus on the cost of control actions as well as control errors. In general, we prefer unaggressive control actions which provides slow reactions to sudden changes to avoid overreacting to transient stochastics. We define error $\mathbf{e}(k)$ as: $\mathbf{e}(k) = \mathbf{r} - \mathbf{y}(k) = \mathbf{r} - \mathbf{x}(k)$, where $\mathbf{r}$ is the reference value for the output ($\mathbf{r} = [r_1 r_2]^T$). The dynamics of the control system based on $\mathbf{e}(k)$ is as follows:

$$\begin{aligned}\mathbf{e}(k+1) &= \mathbf{r} - \mathbf{A}\mathbf{x}(k) - \mathbf{B}\mathbf{u}(k) \\ &= \mathbf{A}\mathbf{e}(k) - \mathbf{B}\mathbf{u}(k) + (\mathbf{I} - \mathbf{A})\mathbf{r}.\end{aligned}$$

Instead of directly using the model presented in Eq. 10.1, we use the system model based on error for the controller design. In addition to $\mathbf{e}(k)$, we also use integral states: $\mathbf{e_I}(k+1) = \mathbf{e_I}(k) + \mathbf{e}(k)$, where $\forall k \leq 0$ we have $\mathbf{e_I}(k) = 0$. Hence, the augmented state space model is:

$$\begin{bmatrix}\mathbf{e}(k+1) \\ \mathbf{e_I}(k+1)\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{0} \\ \mathbf{I} & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} + \begin{bmatrix}-\mathbf{B} \\ \mathbf{0}\end{bmatrix}\mathbf{u}(k) + \begin{bmatrix}\mathbf{I} - \mathbf{A} \\ \mathbf{0}\end{bmatrix}\mathbf{r}.$$

We use dynamic feedback, that is:

$$\mathbf{u}(k) = -\mathbf{K}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} = -\begin{bmatrix}\mathbf{K}_P & \mathbf{K}_I\end{bmatrix}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix},$$

where $\mathbf{K}_P$ and $\mathbf{K}_I$ are 2×2 matrices. By substituting the control law in the state space model we obtain the following closed-loop system model:

$$\begin{bmatrix}\mathbf{e}(k+1) \\ \mathbf{e_I}(k+1)\end{bmatrix} = \left(\begin{bmatrix}\mathbf{A} & \mathbf{0} \\ \mathbf{I} & \mathbf{I}\end{bmatrix} - \begin{bmatrix}-\mathbf{B} \\ \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{K}_P & \mathbf{K}_I\end{bmatrix}\right) \begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} + \begin{bmatrix}\mathbf{I} - \mathbf{A} \\ \mathbf{0}\end{bmatrix}\mathbf{r}.$$

In LQR control design we are looking for gain values ($\mathbf{K}_P$ and $\mathbf{K}_I$) that minimize the following quadratic cost function:

$$J = \sum_{k=1}^{\infty} [\mathbf{e}(k)\mathbf{e_I}(k)]^T \mathbf{Q}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} + \mathbf{u}(k)^T \mathbf{R}\mathbf{u}(k),$$

where $\mathbf{Q}$ specifies the cost of error and $\mathbf{R}$ quantifies the cost of control action. The responsibility of the system integrator is to choose suitable error and control cost matrices.

10.4 Resource manager

Thus far, we have considered adapting the parameters of a single VC. In this section we consider the whole system. The resource manager has the following responsibilities. (1) Admission control based on the minimum resource requirements; (2) cluster compression, when the average resource requirements can not be met; (3) allocation of the VCs to processors, i.e., mapping the VCs to the VPs; (4) adjusting the parameters of VPs and dealing with overloads. In the rest of this section the above responsibilities are explained in detail. At each sampling point k, the resource manager allocates the suggested parameters by the cluster controllers to the VCs. In this section we focus on a single sampling point. Therefore, for simplicity, we drop sampling time k when referring to the output of the cluster controllers.

Admission. The resource manager creates $\{\Pi_1, \cdots, \Pi_n\}$ for hosting $\{\mathfrak{C}_1, \cdots, \mathfrak{C}_n\}$. The system integrator is allowed to admit components such that the sum of components' minimum bandwidth, specified in the component interfaces, is less the available multiprocessor bandwidth: $\sum_{i\in[1\ldots n]} \bar{\alpha_i} - \sigma_{\alpha_i}/2 \leq M$. In doing so, we can guarantee minimum $\bar{\alpha_i} - \sigma_{\alpha_i}/2$ resource provisioning for $\mathfrak{C}_i$.

Cluster compression. The resource manager performs an allocation based on the operating bandwidths $\bar{\alpha_i}$ specified in the component interfaces. Since the admission is done based on the minimum required bandwidths, it is possible to have $\sum_{i\in[1\ldots n]} \bar{\alpha_i} > M$. In such a case, the resource manager first performs a cluster bandwidth compression, that is, compressing the cluster bandwidths such that the total required bandwidth is less than or equal to M. The VCs will receive partial bandwidths after the compression. α'_i and λ_i denote the compressed bandwidth and the compression factor of Π_i respectively, where $\lambda_i\alpha_i = \alpha'_i$. Our objective is to maximize $\sum_{i=1}^{n} \lambda_i\zeta_i$ when performing the compressions. In doing so, components which have less impact on the total value of the system will be subjected to more compressions. Let $\bar{\alpha_i} - \sigma_{\alpha_i}/2 = \phi_i$ and $\frac{\zeta_i}{\alpha_i} = \Delta_i$. The compression problem is formulated as the following:

$$\textbf{Maximize}: \quad \sum_{i=1}^{n} \alpha'_i\Delta_i, \tag{10.2a}$$

$$\textbf{Subject to}: \quad \alpha_i \geq \alpha'_i \geq \phi_i \qquad \forall i \in [1\ldots n], \tag{10.2b}$$

$$\sum_{i=1}^{n} \alpha'_i \leq M. \tag{10.2c}$$

We use Algorithm 5, which has polynomial time complexity ($O(n^2)$), for solving the cluster compression problem. The algorithm treats VCs in the order of Δ_i. Each VC receives at least ϕ_i. In addition, it receives $\alpha_j - \phi_j$ band-

width if the remaining multiprocessor capacity ($\mathfrak{M}$) is sufficient. Otherwise, the remaining capacity is added to the VC's bandwidth.

Theorem 2. *Algorithm 5 is optimal, i.e., the compression factors produced by this algorithm maximizes the total system value.*

The formal proof of the above theorem is presented in the appendix.

Algorithm 5: Cluster compression algorithm.

Require: $\{\Delta_1, \ldots, \Delta_n\}$ and $\{\phi_1, \ldots, \phi_n\}$.
Ensure: $\{\alpha'_1, \ldots, \alpha'_n\}$.
1: G = $\{\Delta_1, \ldots, \Delta_n\}$;
2: $\forall i, \alpha'_i = \phi_i$;
3: $\mathfrak{M} = M - \sum_i \phi_i$;
4: **while** $G \neq \emptyset$ AND $\mathfrak{M} > 0$ **do**
5: $\quad \Delta_j = \max(G)$;
6: $\quad \alpha'_j = \phi_j + \min(\mathfrak{M}, \alpha_j - \phi_j)$;
7: $\quad G = G - \Delta_j$;
8: $\quad \mathfrak{M} = \mathfrak{M} - \alpha'_j + \phi_j$;
9: **end while**

Allocation. The resource manager performs allocations assuming that the overall required bandwidth is less than or equal to the multiprocessor bandwidth. The allocation algorithm creates at most M VPs for each VC such that all VPs collectively provide B_i units of the processor time to Π_i. We use the partitioned EDF algorithm for scheduling the VPs. The allocation algorithm has three objectives. First of all, the number of VPs should be minimized. This is because when components are split, their inner tasks will migrate between the processors. Hence, the components will require extra bandwidth to compensate for the migration overhead. In addition, we favor balanced allocations that is, fairly distribution of the slack time over all processors. The reason behind preferring balanced distributions is to give the cluster controllers more freedom to adapt the VC bandwidths. When a processor is overloaded, more important components can steal bandwidth from the less important ones that coexist with them on the same processor. Hence, we favor an allocation approach that co-allocates more important components with less important ones. This approach gives more freedom to the more important components to adapt their bandwidth in the overload situations. Hence, the third objective of the allocation algorithm is to achieve a balanced importance distribution. Thus, the

allocation problem formulation is as follows:

$$\textbf{Maximize}: \quad w_1 \left(nM - \sum_{i=1}^{n} \sum_{j=1}^{M} f_{i,j} \right) + w_2 z_2 + w_3 z_3, \tag{10.3a}$$

$$\textbf{Subject to}: \quad z_2 \leq \sum_{i=1}^{n} \rho_{i,j} \qquad \forall j \in [1 \dots M], \tag{10.3b}$$

$$z_3 \leq \sum_{i=1}^{n} \frac{\rho_{i,j}}{\alpha_i} \zeta_i \qquad \forall j \in [1 \dots M], \tag{10.3c}$$

$$\sum_{i=1}^{n} \rho_{i,j} \leq 1 \qquad \forall j \in [1 \dots M], \tag{10.3d}$$

$$\sum_{j=1}^{M} \rho_{i,j} = \alpha_i \qquad \forall i \in [1 \dots n], \tag{10.3e}$$

$$\frac{\rho_{i,j}}{\alpha_i} \leq f_{i,j} \qquad \forall i \in [1 \dots n], \forall j \in [1 \dots M], \tag{10.3f}$$

$$f_{i,j} \in \{0, 1\}, \rho_{i,j} \in \mathbb{Z}_{\geq 0}, \tag{10.3g}$$

where z_2 and z_3 correspond to load balancing, and importance balancing objectives respectively. z_2 represents the maximum load assigned to one processor. While, z_3 represents the maximum importance available on one processor. w_1, w_2 and w_3 are the weights of the three aforementioned objectives. $f_{i,j}$ is equal to one when $\pi_{i,j}$ exists, i.e. $\rho_{i,j} > 0$. Note that the allocation algorithm assumes that $\sum_{i=1}^{n} \alpha_i \leq M$.

The optimization problem formulation presented in Eq. 10.3 is a mixed integer linear programming problem. The complexity of solving this problem is exponential in the number of processors and the number of components. Hence, solving it for large $n \times M$ may become intractable. Therefore, we present an allocation heuristic to partition components in polynomial time. The allocation heuristic is presented in Algorithm 6. Let $v_i = \zeta_i \alpha_i$ denote the value of $\mathfrak{C}_i$. In the algorithm $\{\alpha\}$, $\{v\}$ and $\{\rho\}$ represent the set of VC bandwidths, values and VP bandwidths respectively. First we sort the VCs based on their values. The result bandwidth set is descending in value, i.e., $v_i \geq v_{i+1}$. Then we try to allocate each VC to a processor without splitting it. We use the worst fit allocation, i.e., among all candidate processors that can accommodate the current VC, we choose the one that after allocation it will leave the largest slack time. If the allocation fails, then we split the VC, i.e., we create a number of VPs for the VC. For splitting, we start with a processor that has the largest slack time. We allocate all of the slack time to Π_i and move to a processor with the next largest slack. This process continues until all of the bandwidth of the VC is assigned.

Adjusting VCs. When a cluster controller suggests a new bandwidth and a new

Algorithm 6: Heuristic algorithm for allocating the VCs on processors.

Require: set of cluster bandwidths $\{\alpha\}$ and component values $\{v\}$.
Ensure: matrix of virtual processor bandwidths $\{\rho\}$.
1: sort the active components (Γ) based on their values $\{v\}$
2: **for** $i \in \Gamma$ **do**
3: **if** `WorstFit`(α_i, $\{\rho\}$) = **false then**
4: `Split`(α_i, $\{\rho\}$)
5: **end if**
6: **end for**

period for a VC, the resource manager is responsible to adjust the parameters of the VPs associated with that VC. First of all, the resource manager checks if the suggested values are within the operating range of the component. If the values are beyond the operating region, the resource manager overwrites the suggested values with the boundary of the operating region that is closer to the suggested values. The suggested period is assigned to all of the corresponding VPs. However, the suggested bandwidth is distributed among them. Our goal in distributing the total bandwidth among the VPs is to minimize the number of VPs that are assigned to the VC. This is because in the system identification step the components are identified independently using a minimum number of processors. Hence, we start from the largest slack processor and we allocate its slack bandwidth to the VC. If the VC still needs more bandwidth we move to the second largest slack processor. This process continues until the suggested bandwidth is assigned to the VC.

Dealing with overloads. Assume that the cluster controller of Π_i wants to adapt its bandwidth to α_i^{new}. If the slack time on all processors is not enough to accommodate Π_i with its new bandwidth we have to perform a bandwidth compression. We prefer performing the compression without conducting cluster reallocations. This is because reallocation may force VP migrations which in turn incur overhead costs. In this situation, each VP affected by the compression will receive a portion of its original bandwidth: $\rho'_{i,j} = \lambda_{i,j}\rho_{i,j}$, where $\lambda_{i,j}$ is the compression factor of $\pi_{i,j}$ and $\rho'_{i,j}$ is the overload bandwidth of $\pi_{i,j}$. Our objective is to maximize $\sum_{j=1}^{M}\sum_{i=1}^{n}\lambda_{i,j}\zeta_i$. Let $\frac{\zeta_i}{\rho_{i,j}} = \Delta_{i,j}$. The VP

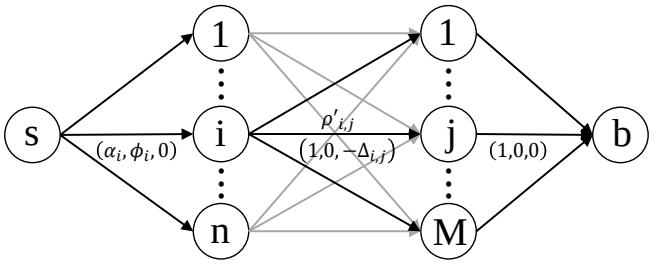


Figure 10.4: The maximum flow formulation of the VP compression algorithm. Node S and b represent the source and the sink nodes respectively. The labels on the edges represent the capacity, demand and cost respectively $(u_{i,j}, d_{i,j}, \kappa_{i,j})$.

bandwidth compression is formulated as the following optimization problem:

$$\textbf{Maximize}: \quad \sum_{j=1}^{M} \sum_{i=1}^{n} \rho'_{i,j} \Delta_{i,j}, \tag{10.4a}$$

$$\textbf{Subject to}: \quad \alpha_i \geq \sum_{j=1}^{M} \rho'_{i,j} \geq \phi_i \qquad \forall i \in [1 \ldots n], \tag{10.4b}$$

$$\sum_{i=1}^{n} \rho'_{i,j} \leq 1 \qquad \forall j \in [1 \ldots M]. \tag{10.4c}$$

This problem can be mapped to the "maximum flow minimum cost with edge demands" problem. Let $H = (V, E)$ be a directed graph with cost $\kappa_{i,j}$, demand $d_{i,j}$ and capacity $u_{i,j}$ associated with every edge $(i, j) \in E$. Figure 10.4 illustrates our model. The edges connecting the source to the n nodes corresponding to the components have a capacity equal to the component bandwidth, a demand equal to the minimum bandwidth of the component and a cost equal to zero. These edges apply the constraint expressed in Eq. 10.4b. The edges connecting the n component nodes to the M processor nodes have a capacity equal to one (maximum bandwidth of a processor), a demand equal to zero and a cost equal to $-\Delta_{i,j}$. The edges connecting the M processor nodes to the sink have a capacity equal to one (to apply the constraint of Eq. 10.4c), a demand and a cost equal to zero. We use the cycle canceling algorithm for solving this problem in polynomial time [10]. Once the problem is solved, the flows of the edges that connect the n component nodes to the M processors will be selected as the compressed VP bandwidths ($\rho'_{i,j}$).

Since the component's bandwidth requirements may change during run-

time and new virtual processors may be created, the initial allocation might become inefficient after some time. Hence, once in a while, the components need to be reallocated. However, in this paper we do not address this problem and we leave it for the future work. We provide some guidelines for selecting the sampling length, operating regions and importance values in the appendix.
Mode change. In our scheme, the cluster parameters are adapted during runtime. This phenomena is referred as mode change in the multi-mode real-time system literature. A potential problem that can happen in mode changes is that, even if the schedulability condition is satisfied before and after the mode change, the system is not necessarily schedulable during the transient mode. Since in this paper we focus only on soft real-time components in which occasional deadline misses can be tolerated, we intentionally neglect this problem. However, if hard real-time components coexist with soft real-time components, the transient overloads can be avoided by introducing a mode change delay similar to [11], and the rest of our method can be directly applied.

10.5 Evaluations

In this section we first evaluate the allocation heuristic. We, then, present a case study consisting of two components. The components are identified using our simulation tool. Thereafter, the performance of the closed-loop system is evaluated both in the simulation tool as well as in our Linux implementation.

10.5.1 Allocation heuristic

We have evaluated the allocation heuristic against the optimal solution. In our evaluations we assumed $M = 4$. We set the total system utilization to two. We changed the number of components from four to 14. For each n we generated 100 random systems. The total utilization was divided among n components using the UUnifast algorithm [12]. Finally, the average achieved objective for each n is reported in Figure 10.6. We have compared five algorithms in the evaluation: (i) optimal load balancing algorithm (ii) optimal importance balancing algorithm (iii) optimal split algorithm (iv) optimal combined objective algorithm (v) our heuristic. We used the CVX solver for solving the optimal algorithms. Each graph in Figure 10.6 illustrates a certain objective achieved by the five algorithms. In all of our evaluations we assumed $w_1 = 1/4(n-1)$, $w_2 = 4/\sum_{i=1}^{n} \alpha_i$ and $w_3 = 4/\sum_{i=1}^{n} \alpha_i\zeta_i$. The figures show that (1) except the optimal combined objective, all other algorithms have poor performance

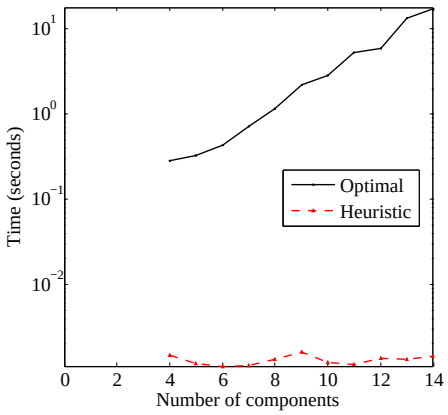


Figure 10.5: Execution time of the allocation algorithms. Note that y-axis is in logarithmic scale.

with respect to some objective, (2) our heuristic outperforms the combined optimal algorithm in the split objective, while the combined optimal algorithm outperforms the heuristic in the rest of the algorithms, (3) our heuristic outperforms all optimal algorithms that consider only one of the three objectives. Figure 10.5 illustrates the execution time of our heuristic allocation against the optimal solution. Each point in the figure is the average of 100 random systems. As shown in the figure, the execution time of the optimal algorithm increases exponentially when increasing the number of components, while the heuristic has approximately constant execution time.

10.5.2 Case study

We have modified the TrueTime [13] simulation tool such that two level hierarchical scheduling is implemented[2]. For system identification and controller design we used our modified TrueTime simulator. We first present two example components. Thereafter, using the examples we explain our modeling approach. Note that a simulation based system identification is only valid if the task execution times are gathered from running tasks on the target hardware platform.

[2]The source code of our modified version is available at: `https://github.com/nimazad/TrueTime-HSF`.

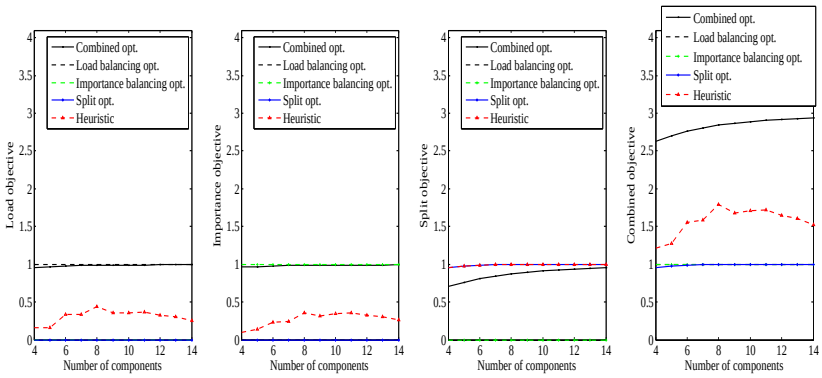
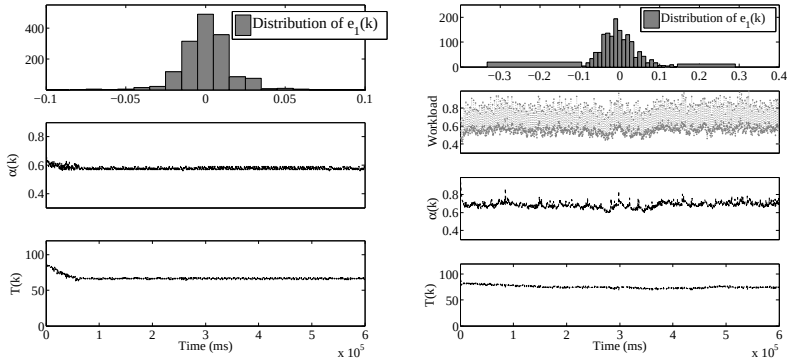


Figure 10.6: Four different objectives achieved by the five allocation algorithms.



(a) Static component. (b) Dynamic component.

Figure 10.7: Distribution of e_1, workload variation and parameter adaptations.

Component 1 (Static component)**.** *Consider a component consisting of three periodic tasks with the following periods* $\{p_1 = 40, p_2 = 50, p_3 = 100\}$ *and the following execution times* $\{c_1 = 12, c_2 = 10, c_3 = 5\}$.

Component 2 (Dynamic component)**.** *Assume a component consisting of three periodic tasks with the following periods* $\{p_1 = 40, p_2 = 50, p_3 = 100\}$. τ_1 *and* τ_2 *are video decoder tasks that decode their input video stream. Therefore, their execution time is highly variable.* τ_3 *is a static task, i.e. its execution time is fixed (*$c_3 = 5$*). Average bandwidth consumed by the video decoder tasks* τ_1 *and* τ_2 *are* 0.29 *and* 0.25 *respectively. This component could, for instance, represent a robotic vision system consisting of two cameras where one real-time task is assigned for decoding video streams captured from each camera. The third task, however, is performing analysis over the decoded streams by executing a predefined number of instructions. The execution time distribution of* τ_1 *and* τ_2*, and the workload distribution of this component is illustrated in Figure 10.8a.*

Let us consider the parameter identification for Component 1, we assumed the task parameters were not available. Instead of the task parameters, the component designer had provided the following interface: $<\bar{\alpha}$=0.65, $\bar{T}$=90, σ_α=0.15, σ_T=100$>$. Therefore, we changed the bandwidth of the VC in the following region $[\bar{\alpha} - \sigma_\alpha/2, \bar{\alpha} + \sigma_\alpha/2]$ while $\forall k\ T(k)$=40. In another experiment we changed $T(k)$ in the following region $[\bar{T} - \sigma_T/2, \bar{T} + \sigma_T/2]$ while keeping $\alpha(k)$=58. The simulation duration was 10 minutes of the tasks' executions. The sampling length was assumed to be 400. We performed parameter estimations over the observed data. The value of matrices **A** and **B** are reported in Table 10.2. Evaluating the result model on the training data (i.e., same data which was used for parameter estimation), we found the following properties: R^2=0.91 RMSE=4.63. In another experiment we altered both $\alpha(k)$ and $T(k)$ at the same time to assess how much our model can explain the new dataset. As a result we had R^2=0.92 and RMSE=5.37. The identified model parameters were used to design a MIMO LQR controller for adapting the VC parameters.

For the dynamic component we used the following interface: $<\bar{\alpha}$=0.72, $\bar{T}$=90, σ_α=0.26, σ_T=50$>$. We followed the same steps (with the same sampling length) as the static component case. The value of matrices **A** and **B** are reported in Table 10.2. Evaluating the result model on the training data we had R^2=0.95, RMSE=4.53. Evaluating the model on a test dataset in which both bandwidth and period were changed simultaneously we had: R^2=0.98, RMSE=1.23. In summary, we conclude that the identified model parameters, for both static and dynamic component, can explain the training dataset as well

as the test dataset to an acceptable extend (see Section 10.3.3 for more details on R^2 and RMSE).

Afterwards, we considered Component 1 which its parameters were already identified. We set **R**=diag$(10, 10)$ and **Q**=diag$(1, 1, 0.1, 0.1)$. The value of the result gain matrix is reported in Table 10.2. We ran the closed-loop system using the obtained gain matrix. The reference values were **r**=$[0.02,\ 1]^T$. Figure 10.7a illustrates the probability distribution of e_1 as well as the bandwidth and period adaptation for the cluster that is serving the static component. Since the task parameters were not changed, the controller stabilized $\alpha(k)$ and $T(k)$ in almost constant values. The average number of deadline misses in this experiment was 1.25 which is very close to the set reference value. The standard deviation of e_2 was 1.66. In another experiment we designed a LQR controller for Component 2 which was identified previously. We set the same **R** and **Q** matrices as the static component example. The value of the result gain matrix is reported in Table 10.2. The reference values were **r**=$[0.06,\ 1]^T$. The VC parameter adaptations are illustrated in Figure 10.7b. The depicted workload variation in the figure is collected independently running the component with the full processor capacity. Since the task execution times were changed, the cluster parameters were also adapted based on the current demand at each time point.

Linux experiments. We have implemented our adaptive framework in the Linux kernel. Inspired by [14], we used kernel loadable modules to implement our scheme[3]. In the rest of this section, we present the results of our Linux evaluations. We used Intel Core i5-3550 processor clocked at 3.3 GHz. Our loadable module can utilize all four cores on this processor. However, for imposing overload situations, we limited the number of available processors to two. The cluster controllers and the resource manager are developed as user space tasks. The controller tasks were attached to a different VC than the component VCs. We considered the two components that we have designed cluster controllers for them using our simulations. The resource manager created two clusters Π_1 and Π_2 for hosting the static component and the dynamic component respectively. Cluster Π_3 was also created for hosting the cluster controller tasks. The bandwidth of Π_3 was equal to 0.05 and it was constant throughout the experiment. All task parameters described in the definition of the components were assigned in milliseconds. We ran the experiment for 10 minutes. Figure 10.8c illustrates the adaptations for this experiment. The observed distribution of e_2 is slightly different than the simulations. The difference is due to the fact that

[3]The source code is available at: `http://nimazad.github.io/FS-CBRTS`.

the simulation does not take into account the overhead of scheduling, adaptation and operating system related interferences. The average observed e_2 was -0.12 for Π_1 and -0.07 for Π_2.

Adaptation overhead. We created a periodic task associated with each cluster which was ran within Π_3. The period of these tasks was equal to 400 (sampling length). The LQR controller as well as the resource manager functionalities are implemented in these tasks. In the above experiment the maximum observed execution time for the controller task of Π_1 and Π_2 were $0.101ms$ and $0.081ms$ respectively. Given that we had two processors available, each adaptive cluster cost approximately 0.01 % of the multiprocessor time. The total adaptation overhead is proportional to the number of adaptive components.

In another set of experiments, to impose overload situations, we created a dummy cluster (Π_4) and assign the following bandwidth to this cluster: $\alpha_4 = 0.42$. With the existence of Π_4, it was not possible to perform reservations based on the worst-case demands anymore. The importance of the clusters were set as follows: $\zeta_1 = 200$, $\zeta_2 = 300$, $\zeta_3 = 2000$ and $\zeta_4 = 3000$. Therefore, the resource manager created two VPs for Π_1 in the beginning of the experiment. Splitting this VC imposes migration overhead to Π_1. We considered three different setups: (1) adaptation was turned off for the both VCs while we assigned $\bar{\alpha}$ and $\bar{T}$ to the VCs; (2) both VCs were adapted; (3) we used the average assigned bandwidth and period observed in the second setup and repeated the experiment with those values. We ran the experiment for 10 minutes. The average observed e_1, e_2 and their standard deviations are reported in Table 10.1. Note that $e_1 < 0$ means that the cluster was idling its bandwidth more than the reference value (r_1) and $e_2 < 0$ means that the number of deadline misses observed at each sampling time was more than one. The results presented in Table 10.1 show that fixed allocation based on the operating points specified in the component interface was inefficient. Note that the operating points are based on the average workloads. When both VCs were adapted, the VP compression was performed 32 times, whereas cluster adaptation was performed 1500 times. Therefore, the additional overhead due to the compressions was insignificant. Since Π_1 has the lowest importance, the compression did not provide extra bandwidth to it. In the adaptive case, the average bandwidth and period assigned for Π_1 and Π_2 were 0.61, 50.65, 0.76 and 74.41 respectively. Hence, in average, there was 0.16 slack bandwidth in the system which permits the admission control to admit new components if required. In the third setup, we used the average bandwidths and periods assigned by the cluster controller in setup 2, and we assigned them as fixed values to the VCs. The average number of deadline misses as well as the standard deviation of the

deadline misses for Π_2 in comparison to the second setup were increased. In addition, in the second setup $3\ \%$ of the VC bandwidth was wasted, whereas in the third setup $8\ \%$ of the VC bandwidth was idled. The results suggest that adaptation helps when the workload is subjected to unpredictable disturbances such as migration overhead and execution time variations.

	Π_1				Π_2			
Exp.	$\bar{e_1}$	σ_{e_1}	$\bar{e_2}$	σ_{e_2}	$\bar{e_1}$	σ_{e_1}	$\bar{e_2}$	σ_{e_2}
1	0.16	0.04	-3.19	1.64	1.00	0.36	-12.94	9.87
2	0.17	0.04	0.31	1.96	0.54	0.07	-0.17	2.17
3	0.62	0.008	0.18	0.91	0.53	0.13	-0.30	3.85

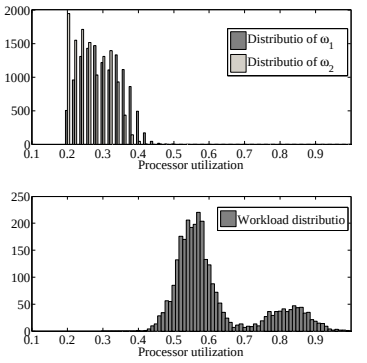
Table 10.1: Mean and standard deviation of e_1 and e_2 for the three setups.

Step response experiment. Figure 10.8b illustrates the response of the static component to a step workload change. The experiment was performed using the same setup as described above. For this experiment the execution time of τ_1 was set to 10 before time $2 \times 10^5 ms$. Afterwards, it was increased to 14. This execution time change caused a $10\ \%$ change in the workload. Note that the reference value for $x_1(k)$ was 0.02. Therefore, the cluster controller provided more bandwidth than the workload. In addition, the cluster controller had to compensate for the workload disturbances such as context switches and scheduling overheads.

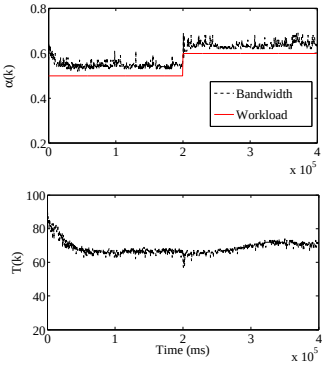
10.6 Related work

Feedback control has found its way in computing systems for helping system designers to deal with uncertainties and dynamicity. For instance, in high-performance computing load is unpredictable and dynamic. A MIMO controller is used to control CPU and memory utilizations in an Apache web server [15]. In [16] a MIMO LQR controller is used to solve a load balancing problem. The controller equalizes the load among different resources to improve response times as well as the throughput.

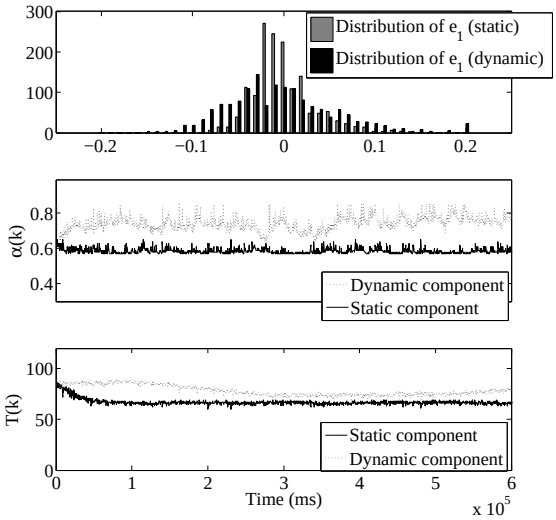
In the context of real-time scheduling, Lu *et al.* proposed a feedback scheduling scheme to cope with unpredictable workloads [17]. In their framework the deadline miss ratio and the system utilization is used as sensors, while the admission control is used as an actuator. The problem of task reweighting under multiprocessor scheduling algorithms is studied in [18] and [19]. In



(a) Execution times distribution of τ_1 and τ_2, and workload distribution of the dynamic component.



(b) Response of the static component to a step workload.



(c) Distribution of e_1, bandwidth adaptation and period adaptation for the two component clusters.

Figure 10.9: Evaluation results (a) running the two sample component (b) for the step workload change.

Static	**A**	$\begin{bmatrix} 0.3711 & -0.5503 \\ 0.1798 & 1.106 \end{bmatrix}$
	B	$\begin{bmatrix} 0.8887 & -0.0413 \\ -0.2952 & 0.0160 \end{bmatrix}$
	K	$\begin{bmatrix} -0.0390 & 0.6150 & -0.0832 & 0.0260 \\ -0.3985 & -1.2376 & -0.0311 & -0.0949 \end{bmatrix}$
Dynamic	**A**	$\begin{bmatrix} 0.7035 & -0.4138 \\ 0.0582 & 1.033 \end{bmatrix}$
	B	$\begin{bmatrix} 0.8443 & -0.0336 \\ -0.2421 & 0.0138 \end{bmatrix}$
	K	$\begin{bmatrix} -0.3506 & 1.1139 & -0.0871 & 0.0090 \\ -0.0850 & -0.7983 & -0.0117 & -0.0992 \end{bmatrix}$

Table 10.2: The value of different matrices corresponding to the case study.

these papers it is assumed that, tasks ask for a new processor utilization during run-time. A number of reweighting rules for partitioned and global scheduling algorithms are presented. In [20] task reweighting is combined with feedback loops that estimate the weight of the next job. In distributed real-time systems, utilization control is performed through rate adaptation to provide quality of service guarantees [21]. In [22] service levels are adapted based on monitoring the number of deadline misses and the processor utilizations. Utilization control is coupled with processor frequency adjustment in [23] and [24]. Targeting end-to-end task models, DEUCON [25] employs a decentralized approach in which task rates (periods) are adapted using MIMO model predictive controllers. The control objective is to minimize the difference between the utilization set points and current utilizations. The main difference of our paper with the aforementioned works is the following. Since we consider component-based systems in which a component is comprised of a set of tasks, a reservation-based scheduling policy is needed to isolate the timing behavior of the components in run-time. While this separation of run-time behavior for components is not supported by the above frameworks.

Adaptive reservation schemes, first introduced in [5], are powerful approaches for controlling the amount of processor allocated to individual tasks that demonstrate dynamic processor requirement. The mathematical model of

a such scheme using Constant Bandwidth Servers (CBS) is derived in [26]. PI controllers are used for controlling the bandwidth of CBS. In [27] stochastic controllers are used for the same purpose. Regarding adaptive reservations in which multiple parameters are adapted, in [28] both periods and budgets of the CBS are adapted. This framework targets legacy tasks which do not communicate with the scheduler. Two different components are used (i) period detector (ii) budget estimator. One centralized controller is used for adapting the periods and the budgets. In the context of the ACTORS project [29], a cascade controller is used on top of CPU reservations for adapting their bandwidths. Our work is different from the above reservation-based approaches in the following main aspects. (i) Except ACTORS, all aforementioned frameworks target single processors. While we target multiprocessors. In contrast to ACTORS, our framework allows spreading VCs (components) over multiple physical processors. This feature allows running components which their utilization is more than one, i.e., component that can not be executed using only one processor. (ii) In the above schemes (including ACTORS) the distance between the task finishing time and its corresponding CBS deadline is used as the sensor. However, we consider a more general component model in which multiple tasks may be in a single component. In our model, the intra-component scheduler coordinates the execution sequence of the tasks inside a component. Hence, the control input used by the above frameworks is not applicable to our model. (iii) Except [28], the other frameworks only adapt reservation budgets, while we adapt the period and budget simultaneity. Our framework is different from [28] in aspect (i) and (ii). In addition, in contrast to [28], we consider software components that are developed using API functions that inform the scheduler when the tasks start executing, finish execution and wait until their next period.

Finally, we proposed adaptive reservation schemes for hierarchical real-time systems in [30, 31]. In our previous works we have addressed single processors while in this paper we consider multiprocessor platforms. Considering multiprocessors resulted in introducing a mechanism for distributing components over the processors. Moreover, we adapt both period and budget of the reservations using MIMO controllers, whereas we only investigated adapting the budgets in the aforementioned publications. In addition, in this paper, we solved the problem of bandwidth distribution in overload situations optimally.

10.7 Conclusions

We proposed a feedback scheduling framework for component-based soft real-time systems. We targeted software components consisting of multiple real-time tasks which exhibit significant processor demand variation during run-time. Our framework uses processor reservations for providing processor time to the components. A component may be distributed over several processors. Hence, the intra-component tasks are scheduled using a global multiprocessor scheduler. First we showed that it is important to adapt both period and bandwidth of the reservations. We, then, used a case study and evaluated our MIMO LQR controllers in the TrueTime simulation tool. Finally, we implemented our framework in the Linux kernel and evaluated the case study in practice. The evaluations show that our framework can efficiently adapt the reservations to deal with the workload disturbances.

In the future, we will investigate the problem of reallocating components systematically by introducing a new metric to understand when it is necessary to perform reallocations. We are also contemplating the elimination of the system identification step by utilizing an adaptive control scheme that can develop the plant model during run-time.

Appendix

Guidelines for parameter selections. Our adaptive framework spares engineers from conducting the WCET analysis. However, the component developers and the system integrator have to carefully design some parameters for maximizing the performance of the framework. Here we provide some guidelines for some of the parameters.

The system integrators have to study the dynamics of the components using different sampling lengths. The choice of the sampling length affects the accuracy of the identified models. This is because, the observed system dynamics are different for different sampling lengths. The larger the sampling length, the smoother the output. However, a too large samplings length results in slow reactions to the changes.

The operating regions of the components are provided by the component developers. These parameters have to be selected based on the task parameters as well as experimental studies. For instance, the operating point of the component period depends on the period of the tasks within the component. While, the bandwidth can be extracted by running the component and profiling

its processor usage. Throughout the system identification step, the observed state space values have to be stored. In doing so, the system integrators can plot the feasible combinations of the state space values. The desired set points are, then, selected from the feasible set. For instance, the system integrator may choose a very small r_2. However, the price for having a too small r_2 often is to select a large r_1 which essentially means that we have to waste some bandwidth in order to achieve a very small number of deadline misses.

Our model of the component importance is quiet flexible. Here we consider two type of systems. (1) Systems in which the contribution of each component to the overall value of the system is clear for the integrator; (2) systems in which the relative importance is relevant, e.g., $\mathfrak{C}_i$ is always prioritized over $\mathfrak{C}_j$ in all overload conditions. For type (1), it is easy to assign the importance values. For instance if $\mathfrak{C}_1$ contributes 10 % to the total system value, we can assign $\zeta_1 = 10$ provided that $\sum_{i=1}^{n} \zeta_i = 100$. For type (2), the system integrator should first sort the components based on their desired priority at run-time. Assume that components are sorted based on their desired run-time overload priority, i.e., for $i \in [1, \ldots, n]$, $\mathfrak{C}_i$ has to be prioritized to $\mathfrak{C}_{i+1}$. The process of importance assignment starts from $\mathfrak{C}_n$. The designer assigns a small number to this component. $\mathfrak{C}_{n-1}$, then, we will have the following condition:

$$\zeta_{n-1} > \zeta_n \frac{\bar{\alpha}_{n-1} + \sigma_{\alpha_{n-1}}}{\bar{\alpha}_n - \sigma_{\alpha_n}}.$$

This condition is because the compression algorithm takes the size of the components into account when compressing the components. In this approach, $\mathfrak{C}_i$ will have $n - i$ conditions for its importance. To summarize we have:

$$\zeta_i > \max_{i+1<j<n} \left(\zeta_j \frac{\bar{\alpha}_i + \sigma_{\alpha_i}}{\bar{\alpha}_j - \sigma_{\alpha_j}} \right).$$

Proof of Theorem 2:

Proof. We prove using the Lagrangian duality [32]. The Lagrangian is

$$L = \sum\nolimits_i \delta_i \Delta_i - \theta(\sum\nolimits_i \delta_i - M) - \sum\nolimits_i \overline{\chi}_i(\delta_i - \alpha_i) + \sum\nolimits_i \underline{\chi}_i(\delta_i - \phi_i)$$

where $\theta, \overline{\chi}_i, \underline{\chi}_i$ are Lagrange multipliers. The Karush-Kuhn-Tucker (KKT)

conditions are:

$$\Delta_i - \theta - \overline{\chi}_i + \underline{\chi}_i = 0, \tag{10.5}$$

$$\overline{\chi}_i(\delta_i - \alpha_i) = 0,\ \overline{\chi}_i \geq 0, \tag{10.6}$$

$$\underline{\chi}_i(\delta_i - \phi_i) = 0,\ \underline{\chi}_i \geq 0, \tag{10.7}$$

$$\sum\nolimits_i \delta_i = M,\ \theta \geq 0. \tag{10.8}$$

In the following, we prove that the solution by Algorithm 5 satisfies the KKT conditions. After Algorithm 5, the set G is partitioned into three subsets as follows: $S_1 = \{i|\delta_{i\in S_1} = \phi_i\}, S_2 = \{i|\delta_{i\in S_2} \in (\phi_i, \alpha_i)\}, S_3 = \{i|\delta_{i\in S_3} = \alpha_i\}$. Set $\Delta_{left} = \max\{\Delta_{i\in S_1}\}, \Delta_{right} = \min\{\Delta_{i\in S_3}\}$. From the searching process by the ***while*** loop in Algorithm 5, we know that: $(i)\Delta_{left} < \Delta_{right}, (ii)\Delta_{i\in S_1} \leq \Delta_{left}, (iii)\Delta_{left} < \Delta_{i\in S_2} < \Delta_{right}, (iv)\Delta_{i\in S_3} \geq \Delta_{right}$. For $i \in S_2$, $\overline{\chi}_i = \underline{\chi}_i = 0$; so, by Eq. (10.5), we have: $\theta = \Delta_i$, $\Delta_{left} < \theta < \Delta_{right}$. For $i \in S_1$, $\overline{\chi}_i = 0$; so, by Eq. (10.5), we have:

$$\underline{\chi}_i = \theta - \Delta_i > \theta - \Delta_{left} \geq 0.$$

For $i \in S_3$, $\underline{\chi}_i = 0$; so, by Eq. (10.5), we have:

$$\overline{\chi}_i = \Delta_i - \theta > \Delta_{right} - \theta \geq 0.$$

Therefore, all KKT conditions are satisfied and the solution is optimal. □

References

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

[2] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 26–35, May 2001.

[3] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems, (ECRTS'08)*, pages 181–190, July 2008.

[4] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*, pages 249–258, December 2010.

[5] L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 70–77, December 1999.

[6] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, January 2009.

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

[8] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, May 2001.

[9] J. L. Hellerstein, Y. Diao, S. Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[10] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[11] L. Santinelli, G. Buttazzo, and E. Bini. Multi-moded resource reservations. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, pages 37–46, April 2011.

[12] E. Bini and G. Buttazzo. Biasing effects in schedulability measures. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 196–203, June 2004.

[13] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzen. How does control timing affect performance? analysis and simulation of timing using Jitterbug and TrueTime. *Control Systems, IEEE*, 23(3):16–30, June 2003.

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

[15] N. Gandhi, D.M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. MIMO control of an apache web server: modeling and controller design. In *Proceedings of the American Control Conference (ACC'02)*, volume 6, pages 4922–4927, 2002.

[16] Y. Diao, J. L Hellerstein, A. J Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *Proceedings of the 2004 American Control Conference (ACC'04)*, volume 3, pages 2045–2050, 2004.

[17] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23:85–126, 2002.

[18] A. Block, J. H. Anderson, and U. C. Devi. Task reweighting under global scheduling on multiprocessors. *Real-Time Systems*, 39(1-3):123–167, 2008.

[19] A. Block, J. H. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 429–435, 2005.

[20] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 23–33, July 2008.

[21] J. Yao, X. Liu, X. Chen, X. Wang, and J. Li. Online decentralized adaptive optimal controller design of cpu utilization for distributed real-time embedded systems. In *Proceedings of the American Control Conference (ACC'10)*, pages 283–288, June 2010.

[22] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 59–70, December 2001.

[23] X. Wang, X. Fu, X. Liu, and Z. Gu. PAUC: Power-aware utilization control in distributed real-time systems. *IEEE Transactions on Industrial Informatics*, 6(3):302–315, Aug 2010.

[24] X. Chen, X. W. Chang, and X. Liu. SyRaFa: Synchronous rate and frequency adjustment for utilization control in distributed real-time embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1052–1061, May 2013.

[25] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. DEUCON: Decentralized end-to-end utilization control for distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):996–1009, July 2007.

[26] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'2)*, pages 71–80, December 2002.

[27] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 238–245, May 2004.

[28] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli. Adaptive real-time scheduling for legacy multimedia applications. *ACM Transactions on Embedded Computing Systems*, 11(4):86:1–86:23, January 2013.

[29] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino. Resource management on multicore systems: The ACTORS approach. *Micro, IEEE*, 31(3):72–81, May-June 2011.

[30] N. Khalilzad, M. Behnam, and T. Nolte. Multi-level adaptive hierarchical scheduling framework for composing real-time systems. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages 320–329, August 2013.

[31] N. Khalilzad, M. Behnam, G. Spampinato, and T. Nolte. Bandwidth adaptation in hierarchical scheduling using fuzzy controllers. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 148–157, June 2012.

[32] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

Chapter 11

Paper D: Adaptive Multi-Resource End-to-End Reservations for Component-Based Distributed Real-Time Systems

Nima Khalilzad, Mohammad Ashjaei, Luis Almeida, Moris Behnam and Thomas Nolte.
In Proceedings of the 13th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'15), October, 2015.

Abstract

Complexity in the real-time embedded software domain has been growing rapidly. The component-based software development approach facilitates the development process of such software systems by dividing a complex system into a number of simpler components. Resource reservation techniques have been widely used for providing resources to real-time software components. In this paper we target real-time components operating on a distributed infrastructure. Furthermore, we target a class of software components which demonstrate dynamic resource consumption behavior. A prime example of such components is a multimedia software component. In the paper we present a framework supporting multi-resource end-to-end resource reservations. We reserve resource bandwidths on both processor resources as well as on the network resources. The proposed framework utilizes a Multiple Input Multiple Output (MIMO) controller which adjusts the sizes of reservations tracking the dynamic resource demands of the software components. Finally, we present a case study using a multimedia component to demonstrate the performance and efficiency of our framework.

11.1 Introduction

Complex distributed systems are currently disseminated over a large range of application domains, particularly inherent in cyber-physical/embedded systems. These systems are typically subject to several non-functional constraints, stretching from resource limitations to timeliness, including safety and other constraints. Taming complexity in their design is particularly important to ensure a swift development and a correct result.

To this end, component-based software engineering is particularly well suited, providing a modular approach that allows independent development of system components which are integrated at the final stage of the development. Combining such an approach with resource reservation techniques, components can be encapsulated in reservations that match their requirements [1, 2, 3]. Moreover, reservations enforce mutual isolation, particularly temporal isolation. Thus, components that run inside adequate reservations can be proved correct independently of other components possibly running in the system.

However, beyond correctness, current design trends aim at resource efficiency, reducing the component footprint over the set of needed resources, particularly computing and communication resources, and changing the resource requirements at run time according to instantaneous needs. In this scope, dynamic reservations can provide a suitable solution to guarantee a continued adequate match between the varying resource requirements and the provided reservations. Dynamic reservation schemes are of particular interest in the multimedia applications in which the instantaneous resource requirements are highly dynamic. Unfortunately, dynamic reservations have been mostly studied for single resource systems (e.g. [4, 5]). Distributed dynamic reservations taken in a holistic way, particularly considering processor and network resources in an integrated fashion, have not received much attention. These, however, are necessary for common multimedia systems ranging from area surveillance to process monitoring and even safety driver assistance.

This paper aims at contributing with a solution to such a problem, making use of dynamic resource reservations on processor and network resources, coupled by dynamic component requirements. We provide a solution which allows for matching of dynamic requirements with the resource reservations, reducing typical overprovisioning of static designs and thus resource usage. In turn, since more resource capacity will be potentially available, the system service will also be improved, e.g. allowing for serving more components and/or with more quality.

In this paper, we consider a component model in which each component

contains multiple tasks spread over a distributed system that communicate through the network using messages. We consider an end-to-end soft real-time model. We reserve a fraction of the processor as well as network resources needed by the component to satisfy its timing requirements. We refer to this reservation scheme as *multi-resource end-to-end reservations*. Furthermore, we continually monitor the actual resource usages of the components, and we adjust the reservation sizes to match their instantaneous resource requirements. In particular, we present the following contributions in this paper:

- we present a new framework featuring multi-resource end-to-end reservations in which the reservation sizes are adaptive;
- we design a Multiple Input Multiple Output (MIMO) Linear Quadratic Regulator (LQR), which adjusts the reservation sizes during run-time;
- we present an on-line system identification method based on Recursive Least Squares (RLS), which identifies the dynamics of the resource requirements;
- we present a surveillance case study in which three processor nodes are used, connected through an Ethernet switch.

The rest of the paper is organized as follows. In the next section we review the related work regarding reservation techniques on processors, networks and distributed systems. Section 11.3 presents our modeling approach with respect to the resources and components. The architecture of our framework is presented in Section 11.4. We present the control design as well as the system identification method in Section 11.5. We present a surveillance component case study in Section 11.6. Finally, we conclude the paper in Section 11.7 where we also describe the future directions.

11.2 Related work

In the following, we review the reservation techniques inherent in three different areas, processor resources, network resources, and end-to-end resources in distributed systems. We also review two groups of works: (i) static reservations, and (ii) dynamic reservations. From a modeling perspective, several resource models have been proposed for modeling resource reservation techniques. For instance, the Periodic Resource (PR) model [1] uses a period and a budget for characterizing a reservation. A reservation is guaranteed to receive

a specific budget during each time interval equal to the period. The budget is reduced while the resource is assigned to a particular component, and it is replenished at the start of the period.

Resource reservations on processors. A number of resource reservation models are realized on the processor resources. For instance, the Constant Bandwidth Servers (CBS) [6] are implemented in the Linux kernel [7], or the PR model is implemented in VxWorks [8]. When the tasks have dynamic resource demands, it is desirable to adapt the reservation parameters to deal with the resource demand changes. Adaptive CBS is promoted in the AQuoSA project [4] for dynamic tasks such as video decoders. The ACTORS projects [9] uses adaptive CBS on multiprocessor platforms. In [5], the budget of periodic servers are adapted tracking the processor demand of the components. In this work the model is hierarchical, i.e., the periodic servers may contain multiple tasks as well as multiple child periodic servers.

Resource reservations on network. The same modeling concepts as in processors have been applied on the network resources. A general category of the resource management in network is traffic shapers [10]. The purpose of these shapers is to limit the amount of traffic that a node submits to the network in a given time interval. Similar to the techniques used by processor servers, the traffic shapers use methods based on capacity which is replenished with different policies, e.g. credit-based shaping in Ethernet AVB [11]. Moreover, some real-time Ethernet protocols enforce a cyclic-based transmission and reserve windows for different classes of traffic (e.g., Ethernet POWERLINK [12], FTT-SE [13] and HaRTES [14]). Also, a hierarchical server model [15] is proposed for the Ethernet switches in the context of the FTT-SE protocol to reserve a portion of bandwidth for different traffic types, hence providing temporal isolation among them. An online QoS management [16] is proposed in the context of a multimedia real-time application, which adapts the video compression parameters and the network bandwidth reservations to provide the best possible QoS to the streams. Our end-to-end reservation framework can use the above network technologies for reserving the network resources.

Registering resource reservations on network. In order to reserve resources for streams in the network several protocols have been proposed, where they use similar concepts. For instance, Stream Reservation Protocol (SRP) [17] defines a set of procedures to reserve network resources for the specific traffic streams, which are crossing through an Ethernet Audio Video Bridging (AVB) network. The SRP protocol forces the traffic to be registered on the AVB switches through its path, before being transmitted. Furthermore, a Resource ReSerVation Protocol (RSVP) [18] was proposed to reserve resources for a

stream with a specific Quality of Service (QoS) requirement. This protocol operates using an admission control, which checks whether there are enough resources to supply the requested QoS requirement. In both protocols, the mechanism performs by sending a request through the network and checking in each node the availability of resources. These protocols provide a support for communicating new reservations in adaptive reservation schemes such as ours.

Resource reservations in distributed systems. Few authors have addressed the end-to-end reservation of resources for distributed systems, including processor and network resources. A distributed kernel framework with a resource manager in each node has been designed and implemented to provide an end-to-end timeliness guarantee [19]. Also, a resource management system, called D-RES [20], has been developed to handle shared resources among multiple applications in distributed systems. A very close work related to ours is the one presented in [21] in which a pipeline task is considered. Tasks may use one of the resources available in the system to carry on their computations. Adaptive CBS is used to track the resource demand of the tasks. In addition, a general model, called Q-RAM [22], has been developed to manage the resources shared among multiple applications. The applications in this framework have different operation levels with different qualities depending on the available resources. However, they have to satisfy their needs such as timeliness, reliability and data quality. The model allocates the resources to the applications considering that the overall system utility becomes maximum while the applications meet their minimum needs. This model has been extended in [23] for the systems with rapidly changing resource usage.

The main difference of our work with [21] and [22] is that we consider adaptation for components which may in turn be composed of multiple tasks. The existence of multiple tasks inside one component makes the system dynamics model in those works inapplicable to our setting. Therefore, we use an on-line model identification method for estimating the parameters of the model. Besides, we perform the adaptations in an integrated fashion for all resources of the component using MIMO controllers. This is because the MIMO control approach allows us to simultaneously adapt the bandwidths of all reservations considering the possible coupling among them. Finally, in our framework we explicitly consider network resources, and we present the result of our case study in which we used a common network technology.

Adaptive distributed systems. Feedback scheduling techniques have been used in the context of distributed systems. In particular a line of work in this context focuses on keeping the utilization of the processors below their schedu-

lability threshold. For instance Stankovic *et al.* have studied this problem for independent tasks [24]. On the other hand, the following two frameworks are proposed for end-to-end task models: EUCON [25] and DEUCON [26]. While EUCON uses a centralized controller, DEUCON employs a decentralized approach in which task rates (periods) are adapted using model predictive controllers. The main difference of our paper with the aforementioned works is the following. Since we consider component-based systems in which a component is comprised of a set of end-to-end tasks, a reservation-based scheduling policy is needed to isolate the timing behavior of the components in run-time. This separation of run-time behavior for components is not supported by the above frameworks. Besides, we explicitly consider network resources in our framework, while the above frameworks only focus on the processor resources.

11.3 Model

We assume a Distributed Resource (DR) infrastructure with M resources. We use r_h to denote the h^{th} resource in the system. The set of resources is denoted by $\mathcal{R} = \{r_1, \ldots, r_M\}$. We consider two types of resources: (i) network resources; (ii) processor resources. We assume that N real-time components are placed on the DR infrastructure. Each component uses a subset of all resources.

Component and task model. We assume that a real-time component $\mathcal{C}^{(\iota)}$ is composed of a set of tasks: $\mathcal{C}^{(\iota)} = \{\tau_1^{(\iota)}, \tau_2^{(\iota)}, \ldots\}$, where $\tau_i^{(\iota)}$ represent the i^{th} task of $\mathcal{C}^{(\iota)}$. We assume an end-to-end sporadic task model in which a task $\tau_i^{(\iota)}$ requires a subset of available resources (processor and/or network) for completing its execution. $\tau_i^{(\iota)}$ begins its execution on a processor resource (source processor), and it finishes its execution on a processor resource (destination processor). The set of all resources consumed by $\tau_i^{(\iota)}$ is denoted using $\mathcal{R}_i^{(\iota)}$, where $\mathcal{R}_i^{(\iota)} \subset \mathcal{R}$. $\tau_i^{(\iota)}$ is characterized with a minimum inter-arrival time $p_i^{(\iota)}$ and an end-to-end soft deadline $d_i^{(\iota)}$. $p_i^{(\iota)}$ refers to the release of the task on the source processor, while $d_i^{(\iota)}$ indicates its relative deadline on the destination processor. $\tau_i^{(\iota)}$ is composed of a set of subtasks each consuming a resource. We use $\tau_{i,j}^{(\iota)}$ to refer to the j^{th} subtask of $\tau_i^{(\iota)}$. Note that we also use the term subtask for the chunks of the end-to-end tasks that consume the network resources (network subtasks). In this model, a message is a set of network subtasks. $\tau_{i,j}^{(\iota)}$ is characterized with a Resource Consumption Time (RCT) $c_{i,j}^{(\iota)}$ which indicates execution/transmission time of the subtask. We assume that the RCTs (i) are not known a priori to run-time; (ii) are changing during run-time. The quality

of service experienced by tasks depends on the number of deadline violations. The objective of our adaptive framework is to minimize the number of deadline violations without significant resource overprovisioning. To this end, we use a controller module in our framework to track the resource requirements of the components and adjust the reservation budgets accordingly.

Virtual DR. Recall that each component is assigned to a subset of available resources denoted by $\mathcal{R}^{(\iota)}$. We use resource reservation polices for partitioning the bandwidth of the resources. For all $r_h \in \mathcal{R}^{(\iota)}$, $\mathcal{C}^{(\iota)}$ receives a fraction of the bandwidth of r_h. We refer to the subset of partially available resources for a component as a *virtual DR*, and we use $\Gamma^{(\iota)}$ for denoting it. The specification of the virtual DR allocated to $\mathcal{C}^{(\iota)}$ is denoted using:

$$\Gamma^{(\iota)} = \left\langle \Pi^{(\iota)}, \{\Theta_1^{(\iota)}, \Theta_2^{(\iota)}, \ldots, \Theta_{M^{(\iota)}}^{(\iota)}\} \right\rangle,$$

where $\Pi^{(\iota)}$ is the period of the resource reservations, $\Theta_h^{(\iota)}$ denotes the budget of r_h reserved for $\mathcal{C}^{(\iota)}$ and $M^{(\iota)}$ is the number of resources used by $\mathcal{C}^{(\iota)}$. Without loss of generality, to avoid a conflict in resource indexing, we consider $M^{(\iota)} = M$. The above abstraction means that $\mathcal{C}^{(\iota)}$ is guaranteed to receive at least $\Theta_h^{(\iota)}$ time units of resource r_h every $\Pi^{(\iota)}$. The reserved bandwidth of $\Gamma^{(\iota)}$ on resource r_h is defined as:

$$\alpha_h^{(\iota)} = \frac{\Theta_h^{(\iota)}}{\Pi^{(\iota)}}.$$

Note that such a periodic resource abstraction model is supported by several processor and network scheduling schemes. For instance, on the processor resource we can use Constant Bandwidth Servers (CBS) [6] or Periodic Servers (PS) [1]. While for the network resource we can use a hierarchical server model [15] or the periodic model presented for the FTT-SE [13] and HaRTES [27] architectures.

11.4 Framework

In this section we present the scheduling scheme of our framework. We also provide an overview of our adaptation mechanism.

Resource scheduler. We assume a resource scheduler per each physical resource. The schedulers (i) enforce resource reservations; (ii) schedule subtasks; (iii) and communicate with the respective controller modules to inform them about the state of the reservations. Although resources are scheduled locally, our end-to-end reservation scheme guarantees a predictable system wide

resource provisioning for the end-to-end tasks. We use a hierarchical scheduling scheme in which scheduling is performed at two levels. In the higher level the resource scheduler schedules the reservations. Within each reservation, however, it is the responsibility of the subtask-scheduler to schedule different subtasks belonging to that reservation.

Controller module. We use a *controller* module per component which is responsible to adjust the reserved bandwidths for the component. The controller monitors the actual resource usages of the component on its allocated resources. This monitoring is performed through communications with the local schedulers. Once the controller decides a new set of reservation bandwidths, it communicates the new requirements to the local resource schedulers. The controller samples and adapts the system periodically. The sampling time is denoted by k. The time distance between two consecutive samples is referred as a sampling interval. We place the controller on a processor resource, and we reserve a portion of that processor resource for the controller executions as well as on the network resources that support the communication with the resource schedulers. These control reservations are static and attached to each component.

Since the reservation sizes are adapted during run-time, resources may temporarily become overloaded. In other words, the overall reserved bandwidth on a resource may pass beyond its schedulability threshold. In such a situation, the components can be prioritized based on their importance in their contribution to the overall goal of the system. If a component's bandwidth gets compressed on one of its resources, then it may be efficient to compress its bandwidth on the remaining component resources as well. Therefore, the framework should follow a protocol in overload situations. The overload management mechanism is out of the scope of this paper, and we leave incorporating an overload manager module to our framework for the future work. In this paper we intend to answer the following question:

"Given a component with requirements that vary dynamically on different resources, how can we define dynamic reservations that track the evolving requirements while satisfying resource constraint?"

Example. In the following we present an example for elaborating our framework. In our example we assume a distributed system consisting of six resources $\{r_1, \ldots, r_6\}$. r_1, r_4 and r_5 are processor resources while r_2, r_3 and r_6 are network resources. We assume that a surveillance component has been placed on this distributed system. Two cameras are attached to r_1 and r_5. We have two end-to-end tasks: $\tau_1^{(1)}$ and $\tau_2^{(1)}$. The video frames are preprocessed in their source processors by the first subtasks of the two tasks. Thereafter,

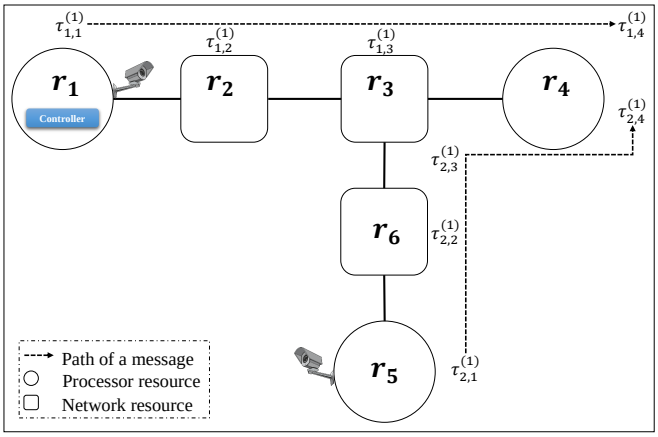


Figure 11.1: Surveillance component example. r_1 and r_5 have two cameras attached. The lines connecting the resources represent logical connections. The network resources are explicitly visualized as boxes.

the video frames are sent to r_4 which hosts the final subtasks. We model this surveillance system as a component placed on the distributed infrastructure:

$$\mathcal{C}^{(1)} = \left\{\tau_1^{(1)}, \tau_2^{(1)}\right\}.$$

$\tau_1^{(1)}$ is composed of four subtasks: $\tau_{1,1}^{(1)}$,$\tau_{1,2}^{(1)}$,$\tau_{1,3}^{(1)}$ and $\tau_{1,4}^{(1)}$. $\tau_{1,1}^{(1)}$ represents the video encoder subtask placed on r_1, while $\tau_{1,4}^{(1)}$ is the decoder and display subtask on r_4. $\tau_{1,2}^{(1)}$ and $\tau_{1,3}^{(1)}$ represent the message that consumes r_2 and r_3 on its path from r_1 to r_4. Similarly $\tau_2^{(1)}$ is composed of four subtasks: $\tau_{2,1}^{(1)}$,$\tau_{2,2}^{(1)}$,$\tau_{2,3}^{(1)}$ and $\tau_{2,4}^{(1)}$ consuming r_5, r_6, r_3 and r_4. The resource reservations of $\mathcal{C}^{(1)}$ is represented using the following interface:

$$\Gamma^{(1)} = \left\langle \Pi^{(1)}, \{\Theta_1^{(1)}, \Theta_2^{(1)}, \Theta_3^{(1)}, \Theta_4^{(1)}, \Theta_5^{(1)}, \Theta_6^{(1)}\} \right\rangle.$$

Note that $\mathcal{C}^{(1)}$ may be sharing the resources with other components.

11.5 Component controller module

The objective of our framework is to satisfy the quality of service requirements of the tasks within the components. We consider meeting the end-to-end

deadlines as a measure of the quality of service satisfaction. Our secondary objective is to allocate the resources efficiently. That is, the deadlines should be respected without significant resource overallocations. To this end, we design a feedback based controller module in this section.

We use a control theoretic approach, similar to [28], for designing the component controller module. In such an approach, to control the plant, we define controlled variables, i.e., measurable variables that indicate the state of the plant, and control inputs, i.e., variables that allow us to manipulate the plant. Note that our plant is the set of resources that are used by the component. The component controller samples and adapts the plant periodically. In our framework, we consider one component controller per component. In the rest of this section, for notational convenience, we drop the component index (ι) when referring to the parameters associated with $\mathcal{C}^{(\iota)}$.

Let $\alpha_h(k)$ indicate the reserved bandwidth on r_h during sampling point $k-1$ and k. The component utilizes a portion of the reserved bandwidth. Let $\alpha'_h(k)$ denote the amount of consumed bandwidth during sampling point $k-1$ and k, where $0 \leq \alpha'(k) \leq \alpha(k)$. The amount of wasted bandwidth on r_h is: $y_h(k) = \alpha_h(k) - \alpha'_h(k)$, where we consider $y_h(k)$ the h^{th} output of our control system. Hence, the vector of system outputs is:

$$\mathbf{y}(k) = [y_1(k) \dots y_M(k)]^T.$$

We selected the assigned bandwidth as our control inputs: $u_h(k) = \alpha_h(k) - \bar{\alpha}_h$, where $\bar{\alpha}_h$ is the operating bandwidth of the component on r_h. This parameter is provided by component developers. For instance it can be an estimate of the average required bandwidth on r_h. Note that this parameter does not need to be exact since we use a feedback loop to adjust the bandwidths. The vector of the control inputs is defined as follows:

$$\mathbf{u}(k) = [u_1(k) \dots u_M(k)]^T.$$

At each sampling time, the goal of the component controller is to find a control input vector $\mathbf{u}(k)$ such that $\mathbf{y}(k) = \mathbf{y}^{ref}$, where $\mathbf{y}^{ref} = [y_1^{ref} \dots y_{N^{(\iota)}}^{ref}]$, and y_h^{ref} is the desired value of $y_h(k)$. We assign this parameter to a small positive value to provide some slack bandwidth for the component. This is because $y_h(k)$ becomes saturated at zero, thus when $y_h(k) = 0$ it is not possible to infer whether the component requires more bandwidth or it is satisfied with the current bandwidth. To reach the goal of the component controller, we use an on-line system identification method along with an optimal controller. Figure 11.2 illustrates the architecture of our control system. In the following we explain the details of the system identification method as well as the

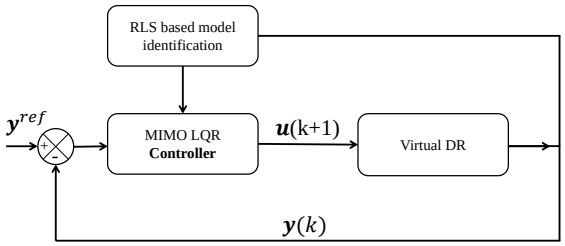


Figure 11.2: The architecture of the control system.

controller design. Note that, although we do not monitor the end-to-end response times, we can indirectly control the response times using our control inputs $\mathbf{u}(k)$. This is because by manipulating the bandwidth of a resource we can affect the response time of subtasks on that particular resource. Since we manipulate the bandwidth of all resources serving the end-to-end tasks, we can affect the end-to-end response times.

We need to model the relation between the control inputs $\mathbf{u}(k)$ and system output $\mathbf{y}(k)$. We use the following auto-regressive MIMO model for modeling this relation:

$$\mathbf{y}(k+1) = \mathbf{A}\mathbf{y}(k) + \mathbf{B}\mathbf{u}(k) + \boldsymbol{\varepsilon}(k+1), \tag{11.1}$$

where $\mathbf{A}$ and $\mathbf{B}$ are $M \times M$ matrices, $\mathbf{u}(k)$ is the control input, $\mathbf{y}(k)$ is the system output, $\{\boldsymbol{\varepsilon}(k+1)\}$ is a sequence of M-dimensional random vectors with zero mean representing the disturbance. $\mathbf{A}$ represents the dependency between the next wasted bandwidth and the current wasted bandwidth, whereas $\mathbf{B}$ represents the dependency between the current assigned bandwidth and the next wasted bandwidth. We explain our approach for deriving these matrices in the following subsection. Note that although the plant is non-linear by nature, linear models often work well for nonlinear systems [29].

11.5.1 System identification

In the above system model (Equation 11.1) matrices $\mathbf{A}$ and $\mathbf{B}$ are unknown. Since the load situation of the components may change during run-time, the above two matrices have to be tuned to improve the accuracy of the model. Therefore, we tune these matrices on-line at each sampling point. This self-learning technique enables the controller to learn the couplings among differ-

ent system outputs/control inputs which may emerge during run-time. In this subsection we assume that **A** and **B** are also functions of the sampling time, i.e. $\mathbf{A}(k)$ and $\mathbf{B}(k)$. Let us rewrite the system model in the following form:

$$\mathbf{y}(k+1) = \mathbf{X}(k)\boldsymbol{\phi}(k) + \boldsymbol{\varepsilon}(k+1), \tag{11.2}$$

where

$$\boldsymbol{\phi}(k) = \Big[(\mathbf{u})^T(k) \quad (\mathbf{y})^T(k)\Big]^T,$$
$$\mathbf{X}(k) = \Big[\mathbf{B}(k) \quad \mathbf{A}(k)\Big].$$

We identify $\mathbf{X}(k)$ during run-time by observing the system outputs, and by making a correction action. We use the Recursive Least Squares (RLS) method [30]. In this method the estimated value of matrix $\mathbf{X}(k)$, denoted by $\hat{\mathbf{X}}(k)$, is calculated using the following equations:

$$\begin{aligned}
\hat{\mathbf{X}}(k+1) &= \hat{\mathbf{X}}(k) + \frac{\boldsymbol{\epsilon}(k+1)(\boldsymbol{\phi})^T(k)\mathbf{P}(k-1)}{\lambda + (\boldsymbol{\phi})^T(k)\mathbf{P}(k-1)\boldsymbol{\phi}(k)}, \\
\boldsymbol{\epsilon}(k+1) &= \mathbf{y}(k+1) - \hat{\mathbf{X}}(k)\boldsymbol{\phi}(k), \\
\mathbf{P}^{-1}(k) &= \mathbf{P}^{-1}(k-1) + \Big(1 + (\lambda-1)\frac{(\boldsymbol{\phi})^T(k)\mathbf{P}(k-1)\boldsymbol{\phi}(k)}{[(\boldsymbol{\phi})^T(k)\boldsymbol{\phi}(k)]^2}\Big) \\
&\quad \boldsymbol{\phi}(k)(\boldsymbol{\phi})^T(k),
\end{aligned} \tag{11.3}$$

where $\boldsymbol{\epsilon}(k+1)$ is the estimation error vector, $\mathbf{P}(k)$ is the covariance vector and λ is the forgetting factor [30].

11.5.2 Controller design

In the following, given that the system model is identified, we design an LQR controller which provides optimal control actions ($\mathbf{u}^*(k)$) at each sampling point k. Note that the LQR controller works after the RLS system identifier. Therefore, in this subsection we assume that the model parameters are already estimated, and we use $\hat{\mathbf{A}}$ and $\hat{\mathbf{B}}$ to indicate the estimated values of **A** and **B**. We define error $\mathbf{e_P}(k)$ as:

$$\mathbf{e_P}(k) = \mathbf{y}^{ref} - \mathbf{y}(k). \tag{11.4}$$

The dynamics of the control system based on $\mathbf{e_P}(k)$ is as follows:

$$\begin{aligned}
\mathbf{e_P}(k+1) &= \mathbf{y}^{ref} - \hat{\mathbf{A}}\mathbf{y}(k) - \hat{\mathbf{B}}\mathbf{u}(k) \\
&= \hat{\mathbf{A}}\mathbf{e_P}(k) - \hat{\mathbf{B}}\mathbf{u}(k) + (\mathbf{I} - \hat{\mathbf{A}})\mathbf{y}^{ref} - \boldsymbol{\varepsilon}(k+1).
\end{aligned}$$

Instead of directly using the model presented in Equation 11.1, we use the system model based on error for the controller design. In addition to $\mathbf{e_P}(k)$, we also use integral errors:

$$\mathbf{e_I}(k+1) = \mathbf{e_I}(k) + \mathbf{e_P}(k),$$

where $\forall k \leq 0$ we have $\mathbf{e_I}(k) = 0$. Hence, the augmented system model is:

$$\mathbf{e}(k+1) = \hat{\mathbf{H}}\mathbf{e}(k) + \hat{\mathbf{S}}\mathbf{u}(k) + \hat{\mathbf{L}} - \boldsymbol{\varepsilon}(k+1), \tag{11.5}$$

where

$$\mathbf{e}(k) = \begin{bmatrix} \mathbf{e_P}(k) \\ \mathbf{e_I}(k) \end{bmatrix}, \quad \hat{\mathbf{H}} = \begin{bmatrix} \hat{\mathbf{A}} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} \end{bmatrix},$$
$$\hat{\mathbf{S}} = \begin{bmatrix} -\hat{\mathbf{B}} \\ \mathbf{0} \end{bmatrix}, \quad \hat{\mathbf{L}} = \begin{bmatrix} \mathbf{I} - \hat{\mathbf{A}} \\ \mathbf{0} \end{bmatrix} \mathbf{y}^{ref}.$$

For our notational convenience we assume $\hat{\boldsymbol{\Psi}} = [\hat{\mathbf{S}} \quad \hat{\mathbf{H}}]$, and we rewrite the above equation:

$$\mathbf{e}(k+1) = \hat{\boldsymbol{\Psi}}\begin{bmatrix} \mathbf{u}(k)^T & \mathbf{e}(k)^T \end{bmatrix}^T + \hat{\mathbf{L}} - \boldsymbol{\varepsilon}(k+1). \tag{11.6}$$

We define the objective function as:

$$J = E\Big\{ \Big|\Big| \mathbf{W}(\mathbf{e}(k+1)) \Big|\Big|^2 + \Big|\Big| \mathbf{Q}(\mathbf{u}(k) - \mathbf{u}(k-1)) \Big|\Big|^2 \Big\}, \tag{11.7}$$

where $\mathbf{W}$ and $\mathbf{Q}$ represent the cost of control error and the cost of control action, respectively. These two matrices allow the system designers to prioritize among different resources by assigning larger cost values to more important resources in case such a logical prioritization is needed. We derive the optimal control action by explicitly capturing the dependency of J on $\mathbf{u}(k)$, and by assigning the derivative of J with respect to $\mathbf{u}(k)$ equal to zero [28]. The optimal control action is:

$$\mathbf{u}^*(k+1) = \Big((\mathbf{W}\hat{\mathbf{S}})^T \mathbf{W}\hat{\mathbf{S}} + \mathbf{Q}^T\mathbf{Q} \Big)^{-1} \Big[(\mathbf{W}\hat{\mathbf{S}})^T \mathbf{W}$$
$$(\mathbf{L} - \hat{\boldsymbol{\Psi}}\tilde{\boldsymbol{\phi}}(k+1)) + \mathbf{Q}^T\mathbf{Q}(\mathbf{u})^T(k) \Big], \tag{11.8}$$

where

$$\tilde{\boldsymbol{\phi}}(k+1) = [\mathbf{0}, (\mathbf{e})^T(k)]^T.$$

At each sampling time k the controller module takes the following actions:

1. It reads the system output vector $\mathbf{y}(k)$ provided by the local resource schedulers.

2. It updates its model of the plant's dynamics using Equation 11.3. The result system model is used in the next step.

3. It calculates the new control input vector $\mathbf{u}^*(k)$ using Equation 11.8.

4. It sends the new reservation bandwidths to the local resource schedulers.

The fact that we selected the wasted resource bandwidth as our system output ($\mathbf{y}$) offers the following advantages: (i) the adaptation scheme is independent of the number of tasks within the components; (ii) the adaptation scheme is independent of the assumed task model. In other words, the above adaptation scheme works under other task models, e.g. models in which a task can branch within its path from the source to different destinations. Nevertheless, in our evaluations we have only considered the task model presented in Section 11.3.

11.6 Evaluations

In order to perform the evaluations we have used a simulation tool, that is called SEtSim and presented in [31]. SEtSim was initially developed to support different real-time network protocols, such as the HaRTES architecture [27]. It was also recently extended to support AVB networks [32]. In this work, we modified the tool such that it supports our end-to-end task model as well as our multi-resource end-to-end reservation scheme. We consider the surveillance component case study presented in Figure 11.1. This is representative of typical network-based surveillance systems in which the bandwidth taken by each camera varies according to the scenario being captured, e.g., variable people walking, variable number of cars or other vehicles, and also where the cameras can be switched on and off on-line. We used a specific network technology for scheduling the network resource as well as a server-based scheduling policy for scheduling the processor resources. Both reservation mechanisms are compliant with the periodic abstraction model assumed in this paper (see section 11.3). We present two simulations in this section. In the first simulation, we studied the response of our controller module to a step load change. That is, we used fixed RCTs times until a certain point in time. Thereafter, we increased the RCTs to larger numbers. In the second simulation, on the other hand, we used RCTs gathered from a real multimedia application to evaluate the performance of the controller module in a real scenario.

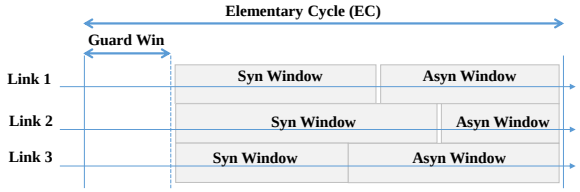


Figure 11.3: The EC partitioning in the HaRTES architecture.

11.6.1 Simulation setup

In the following we explain the details of the resource reservation techniques as well as other parameters that are the same for the two case studies.

We used CBS [6] with hard replenishments on the processor resources. The hard CBS scheme works as follows. The server budget is periodically replenished to its maximum at each server release ($\Pi^{(\iota)}$). The tasks within the server are only allowed to run if the remaining budget is positive. The server budget is reduced while an active task belonging to the server is consuming the processor time. The tasks must stop as soon as the server budget is depleted.

The task parameters were set as follows. The periods of the two end-to-end task (p_1 and p_2) were set to $40ms$. Therefore, their first subtasks, i.e. $\tau_{1,1}$ and $\tau_{2,1}$, were released periodically with the same period. We assumed that $d_1 = d_2 = 40ms$. The messages are activated at the end of the execution of the sender subtasks, i.e. $\tau_{1,1}$ and $\tau_{2,1}$. The final subtasks, i.e. $\tau_{1,4}$ and $\tau_{2,4}$, were activated when they received their corresponding messages. We also set the reservation replenishment period (Π) to $40ms$. We used different RCTs for the two case studies.

We used the HaRTES architecture [27] to connect the processor nodes. Although we used a specific network technology in this evaluation, other network technologies that provide a resource management mechanism can be used in our framework. The HaRTES architecture uses modified Ethernet switches, so called HaRTES switches, which separate traffic in two classes, synchronous and asynchronous (Figure 11.3). The former is scheduled by the switch while the latter is shaped by the switch. Both the synchronous scheduling and the traffic shaping use the temporal resolution dictated by a pre-configured Elementary Cycle (EC). Within each EC, the traffic of each kind is confined to respective windows that vary dynamically in each network link according to the traffic needs (Figure 11.3).

In the case study, the messages are activated by the first subtasks, i.e. $\tau_{1,1}$

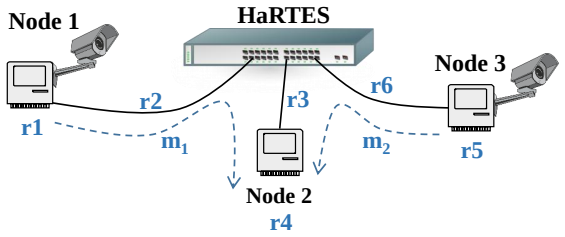


Figure 11.4: The architecture of the case study from a network perspective. Figure 11.1 shows the resource perspective of the same system. Message one (m_1) denotes the following set of subtasks $\{\tau_{1,2}, \tau_{1,3}\}$, while message 2 ($m_2$) represents $\{\tau_{2,2}, \tau_{2,3}\}$.

and $\tau_{2,1}$. Depending on the RCTs of the subtasks, the activation of the messages can be different. Therefore, we considered the messages as asynchronous traffic. A network perspective of the case study is shown in Figure 11.4. The network parameters are set as follows. The network bandwidth capacity is set to 100Mbps. The EC size is selected to $40ms$. The bandwidth reservations for the transmission of the messages ($\alpha_h(k)$) are done within the asynchronous window and changed by the controller during run-time depending on the load and bandwidth usage (resources r_2, r_3 and r_6). The processor nodes (Node 1 and Node 3 in Figure 11.4) generate two messages, respectively, denoted by m_1 and m_2 in the figure. The destination of the messages is Node 2, where the processing of the data is performed. We used fixed priority scheduling for scheduling subtasks within each resource reservation. We assumed that τ_1 (also its subtasks) has a higher priority than τ_2.

Regarding the controller parameters, we set the sampling interval to $200ms$. Also, we set $\mathbf{Q} = 0.1 \times \mathbf{I}$ and $\mathbf{W} = \text{tri}(0.1, 0.1, 0.1, 0.1, 0.1, 0.1) + \mathbf{I}$ to give a smaller weight for the integral errors. Note that tri returns a lower triangular matrix of its input vector. Based on the discussion presented in Section 11.5, we assigned the following reference vector: $\mathbf{y}^{ref} = [0.08\ 0.10\ 0.15\ 0.08\ 0.08\ 0.10]^T$.

11.6.2 Case study (1): step response

Table 11.1 shows the RCTs used in this case study before and after a step change in the RCTs. We changed the RCTs $4s$ after the beginning of the simulations (i.e., after 100 task instances), and we ran the simulations for $10s$.

	Before step	After step
$c_{1,1}$, $c_{1,4}$, $c_{2,1}$, $c_{2,1}$	$4ms$	$10ms$
$c_{1,2}$, $c_{2,2}$	$2ms$	$5ms$
$c_{1,3}$, $c_{2,3}$	$4ms$	$10ms$

Table 11.1: The RCTs used in case study (1).

Figure 11.5 shows the consumed bandwidth α'_h, assigned bandwidth α_h, and the control error (Equation 11.4) for each resource separately. Although the RTCs are fixed throughout the first 10 samples, the controller modifies the assigned bandwidths. This is because in the beginning of the simulations the controller needs to adjust its model. Let $\bar{\boldsymbol{\epsilon}}$ denote the average observed value of $\boldsymbol{\epsilon}(k)$ (Equation 11.3). We observed the following average differences between the estimated system outputs and the real system outputs:

$$\bar{\boldsymbol{\epsilon}} = 10^{-4} \times [-74;\ 49;\ 54;\ -107;\ -75;\ 48],$$

which shows that the RLS approach has been successful in identifying the system model.

The task response times are illustrated in Figure 11.6. During the transient period of load adjustment, a few instances of the end-to-end tasks missed their deadlines. τ_1 missed 20 deadlines, while τ_2 missed 44 deadlines. Recall that τ_2 had a lower priority than τ_1, hence it was scheduled later on the shared resources (still within the component reservations). In addition, since after the step load a backlog was built, it took a while until the response times became stable again.

11.6.3 Case study (2): multimedia application

In this simulation, in order to have realistic evaluation of our framework, we used RCTs gathered from running multimedia tasks on a real hardware platform [33]. We also considered $10\,\%$ high priority interfering workload on all of the resources to simulate existence of other components. Figure 11.7 presents the evolution of the used bandwidths, assigned bandwidths and control errors on all resources during the $100s$ experiment (i.e., 500 control samples). The figure shows that the controller manages to successfully track the evolution of the workload. Figure 11.8 illustrates the response times of the two end-to-end tasks. Since τ_2 has a lower priority than τ_1, it has larger response times, and it occasionally violates its deadline. In total, τ_1 missed 4 deadlines, while τ_2

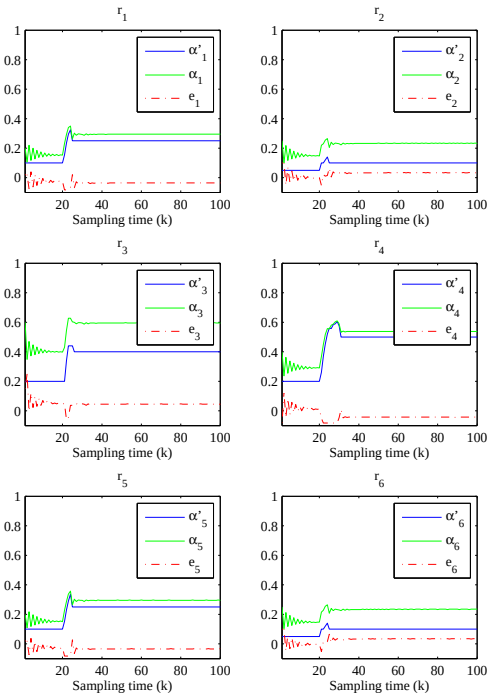


Figure 11.5: α'_h, α_h and e_h of the six resources during case study (1).

missed 40 deadlines. We observed:

$$\bar{\boldsymbol{\epsilon}} = 10^{-4} \times [5;\ 1;\ -9;\ 15;\ 2;\ 16].$$

Let $\boldsymbol{\alpha}^{avg}$ denote the vector of average assigned bandwidths throughout the experiment. We had:

$$\boldsymbol{\alpha}^{avg} = [0.1977;\ 0.2600;\ 0.6378;\ 0.3686;\ 0.2030;\ 0.2174].$$

In a new experiment, we used the above observed average bandwidths $\boldsymbol{\alpha}^{avg}$, and we assigned them as fixed bandwidths to the component. In total, the number of deadline misses for τ_2 was 46, while τ_1 missed all of its

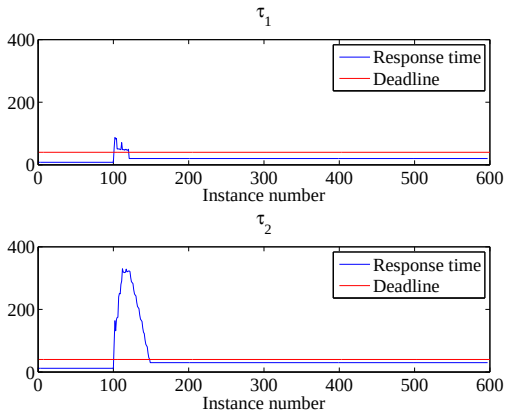


Figure 11.6: End-to-end response times of τ_1 and τ_2 in case study (1).

deadlines. This is because the first instance of $\tau_{1,1}$ finished in the 6^{th} reservation period. This phenomenon caused a large backlog for τ_1. On the other hand, since α_2 was not enough for sending multiple messages, τ_1 never recovered from this backlog. The average response time for τ_1 was $248ms$. This experiment shows that given a certain resource efficiency level (i.e. reservation bandwidth) our adaptive framework works significantly better than the static design approach. In should be noted that the static design approach requires bandwidth estimations prior to the run-time.

11.6.4 Overhead

Our adaptive framework performs adaptations at the cost of imposing two types of overhead: (i) communication overhead; (ii) computation overhead. The controller requires the bandwidth usage information ($\mathbf{y}(k)$) during run-time to adjust the bandwidth for each network resource. This information is gathered by the resource scheduler in the nodes and switches per link, and it is transmitted to the controller by means of messages. Therefore, besides the data messages transmitted through the network, the controller messages are sent through the same links. For this particular information a specific bandwidth is reserved in the network, that is isolated from the data message bandwidth. For instance, in the HaRTES architecture, the data messages are transmitted within the asynchronous window, while the control messages are transmitted within the syn-

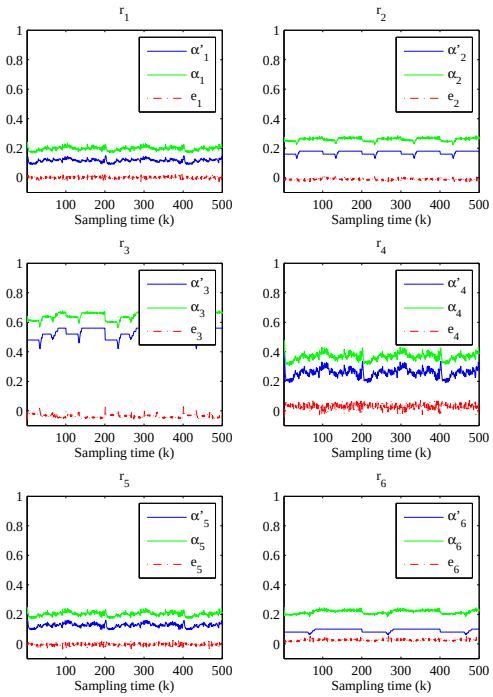


Figure 11.7: α'_h, α_h and e_h of the six resources during case study (2).

chronous window. Note that the controller is performed periodically, thus the control messages can be transmitted periodically as a set of synchronous messages. For this purpose we can send messages with the size equal to 500 Bytes ($40\mu s$) every reservation period ($40ms$) which imposes 0.1 % overhead on the network. The number of control messages can be reduced by devising a decentralized control scheme similar to [34], where we can decompose the system model presented in Equation 11.1. However, we leave investigating this approach for the future work.

Furthermore, the controller performs computations to first update its model of the system using the RLS technique (Equation 11.3), and to calculate the control input $\mathbf{u}(k)$ (Equation 11.8). For the purpose of the above case study,

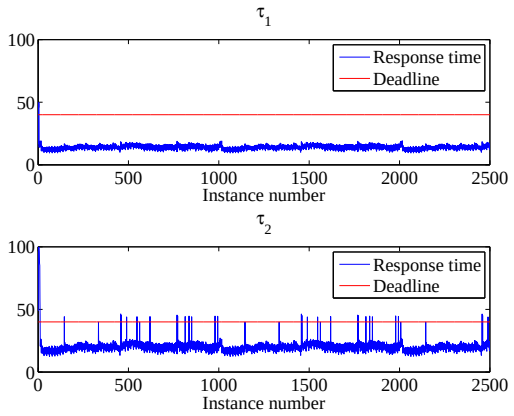


Figure 11.8: End-to-end response times of τ_1 and τ_2 in case study (2).

we implemented the two functions in Matlab, and we ran them on our machine featuring an Intel Core i7-4600U processor with 12 GB RAM. The RLS identification took in average $373\mu s$, while the LQR computations took in average $240\mu s$. Therefore, given the sampling period, the controller module imposed around $0.31\ \%$ computation overhead. Note that the above values are measured running Matlab code on a Windows operating system. Therefore, an efficient implementation executed on a real-time operating system will potentially impose lower overhead. In general, overhead is proportional to the total number of resources used by the component $M^{(\iota)}$.

11.6.5 Discussions

We conclude from the above case studies that our controller module manages to fulfill the objectives described in Section 11.5, that are (i) to serve the components with negligible deadline violations; (ii) to allocate the resources efficiently by matching the reservation bandwidths to the instantaneous needs during run-time (instead of overprovisioning the resources). Considering the network resources in the evaluation, we changed the reservations within the asynchronous window sizes in the EC during run-time. This allows other components in the asynchronous window to utilize more bandwidth. Also, we can reduce the size of the asynchronous window if it is not used by the components within this window. In doing so, the size of the other window, i.e., the

synchronous window can be increased. Therefore, more resource is available for the components that use the synchronous window in the EC. The same argument applies for the processor resources. The component allows other components that share the resources with itself to utilize larger bandwidths by keeping its reservation size low.

11.7 Conclusions and future work

In this paper we designed an adaptive framework for scheduling component-based distributed real-time systems. In our framework we enforce end-to-end reservations across all of the resources needed by the end-to-end tasks within the components. The sizes of the reservations are adjusted during run-time to cope with dynamic resource needs. We showed, using two case studies, that our framework reduces the number of deadline violations to a negligible level, while keeping the reservation sizes close the actual demands.

In the future we will propose a protocol to manage overload situations in which the overall resource reservation on a resource is beyond its schedulability threshold. In addition, we will investigate a decentralized control approach to see whether we can reduce the communication overhead of the controller while keeping its performance at an acceptable level.

References

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

[2] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 26–35, May 2001.

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

[4] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, January 2009.

[5] N. Khalilzad, M. Behnam, and T. Nolte. Multi-level adaptive hierarchical scheduling framework for composing real-time systems. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages 320–329, August 2013.

[6] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 4–13, December 1998.

[7] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina. An implementation of the earliest deadline first algorithm in Linux. In *Proceedings of the ACM symposium on Applied Computing (SAC'09)*, pages 1984–1989, March 2009.

[8] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril. Towards hierarchical scheduling on top of VxWorks. In *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008.

[9] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino. Resource management on multicore systems: The ACTORS approach. *Micro, IEEE*, 31(3):72–81, May-June 2011.

[10] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, June 2004.

[11] IEEE. IEEE Std. 802.1qav, ieee standard for local and metropolitan area networks, virtual bridged local areanetworks, amendment 12: Forwarding and queuing enhancements for time-sensitive streams. Technical report, IEEE, 2011.

[12] Ethernet POWERLINK Standardisation Group. *EPSG Draft Standard 301 Ethernet POWERLINK Communication Profile Specification Version 1.2.0*, 2013.

[13] M. Ashjaei, M. Behnam, L. Almeida, and T. Nolte. Performance analysis of master-slave multi-hop switched ethernet networks. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13)*, June 2013.

[14] R. Santos, A Vieira, P. Pedreiras, A Oliveira, L. Almeida, R. Marau, and T. Nolte. Flexible, efficient and robust real-time communication with server-based Ethernet switching. In *Proceedings of the 8th IEEE International Workshop on Factory Communication Systems (WFCS'10)*, May 2010.

[15] R. Santos, M. Behnam, T. Nolte, P. Pedreiras, and L. Almeida. Multi-level hierarchical scheduling in ethernet switches. In *Proceedings of the of the International Conference on Embedded Software (EMSOFT'11)*, October 2011.

[16] J. Silvestre-Blanes, L. Almeida, R. Marau, and P. Pedreiras. Online QoS management for multimedia real-time transmission in industrial networks. *IEEE Transaction on Industrial Electronics*, 58(3), March 2011.

[17] IEEE 802.1Qat, draft standard for local and metropolitan area networks virtual bridged local area networks amendment 9: Stream reservation protocol (SRP).

[18] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. Rsvp: a new resource reservation protocol. *IEEE Communications Magazine*, 40(5):116–127, May 2002.

[19] K. Lakshmanan and R. Rajkumar. Distributed resource kernels: OS support for end-to-end resource isolation. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, April 2008.

[20] A. Oliveira, A. Azim, S. Fischmeister, R. Marau, and L. Almeida. D-RES: Correct transitive distributed service sharing. In *Proceedings of the Work-in-Progress Session of the Conference on Emerging Technologies and Factory Automation (ETFA'14)*, September 2014.

[21] T. Cucinotta and L. Palopoli. QoS control for pipelines of tasks using multiple resources. *IEEE Transactions on Computers*, 59(3):416–430, March 2010.

[22] R. Rajkumar, C. Lee, J. Lehoczky, and Dan Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, December 1997.

[23] S. Ghosh, J. Hansen, R. Rajkumar, and J. Lehoczky. Integrated resource management and scheduling with multi-resource constraints. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, December 2004.

[24] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 59–70, December 2001.

[25] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, June 2005.

[26] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. DEUCON: Decentralized end-to-end utilization control for distributed real-time systems. *IEEE*

Transactions on Parallel and Distributed Systems, 18(7):996–1009, July 2007.

[27] M. Ashjaei, M. Behnam, P. Pedreiras, R. J. Bril, L. Almeida, and T. Nolte. Reduced buffering solution for multi-hop HaRTES switched Ethernet networks. In *The 20th IEEE International Conference on embedded and Real-Time Computing Systems and Applications*, August 2014.

[28] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *Proceedings of the 46th IEEE Conference on Decision and Control (CDC'07)*, pages 3792–3799, December 2007.

[29] J. L. Hellerstein, Y. Diao, S. Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[30] K. J. Åström and B. Wittenmark. *Adaptive control*. Addison-Wesley, Reading, Mass., 2. ed. edition, 1995.

[31] M. Ashjaei, M. Behnam, and T. Nolte. The design and implementation of a simulator for switched ethernet networks. In *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2012.

[32] M. Ashjaei, M. Behnam, and T. Nolte. SEtSim: A modular simulation tool for switched Ethernet networks. Technical report, September 2014.

[33] C. C. Wust, L. Steffens, W. F. J. Verhaegh, R. J. Bril, and C. Hentschel. QoS control strategies for high-quality video processing. *Real-Time Systems*, pages 3–12, 2005.

[34] J. Yao, X. Liu, X. Chen, X. Wang, and J. Li. Online decentralized adaptive optimal controller design of cpu utilization for distributed real-time embedded systems. In *Proceedings of the American Control Conference (ACC'10)*, pages 283–288, June 2010.

Chapter 12

Paper E: Exact and Approximate Supply Bound Function for Multiprocessor Periodic Resource Model: Unsynchronized Servers

Nima Khalilzad, Moris Behnam and Thomas Nolte.
In ACM SIGBED Review special issue on the 5th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'12), Volume 10, Number 3, October, 2013.

Abstract

The Multiprocessor Periodic Resource (MPR) model has been proposed for modeling compositional real-time systems which run on a shared multiprocessor hardware. In this paper we extend the MPR model such that the execution of virtual processors (servers) is not assumed to be synchronized i.e., the servers can have different phases. We believe that relaxing the server synchronization requirement provides greater deal of compatibility for implementing such a compositional method on various hardware platforms. We derive the resource supply bound function of the extended MPR model using an algorithm. Furthermore, we suggest an approach to calculate an approximate supply bound function with lower computational complexity for systems where calculating their supply bound function is computationally expensive.

12.1 Introduction

In order to deal with increasing complexity of real-time systems, hierarchical scheduling techniques have been proposed and investigated for scheduling complex real-time systems consisting of multiple real-time components (applications) on a shared underlying hardware platform. Using such techniques components are developed independently and their timing behaviors are studied in isolation, while the correctness of the system is inferred from the correctness of its components. In the mean time, following the trend of servers and PCs, embedded real-time systems are subjected to the paradigm shift from single processor to multiprocessor hardware platforms. Therefore, there is a need for new techniques that can enable hierarchical scheduling on multiprocessor platforms which allow us to compose real-time systems and run them on a multiprocessor hardware. Recently, many studies have been conducted on this subject and a variety of models have been proposed.

In modeling hierarchical real-time systems, single processors alike multiprocessors, the system model often consists of two parts: resource supply model and task demand model. The resource supply model abstracts the underlying hardware resource such that each application has the illusion of running solo on an independent hardware, this virtual hardware is often called a *server*. The resource supply model represents the minimum amount of resource that a server provides in a given time interval. The amount of provided resource is often represented using a Supply Bound Function ($\texttt{sbf}(t)$). The resource demand model, however, represents the resource demand of real-time tasks. Similarly, the maximum demand is often represented using a Demand Bound Function ($\texttt{dbf}(t)$) [1]. Consequently, the schedulability test is performed using the $\texttt{sbf}(t)$, which is dependent on the resource model, and the $\texttt{dbf}(t)$ which is dependent on the scheduling policy.

When it comes to multiprocessor platforms, the resource supply model can either be flexible and represent the collectively provided resource of a set of processors [2], or it can be more detailed and represent the exact amount of provided resource by each processor. In the former case, as Lipari and Bini state in [3], the $\texttt{sbf}(t)$ depends on the fact that whether different servers (on different processors) are synchronized together or not. A number of works on multiprocessor hierarchical scheduling assume that the servers are synchronized [2, 4], while in this paper alike [3], we assume that the servers are not synchronized. Indeed synchronization on some hardware platforms can be expensive, therefore, we simplify the implementation phase of the composition for the system developers by relaxing this assumption. Figure 12.1 illustrates

the supply bound function of a multiprocessor periodic resource model for two cases: synchronized servers and unsynchronized servers (the specifications are explained later in Example 1). The figure shows that when the servers are not synchronized the supply bound function at some points in time is lower than the synchronized servers case. The figure indicates that the schedulability analysis that is based on the assumption of having synchronized servers is not valid when this assumption is relaxed.

In this paper, we focus on the supply bound function of the multiprocessor periodic resource model, and we present an approach to calculate the $\texttt{sbf}(t)$ of the flexible resource model that Easwaran *et al.* presented in [2] with no assumptions on the server synchronization. Our approach is based on mapping the flexible model to a model that represents the exact amount of the contributed budget by each processor to the total budget, and then we derive the $\texttt{sbf}(t)$ for the new model. Furthermore, we present an approach for approximating the $\texttt{sbf}(t)$ which has lower computational complexity than calculating the actual $\texttt{sbf}(t)$.

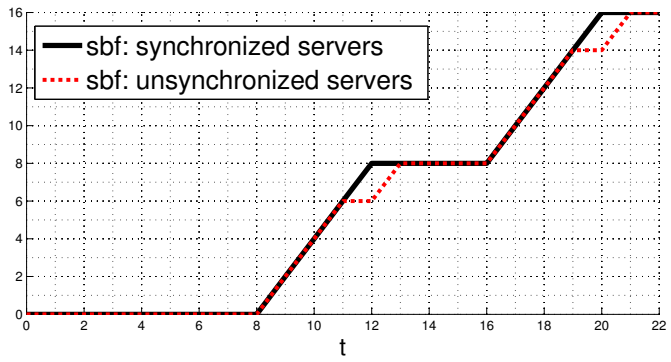


Figure 12.1: The $\texttt{sbf}(t)$ of synchronized and unsynchronized servers

The rest of the paper is organized as follows. We first review the related work in Section 12.2, then we present the resource model in Section 12.3. The algorithm for calculating the exact supply bound function is presented in Section 12.4. Thereafter, we present the approximate supply bound function in Section **??**. Finally, we conclude the paper in Section 12.6.

12.2 Related work

Hierarchial scheduling was first proposed as a method for composing real-time systems on single processor hardware platforms. Enabling independent development of real-time systems, Deng and Liu proposed hierarchical scheduling in [5]. Schedulability analysis under global fixed priority scheduling is presented in [6]. Mok *et al.* presented the bounded-delay model for single processor hardware platforms in [7]. Shin and Lee presented the periodic resource model for single processors in [8].

Virtual clustered-based multiprocessor scheduling [2], which is the extension of the periodic resource model for multiprocessor platforms, provides a flexible mechanism for scheduling hierarchical systems. In this approach the resource supply is abstracted using a Multiprocessor Periodic Resource (MPR) interface. The MPR interface consists of P, Q and m parameters which denote the total budget Q is provided in each period P using m virtual processors. This model provides a great deal of run-time flexibility since the budget distribution among m processors is performed during run-time depending on the load of processors. This flexibility can be exploited by the scheduler to serve the real-time tasks in an efficient way. It is shown in [2] that the minimum supply bound happens in the case where the total required budget is evenly divided among all processors and each processor's budget is equal to $\frac{Q}{m}$. Therefore, the supply bound function is derived based on this worst-case budget distribution setting (this setting is called the *worst-case platform* in [3]).

The main problem with the MPR interface is that it has an implicit assumption of the synchronization among virtual processors. It has been shown in [3] that the worst-case platform does not exist if the virtual processors are not synchronized. Therefore, Lipari and Bini suggested a new interface model, namely the Bounded-Delay Multipartition (BDM) model to overcome this problem. The BDM interface consists of m, Δ and $[\beta_1,...,\beta_m]$ parameters which represent the number of virtual processors, the length of the longest interval with no resource and the bandwidth at each parallelism level respectively. In fact, the DBM model replaces the notion of period P in the MPR with delay (the longest interval with no resource) Δ. The BDM model does not require the servers to be synchronized. Nevertheless, due to the nature of the delay based models, the BDM can be very pessimistic which can result in low system run-time utilization and consequently higher cost of the system production. Besides, from an implementation point of view, periodic servers are more straight forward to implement, and the BDM model perhaps should be mapped to the MPR or any other periodic server based model for the implementation.

Bini *et al.* presented the Multi Supply Function (MSF) model in [9] for modeling the resource supply in hierarchial scheduling on multiprocessor platforms. The MSF is indeed a set of supply functions one associated with each server. The Parallel Supply Function (PSF) model [10] is also proposed as an alternative for modeling the resource supply of hierarchial multiprocessor systems. This model indicates a set of supply functions where each of them represent the minimum available supply at a certain parallelism level (from 1 to m). Since the MPR model offers greater deal of abstraction than the MSF and the PSF model, from a system integrator perspective, the MPR can be more suitable when composing real-time systems.

Zhu *et al.* have extended deferable servers to the context of multiprocessor platforms [4] where m deferable servers with a common period and different budgets are running on m processors. Analogous to the MPR model, the servers are assumed to be synchronized in this work.

Targeting soft real-time tasks, Leontyev and Anderson have presented a multi-level scheme for scheduling real-time tasks and they showed that under their scheme, the deadline tardiness of the tasks is bounded [11]. In contrast with other hierarchical schemes, in this work there is no loss in overall utilization moving down through the levels of hierarchy.

12.3 Resource model

We present the resource model for a processor cluster in this section. On a multiprocessor platform consisting of a total of n processors, a processor cluster is a set of m processors where $1 \leq m \leq n$. The processor clusters can be either physically or virtually mapped to the physical processors [2].

We present two types of resource interface models for the processor clusters: flexible and rigid. While the flexible model is the main focus in this paper, the reason behind introducing the rigid interface is that we use it to derive the supply bound function of the flexible interfaces.

12.3.1 Flexible interface model

Our flexible resource interface model is equivalent to the one that Easwaran *et al.* introduced in [2]. In this model resources are specified by the following tuple: $\Gamma = \langle m, P, Q \rangle$, which denotes that the multiprocessor cluster consisting of m processors in total provides Q units of budget every P period to its corresponding consumers. From a run-time point of view, this model is very flexible

in the sense that the scheduler can decide how the total budget should be distributed among the processors in the cluster, in other words, each processor is free to provide as much resource as it wants, as long as the collective provided budget is equal to Q every P time units.

12.3.2 Rigid interface model

In contrast to the flexible interface model, in the rigid interface model each processor in the cluster is required to provide a specific amount of resource to its corresponding consumers. The rigid interface model is represented as follows: $\psi = \langle m, P, [q_1, ..., q_m] \rangle$, where q_i represents the exact amount of the budget of processor i ($1 \leq i \leq m$). Without loss of generality we assume that all q_i are stored non-increasingly i.e. $\forall i \; q_i \geq q_{i+1}$. In this model, the total provided budget is calculated by accumulating the budget of all processors in the cluster: $\sum_{i=1}^{m} q_i$. We use the following notation to refer to the budget distribution of a known platform (ψ): q_i^{ψ} where $1 \leq i \leq m$. Similarly Q^{Γ} represents the total available budget of the flexible interface Γ. Note that in this model processor i, regardless of the budget of other processors in the cluster, is obliged to provide q_i budget each period and it does not need to be synchronized with the other processors in its cluster.

We overload the word "platform" in the rest of the paper to refer to a rigid processor cluster interface ψ. Since the total budget can be distributed among processors in many ways, a single flexible interface Γ can be mapped to many platforms. We call the set of all possible platforms derived from a flexible interface Γ the *possible platforms of* Γ and we represent this set as follows

$$\Psi^{\Gamma} = \left\{ \forall \psi : \sum_{i=1}^{m} q_i^{\psi} = Q^{\Gamma} \right\}.$$

Note that when Q is not integer, we solve the mapping problem for $\lfloor Q \rfloor$ and then we add $Q - \lfloor Q \rfloor$ to q_1. Therefore, we assume that in the rigid model there exist at most one real budget (q_1), while the rest of the processors have integer budgets. The restriction that only a single processor will have real budget, limits the set of possible rigid interfaces that can be derived from a flexible interface.

12.3.3 Flexible interface versus rigid interface

So far we have introduced two interface models which can be used for composing real-time components on a multiprocessor hardware. When using rigid

models, each component has its own q_i^{ψ} and the system integrator has to find a way to allocate q_i to the physical processors. This problem is a bin packing like problem which is known to be difficult to be solved. In addition, when adding or removing components, the allocation should be repeated.

On the other hand, when using a flexible interface we do not face this allocation problem and the scheduler is free to decide the allocations in any fashion at run-time. This property makes the flexible interfaces more suitable for compositional analysis in the sense that the integration phase is done without the need to consider the physical allocation of the components. However, calculating the supply bound function when using flexible interfaces, as we discuss in this paper, requires more computations than using rigid interfaces.

Therefore, there is a downside to both of the models and choosing either of them is a design decision which should be made by the system designers.

12.3.4 Packed platform of a flexible interface

The packed platform (ψ_p) of a flexible interface Γ is a member of Ψ^{Γ} in which the total budget Q is packed onto the minimum number of processors. A packed platform consists of $h = \lfloor \frac{Q}{P} \rfloor$ full budgets ($q_i = P$), one budget equal to $\mod(Q, P)$, and $m - h$ empty budgets ($q_i = 0$). For example the corresponding packed platform of the flexible interface $\Gamma = \langle 4, 8, 18 \rangle$ is $\psi_p = \langle 3, 8, [8, 8, 2, 0] \rangle$.

12.3.5 Balanced platform of a flexible interface

The balanced platform (ψ_b) of a flexible interface Γ is a member of Ψ^{Γ} in which the total budget is evenly divided among all processors in the cluster. Therefore, the balanced platform consists of $k = \mod(Q, m)$ budgets equal to $\lfloor \frac{Q}{m} \rfloor + 1$ and $m - k$ budgets equal to $\lfloor \frac{Q}{m} \rfloor$. For instance the balanced platform of $\Gamma = \langle 4, 8, 18 \rangle$ is $\psi_b = \langle 4, 8, [5, 5, 4, 4] \rangle$.

12.3.6 Deriving the possible platforms of a flexible interface

The problem of deriving the possible platforms of a flexible interface is analogous to the well know integer partitioning problem in number theory [12], where the problem is to find all possible ways that an integer number x can be written as a sum of some integer numbers which are called the partitions of x. However, in our problem we have two additional constraints which are the maximum number of partitions (m) and the maximum value of each partition

(P). Hence, we can not directly use the algorithms presented for deriving the partitions of integer numbers. Therefore, the problem is to find all possible ways of writing Q as sum of ℓ integer numbers (q_i) where $\ell \leq m$, and each partition value is less than or equal to P. Recall that for avoiding redundant platforms we enforce the following requirement $\forall i\ q_i \geq q_{i+1}$, which is due to the fact that redundant platforms have equivalent $\texttt{sbf}(t)$ and therefore are not of our interest.

In the rest of this section we present an algorithm for deriving the possible platforms of a given flexible interface. We start from the balanced platform and construct a tree where the root is ψ_b and new nodes are created by transferring a unit of the budget from one processor to another one. We present some definitions before presenting the algorithm.

Budget donor candidate is a processor that if its budget is reduced by one the remaining budget set is still ordered (non-increasingly). Any given platform has a budget donor candidate set (D^ψ) that is found using Algorithm 7. The algorithm loops through all budgets and selects the ones that are compatible with donating a unit of budget. `insert` is a function that inserts a new entry (i) to its input set (here D^ψ).

Algorithm 7: Deriving the budget donor candidate set

```
1:  function donors(ψ)
2:  for i = 2;i < m;i++ do
3:     if q_i > 0 & q_i > q_{i+1} then
4:        insert(D^ψ, i);
5:     end if
6:  end for
7:  if q_m > 0 then
8:     insert(D^ψ, m);
9:  end if
10: end function
```

Budget receiver candidate is a processor that if its budget is increased by one the remaining budget set is still ordered (non-increasingly). There is a budget receiver set associated with each budget donor of platforms (R_d^ψ) which is derived using Algorithm 8. The algorithm only loops through the budgets that are at the left hand side of the budget donor d, and finds the processors that are compatible with receiving a unit of budget.

Budget donation operation is an operation in which one unit of a budget

Algorithm 8: Deriving the budget receiver candidate set of a given budget donor (d)

```
1: function receivers(ψ, d)
2: for i = 2;i < d;i++ do
3:    if q_i < P & q_i < q_{i-1} then
4:       insert(R_d^ψ, i);
5:    end if
6: end for
7: if q_1 < P then
8:    insert(R_d^ψ, 1);
9: end if
10: end function
```

donor's budget q_d is transferred to a budget receiver budget q_r.

In order to derive Ψ^{Γ}, we start off by running the budget donation operation on ψ_b, for all combinations of the donors and their corresponding receivers. Thereafter, we repeat this step for all children of ψ_b and create the next level of the tree. The procedure continues until we reach ψ_p, which is the packed platform, and since the budget donation operation can not be performed on the packed platform the algorithm stops branching. The pseudocode of this procedure is presented in Algorithm 9. `isNew` is a function that looks for its input platform (ψ') in its input set (Ψ^{Γ}) and returns true if it fails to find the platform. The budget donation operation takes place in line 7 and 8 of the algorithm, and in line 11 (when we find a new platform) we do a recursive call passing the recently found platform. Since the budget donation operation transfers only one unit of the budget at each step, and we run this operation on all combinations of donors and their corresponding receivers, we are guaranteed to i) reach ψ_p which is the termination condition of our recursive algorithm ii) find all possible platforms between ψ_b and ψ_p.

Since we start from ψ_b where

$$\overbrace{q_1, ..., q_k}^{=\lfloor \frac{Q}{m} \rfloor + 1}, \overbrace{q_{k+1}, ..., q_m}^{=\lfloor \frac{Q}{m} \rfloor}$$

and we want to reach ψ_p where

$$\overbrace{q_1, ..., q_h}^{=P}, \quad \overbrace{q_{h+1}}^{=\ \mathrm{mod}\ (Q,P)}, \overbrace{q_{h+2}, ..., q_m}^{=0}$$

Algorithm 9: Deriving possible platforms of a flexible interface

```
1:  function platforms(ψ, ψ_p)
2:  D^ψ = donors(ψ);
3:  for all d ∈ D^ψ do
4:      R_d^ψ = receivers(ψ, d);
5:      for all r ∈ R_d^ψ do
6:          ψ' = ψ;
7:          ψ'.q_d = q_d − 1;
8:          ψ'.q_r = q_r + 1;
9:          if ψ' ≠ ψ_p & isNew(ψ', Ψ^Γ) then
10:             insert(ψ', Ψ^Γ);
11:             platforms(ψ', ψ_p);
12:             return ψ';
13:         end if
14:     end for
15: end for
16: end function
```

therefore q_{h+1} to q_m in ψ_b should be moved to a place between q_1 and q_h. In this process each q_i can at most move $i-1$ steps to the left, and the longest depth happens when the donor processor and the receiver processors at all steps are neighbors ($d = r+1$). Therefore, the longest depth (κ) is calculated by the following equation

$$\kappa = \sum_{i=h+1}^{m} q_i \times (i-1), \tag{12.1}$$

where m is the number of processors, $h = \lfloor \frac{Q}{P} \rfloor$, and q_i are the budgets of ψ_b. The total number of possible platforms is exponential in κ. Note that κ is derived without considering that i) all q_i are sorted non-increasingly ii) redundant nodes are not allowed to branch. Therefore, in practice the longest depth is less than or equal to κ. However, since the growth rate of the algorithm is exponential it is considered as a high complexity problem which might be intractable for some configurations of m, P and Q. A sample trace of the algorithm for the flexible interface $\Gamma = \langle 3, 8, 6 \rangle$ is presented in Figure 12.2. The redundant nodes are presented as gray nodes which are eliminated using the `isNew` function. The figure illustrates the necessity of eliminating redundant nodes since they are a considerable number of the total nodes.

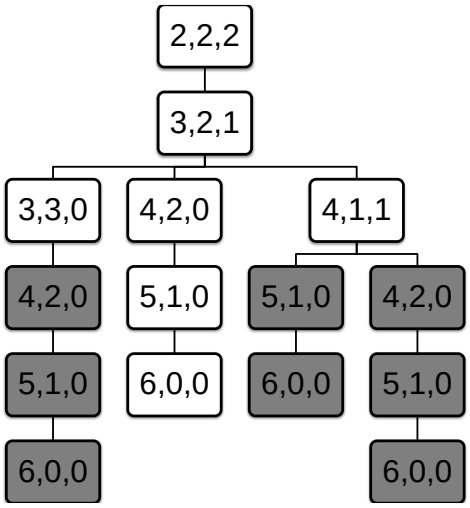


Figure 12.2: possible rigid platforms of $\Gamma = \langle 3, 8, 6 \rangle$

12.4 Supply bound function

The Supply Bound Function ($\texttt{sbf}(t)$) represents the minimum amount of the resources that servers provide to their task set in a given time interval t. In this section we derive the supply bound function of both the flexible and the rigid interface models. For simplifying the presentation we drop t when referring to the supply bound function in the rest of the text.

12.4.1 The `sbf` of rigid interfaces

The `sbf` of a rigid interface $\texttt{sbf}^{\psi}$ can be seen as sum of m servers' `sbfs` with the period equal to P and given budgets q_i. Therefore, using the same equation that is presented in [8] for calculating the `sbf` we have:

$$\texttt{sbf}^{\psi}(t) = \sum_{i=1}^{m} \Big(\Big\lfloor \frac{t - (P - q_i^{\psi})}{P} \Big\rfloor \times q_i^{\psi} + \epsilon_i(t) \Big), \tag{12.2}$$

where

$$\epsilon_i(t) = \max\Big(t - 2(P - q_i^{\psi}) - p \times \Big\lfloor \frac{t - (P - q_i^{\psi})}{P} \Big\rfloor, 0\Big). \tag{12.3}$$

12.4.2 The sbf of flexible interfaces

In order to calculate the minimum supply provision of a flexible interface ($\texttt{sbf}^{\Gamma}$) we need to derive the worst-case platform which provides the least amount of resources among all possible platforms. However, as Lipari and Bini state in [3], the worst-case platform does not exist for the flexible interfaces. Although the balanced rigid platform is the worst-case platform when we assume that the virtual processors are synchronized [2], when the assumption is relaxed it is not the worst-case platform anymore. Therefore, a potential solution for calculating the sbf of flexible interfaces is to take the following steps:

1. Derive Ψ^{Γ} using Algorithm 9.
2. From the definition of the supply bound function, $\texttt{sbf}^{\Gamma}$ at each time point is the minimum of all $\texttt{sbf}^{\psi}$ at that time:

$$\texttt{sbf}^{\Gamma}(t) = \min\Big\{\texttt{sbf}^{\psi}(t)\Big\},\ \forall \psi \in \Psi^{\Gamma}. \tag{12.4}$$

As we discussed in the previous section, the complexity of step one is exponential, hence this solution might be intractable for some flexible interfaces. Therefore, in the rest of this section we take some actions in reducing the complexity of step one by removing the platforms that are not contributing in calculating the $\texttt{sbf}^{\Gamma}$. Indeed, we are looking for the platforms where $\texttt{sbf}^{\psi}$ crosses $\texttt{sbf}^{\psi_b}$ at least at one time point, or mathematically:

$$\forall \psi\, \exists t\ : \texttt{sbf}^{\psi}(t) < \texttt{sbf}^{\psi_b}(t).$$

Example 1. *Consider the following flexible interface $\Gamma_1 = \langle 2, 8, 8\rangle$, the* sbf *of all possible platforms is shown in Figure 12.3 (redrawn from [3]). Among all possible platforms of Γ_1, only $\texttt{sbf}^{\psi_1}$ and $\texttt{sbf}^{\psi_2}$ are crossing $\texttt{sbf}^{\psi_b}$. Therefore, in the proposed approach for calculating $\texttt{sbf}^{\Gamma}$, it is sufficient to only derive ψ_1 and ψ_2 in step 1, and proceed with the second step. Roughly speaking, we present an approach to exclude the platforms where the lower bound of their* sbf *is higher than the upper bound of $\texttt{sbf}^{\psi_b}$ (ψ_3 and ψ_p in this example).*

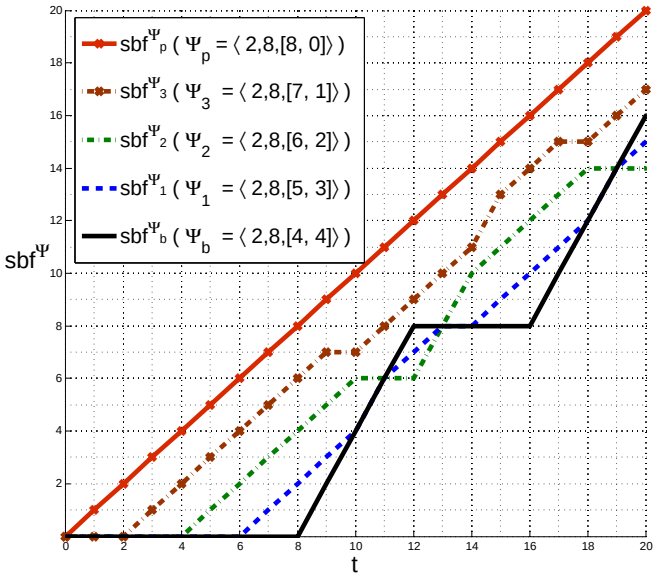


Figure 12.3: The sbf of all possible platforms of $\Gamma_1 = \langle 2, 8, 8 \rangle$

To this end, we first show how to calculate the lower bound (lsbf) and upper bound (usbf) of the supply bound function, afterwards we derive a subset of Ψ^Γ which is sufficient for calculating the $\texttt{sbf}^\Gamma$.

12.4.3 The lsbf of rigid interfaces

Analogous to the presented approach for calculating the $\texttt{sbf}^\psi$, linear lower bound for a rigid interface $\texttt{lsbf}^\psi$ can be calculated by accumulating the linear lower bound of m independent servers' lsbf with a given period (P) and budget (q_i). According to [8], the lsbf is calculated as follows

$$\texttt{lsbf}(t) = \frac{q_i}{P}\Big(t - 2(p - q_i)\Big), \tag{12.5}$$

therefore the lsbf of ψ is

$$\texttt{lsbf}^\psi(t) = \sum_{i=1}^{m} \frac{q_i^\psi}{P}\Big(t - 2(p - q_i^\psi)\Big) = \alpha(t - \Delta^\psi), \tag{12.6}$$

where

$$\alpha = \frac{Q}{P}, \tag{12.7}$$

and

$$\Delta^{\psi} = 2\Big(P - \frac{\sum_{i=1}^{m} (q_i^{\psi})^2}{Q}\Big). \tag{12.8}$$

Note that we overload Δ in the rest of the paper and it does not refer to the delay in the bounded delay model anymore. α of all platforms in Ψ^{Γ} are equal, however, their Δ can differ.

Lemma 3. *The budget donation operation always outputs a platform in which its Δ is less than the Δ of its input platform ($\Delta^{\psi^{child}} < \Delta^{\psi^{parent}}$).*

Proof. Assuming that $q_i^{\psi^{parent}}$ and $q_i^{\psi^{child}}$ represent the budget distributions of the input and output platform of the budget donation operation respectively, based on Equation 12.8 we should show:

$$\sum_{i=1}^{m} (q_i^{\psi child})^2 - \sum_{i=1}^{m} (q_i^{\psi parent})^2 > 0. \tag{12.9}$$

Since all budgets except q_d (q of the budget donor) and q_r (q of the budget receiver) are equal we can write:

$$(q_d - 1)^2 + (q_r + 1)^2 - q_d^2 - q_r^2 > 0, \tag{12.10}$$

$$q_r > q_d - 1, \tag{12.11}$$

which is true since $r < d$ and the budgets are sorted non-increasingly. □

Lemma 4. *The* `lsbf` *of the balanced platform (*$\texttt{lsbf}^{\psi_b}$*) is lower than any other possible platforms'* `lsbf`.

Proof. According to Lemma 3, and given that ψ_b is the root of Algorithm 9

$$\Delta^{\psi_b} > \Delta^{\psi_b{}'} \tag{12.12}$$

where $\psi_b{}'$ represent any non-balanced platform derived from Algorithm 9, which according to 12.6 yields to:

$$\texttt{lsbf}^{\psi_b}(t) < \texttt{lsbf}^{\psi_b{}'}(t). \tag{12.13}$$

□

12.4.4 The lsbf of flexible interfaces

According to Lemma 4, the lsbf of a flexible interface ($\texttt{lsbf}^{\Gamma}$) is calculated by deriving the corresponding balanced platform and calculating $\texttt{lsbf}^{\psi_b}$,

$$\texttt{lsbf}^{\Gamma}(t) = \texttt{lsbf}^{\psi_b}(t) = \alpha(t - \Delta^{\psi_b}). \tag{12.14}$$

12.4.5 Upper bound of the sbf

In this section we present an upper bound for the sbf which is used for excluding the irrelevant platforms in calculating the $\texttt{sbf}^{\Gamma}$. The upper bound of the supply bound function (usbf) for independent servers on single processors with a common period P and given budget q_i according to [13] is as follows:

$$\texttt{usbf}(t) = \frac{q_i}{P}\Big(t - (P - q_i)\Big). \tag{12.15}$$

Therefore, $\texttt{usbf}^{\psi}(t)$ is:

$$\texttt{usbf}^{\psi}(t) = \sum_{i=1}^{m} \frac{q_i^{\psi}}{P}\Big(t - (P - q_i^{\psi})\Big) = \alpha(t - \theta^{\psi}), \tag{12.16}$$

where

$$\theta^{\psi} = P - \frac{\sum_{i=1}^{m} (q_i^{\psi})^2}{Q}. \tag{12.17}$$

Lemma 5. *In calculating the* $\texttt{sbf}^{\Gamma}$ *for flexible interfaces it is sufficient to consider the following subset of* Ψ^{Γ}*:*

$$\Psi^{\theta} = \Big\{\forall \psi \in \Psi^{\Gamma} : \quad \Delta^{\psi} \geq \theta^{\psi_b}\Big\}.$$

Proof. Recall step two in calculating $\texttt{sbf}^{\Gamma}$, since we are using the $\min$ function to calculate $\texttt{sbf}^{\Gamma}$ at each time point, the platforms where their sbf are absolutely more than $\texttt{sbf}^{\psi_b}$ do not affect the $\min$ function. Therefore, we want to exclude the platforms that fulfill the following condition:

$$\forall t : \texttt{sbf}^{\psi}(t) > \texttt{sbf}^{\psi_b}(t), \tag{12.18}$$

or

$$\forall t : \texttt{sbf}^{\psi}(t) \geq \texttt{usbf}^{\psi_b}(t), \tag{12.19}$$

which yields to excluding the following set:

$$\forall \psi : \forall t : \texttt{lsbf}^{\psi}(t) \geq \texttt{usbf}^{\psi_b}(t), \tag{12.20}$$

therefore Ψ^{θ} is of our interest in calculating $\texttt{sbf}^{\Gamma}$. □

Based on Lemma 5, Algorithm 9 can be altered such that the stop condition ($\psi' = \psi_p$) is replaced by the following condition:

$$\Delta^{\psi} < \theta^{\psi_b}, \tag{12.21}$$

and using Equation 12.8 we have:

$$\sum_{i=1}^{m}(q_i^{\psi})^2 \geq Q(P - \frac{\theta^{\psi_b}}{2}), \tag{12.22}$$

therefore, the condition at line 9 in Algorithm 9 ($\psi' = \psi_p$) should be replaced by Inequality 12.22. Thereafter, we need to consider all output platforms of the altered algorithm for calculating the $\texttt{sbf}^{\Gamma}$:

$$\texttt{sbf}^{\Gamma}(t) = \min\left\{\texttt{sbf}^{\psi}\right\} \forall \psi \in \Psi^{\theta}. \tag{12.23}$$

For instance lets take the flexible interface $\Gamma_1 = \langle 2, 8, 8\rangle$ presented in Example 1. For this example we have: $\theta^{\psi_b} = 4$, hence the stop condition is: $\sum_{i=1}^{m}(q_i^{\psi})^2 \geq 48$. Therefore, ψ_3 $\left(\sum_{i=1}^{m}(q_i^{\psi_3})^2 = 50\right)$ and ψ_p $\left(\sum_{i=1}^{m}(q_i^{\psi_p})^2 = 64\right)$ do not need to be considered for calculating the $\texttt{sbf}^{\Gamma}$:

$$\texttt{sbf}^{\Gamma_1}(t) = \min\left\{\texttt{sbf}^{\psi_b}(t), \texttt{sbf}^{\psi_1}(t), \texttt{sbf}^{\psi_2}(t)\right\}.$$

12.5 Approximate sbf of the flexible interfaces

According to 12.23, we can reduce the number platforms that have to be investigated when calculating the $\texttt{sbf}^{\Gamma}$, however, this subset (Ψ^{θ}) may still include too many platforms that makes the computations intractable. In this section we propose an approach to derive an approximate supply bound function for the flexible interfaces (asbf). In this approach we consider λ such that $\theta^{\psi_b} \leq \lambda \leq \Delta^{\psi_b}$ and we replace θ^{ψ_b} with λ in Equation 12.22 to get a new termination condition for branching ($\psi' = \psi_p$) in Algorithm 9:

$$\sum_{i=1}^{m}\left(q_i^{\psi}\right)^2 \geq Q(P - \frac{\lambda}{2}). \tag{12.24}$$

Therefore, in this approach we consider the following subset of Ψ^{Γ}:

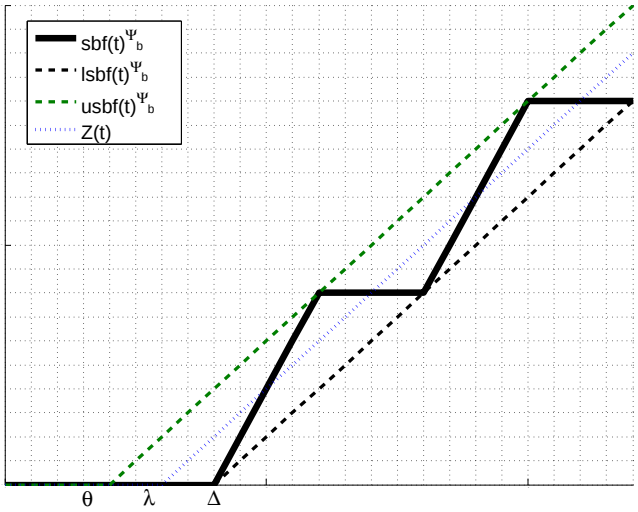


Figure 12.4: The `sbf`, the `lsbf` and the `usbf` of a balanced platform

$$\Psi^{\lambda} = \Big\{ \forall \psi \in \Psi^{\Gamma} : \quad \Delta^{\psi} \geq \lambda \Big\}.$$

This new condition confines the search space, therefore we can get an approximate `sbf` investigating a lower number of platforms. Using λ we cut the tree in earlier branches than the original algorithm, therefore, the time complexity of the algorithm is reduced. Figure 12.4 illustrates the relation between the actual upper bound and the approximate upper bound ($Z(t)$). For calculating the $\texttt{sbf}^{\Gamma}$ we exclude the platforms where the `sbf` is located at the left hand side of $\texttt{usbf}^{\psi_b}$, however, for calculating the $\texttt{asbf}^{\Gamma}$ we exclude all the platforms where the `sbf` is located at the left hand side of $Z(t)$.

The approximate `sbf` is:

$$\texttt{asbf}^{\Gamma}(t) = \min \Big\{ \texttt{sbf}^{\psi}(t), Z(t) \Big\} \quad \forall \psi \in \Psi^{\lambda}, \tag{12.25}$$

where

$$Z(t) = \alpha(t - \lambda), \tag{12.26}$$

because according to Lemma 3, all other platforms that we are not considering in the $\min$ function have smaller Δ which means their `lsbf` (and consequently

Γ	$\langle 8, 16, 40\rangle$	$\langle 4, 64, 80\rangle$
Ψ^{Γ}	6360	4089
Ψ^{θ}	5650($\simeq 88\%\Psi^{\Gamma}$)	3652($\simeq 89\%\Psi^{\Gamma}$)
ψ^{λ_1}	2259($\simeq 35\%\Psi^{\Gamma}$)	2245($\simeq 54\%\Psi^{\Gamma}$)
ψ^{λ_2}	507($\simeq 7\%\Psi^{\Gamma}$)	938($\simeq 22\%\Psi^{\Gamma}$)

Table 12.1: Ψ^{Γ}, Ψ^{θ} and Ψ^{λ} for sample flexible interfaces ($\lambda_1 = 0.5(\theta + \Delta)$ and $\lambda_2 = 0.75(\theta + \Delta)$).

their sbf) are more than $Z(t)$ at all time points. Note that the min operation in Equitation 12.25 is critical in ensuring that the approximation is safe.

Recall Example 1, if we assign $\lambda = 6$, the termination condition is $\sum_{i=1}^{m}(q_i^{\psi})^2 \geq 40$, therefore, for calculating $\texttt{asbf}^{\Gamma_1}$ we only need to consider ψ_b and ψ_1 together with the following line $Z_1(t) = (t - 6)$. Hence, we have:

$$\texttt{asbf}^{\Gamma_1}(t) = \min\Big\{\texttt{sbf}(t)^{\psi_b}, \texttt{sbf}(t)^{\psi_1}, Z_1(t)\Big\}.$$

Table 12.1 shows two flexible interfaces and corresponding number of rigid interfaces that should be investigated in order to calculate the $\texttt{sbf}^{\Gamma}$ and the $\texttt{asbf}^{\Gamma}$. In these two examples, more than 10% of the possible platforms are irrelevant in calculating $\texttt{sbf}^{\Gamma}$, and when calculating $\texttt{asbf}^{\Gamma}$ the more the number of the platforms included in the $\min$ function, the higher the accuracy of the approximation.

The number of possible platforms of a flexible interface is positively correlated with P and m because when increasing them, there are more possibilities for the total budget to be distributed on different processors. However, increasing Q does not necessarily increases the number of possible platforms. For instance when $Q = m \times P$, we have $\psi_b = \psi_p$ and the number of possible platforms is one. Figure 12.5 shows the relation between Q and the number of possible platforms for the flexible interface $\Gamma = \langle 5, 16, Q\rangle$ ($Q \in [1, m \times P]$). The figure indicates that the number of possible platforms increases until $Q = \frac{m\times P}{2}$, and decreases afterwards. The trend is analogous for all flexible interfaces, which can be explained by the longest depth (κ) presented in Equation 12.1, where increasing Q has two consequences: i) increases q_i in ψ_b which increases κ ii) increases h ($h = \lfloor \frac{Q}{P} \rfloor$) and therefore decreases κ. Hence, depending on the dominate factor, the number of possible platforms may either be positively or negatively correlated with Q. From the figure we observe that the dominant factor is (i) until $Q = \frac{m\times P}{2}$ and thereafter

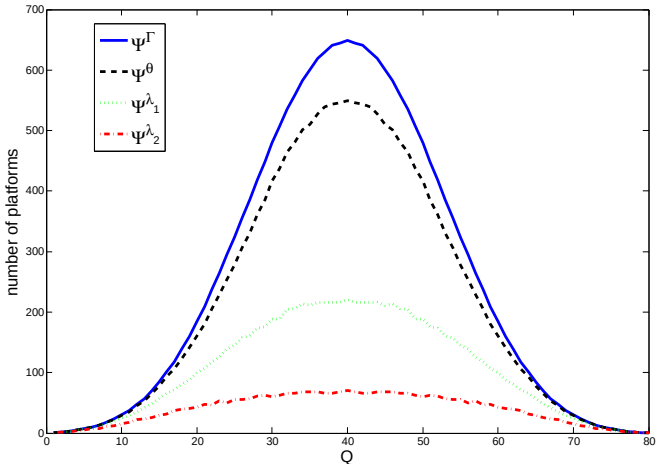


Figure 12.5: Number of platforms in Ψ^{Γ}, Ψ^{θ} and Ψ^{λ} for $\Gamma = \langle 5, 16, Q \rangle$ ($\lambda_1 = 0.5(\theta + \Delta)$ and $\lambda_2 = 0.75(\theta + \Delta)$).

it is (ii), therefore the difference between the number of platforms in calculating `asbf` and `sbf` is more significant in platforms where Q is around $\frac{m \times P}{2}$.

12.6 Conclusion

In this paper we presented an approach for calculating the supply bound function of multiprocessor periodic resource interfaces when the servers are not synchronized. Being independent from server synchronization makes the model compatible with all types of hardware platforms (even with the ones where synchronization is expensive). Furthermore, due to the exponential complexity of calculating the actual supply bound function, we proposed an approach for calculating an approximate supply bound function with lower computational complexity.

The next step in our work is to evaluate the difference between using actual and approximate supply bound functions using an extensive number of randomly generated systems. We also intend to compare the periodic resource interface with the bounded delay interface. Finally we will look into the presented algorithm for mapping the flexible interface to the rigid interface(s) and try to further reduce its complexity using some heuristics.

References

[1] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium (RTSS'90)*, pages 182 –190, December 1990.

[2] A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[3] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*, pages 249–258, December 2010.

[4] H. Zhu, S. Goddard, and M. B. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In *32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 239–248, December 2011.

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

[6] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 256–267, December 1999.

[7] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, May 2001.

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

[9] E. Bini, G. Buttazzo, and M. Bertogna. The multi supply function abstraction for multiprocessors. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, (RTCSA'09)*, pages 294–302, August 2009.

[10] E. Bini, M. Bertogna, and S. Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, (RTSS'09)*, pages 437–446, December 2009.

[11] H. Leontyev and J. H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 191–200, July 2008.

[12] H. S. Wilf. *Lectures on Integer Partitions*, July 2000. Available at `http://www.math.upenn.edu/~wilf/PIMS/PIMSLectures.pdf`.

[13] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium, (RTSS'07)*, pages 129–138, December 2007.

Chapter 13

Paper F: On Component-Based Software Development for Multiprocessor Real-Time Systems

Nima Khalilzad, Moris Behnam and Thomas Nolte.
In Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'15), August 2015.

Abstract

Component-based software development provides a modular approach to develop complex software systems. In the context of real-time systems, it is desirable to abstract the timing properties of software components using an interface for each component. The timing properties of the whole system, composed of multiple components, is studied using the component interfaces. In this paper we focus on periodic interface models. In the case of components developed for single processor platforms, for examining the system schedulability, the interfaces can be regarded as periodic tasks. Thus, making it possible to use the conventional schedulability analyses for the system level schedulability test. In the case of components developed for multiprocessors, since interfaces may have utilization larger than 100 % of a single processor, it is not possible to directly use the component interfaces for the system schedulability test. Therefore, the interfaces have to be decomposed before performing the system level schedulability test.

In this paper, we target the special case of partitioned EDF for scheduling the components integrated on a multiprocessor. Therefore, the system level schedulability test is equivalent to finding a feasible allocation of component interfaces on the multiprocessor. We propose two algorithms for allocating the multiprocessor periodic interfaces. In addition, we propose an orthogonal approach for developing component-based real-time systems on multiprocessors in which components with utilization more than 100 % of a single processor are divided into smaller subcomponents before abstracting their interfaces. We show, through extensive evaluations, that our alternative approach significantly reduces the interface overhead.

13.1 Introduction

Multiprocessor platforms provide a great amount of computational capacity on a single hardware. Therefore, it is possible to design and run large software systems on a single chip. Component-based software development facilitates the development process of large software systems. We consider component models in which a real-time software component is composed of multiple real-time tasks. In this approach, software components are developed independently, possibly by different teams, and later integrated. In the real-time systems arena, component-based development approaches (e.g. [1, 2]) often follow a two step process. Firstly, the processor demand of the tasks within each component is abstracted. This step produces component interfaces. Secondly, the components are integrated and their schedulability is studied using the component interfaces. Abstracting the requirements of components comes at a price. There is often a gap between the processor utilization of the component interfaces and the utilization of the task set within components. This gap (henceforward referred as the abstraction overhead) results in a processor utilization efficiency loss. Efficient utilization of the processor resource is particularly important in resource constrained embedded systems. To this end, it is important to study the abstraction overhead of different approaches to understand their practical applicability.

In this paper we focus on a periodic interface model, namely the Multiprocessor Periodic Resource (MPR) model [3]. The reason behind focusing on the periodic models is that they can easily be implemented in practice. The schedulability of systems, composed of multiple components, is investigated through studying the MPR interfaces. It is desirable to use the same schedulability techniques used for studying the schedulability of real-time tasks, and investigate the schedulability of the components. However, the task schedulability tests cannot be directly applied to the components for which their interface utilizations are more than 100 % of a single processor (i.e. one). This is because the basic assumption in all of the schedulability tests is that the task utilization is less than or equal to one. Therefore, components with interface utilization more than one have to be decomposed to smaller subcomponents with utilization less than or equal to one. The component schedulability test, then, can be performed using the decomposed subcomponent interfaces. In this paper we use the partitioned Earliest Deadline First (pEDF) [4] algorithm for scheduling the components. We propose two algorithms which perform decomposition and allocation simultaneously, each algorithm following a different objective.

In all of the proposed approaches for developing component-based real-

time systems on multiprocessors (e.g. [3, 5, 2]) the component decomposition is performed after abstracting the components. In this paper, we investigate an alternative approach. We first decompose components for which their utilization is more than one. Thereafter, we abstract the component processor requirements using an abstraction technique proposed in [6]. We show that, using extensive simulations, performing the decomposition before abstraction significantly reduces the abstraction overhead. Also, we provide three integration algorithms for components abstracted using our approach, i.e., decomposed before the abstraction. Finally, using extensive simulations, we compare the number of processors required for integrating the components developed using the two alternative approaches. We compare the performance of the proposed integration algorithms within each approach.

Contributions. In this paper we study the complete process of component-based development approaches for real-time systems (focusing only on timing properties) from component abstraction to the system integration using periodic interfaces. We present the following contributions. (i) We propose two integration algorithms for integrating components abstracted using the MPR model. (ii) We propose a new approach in which component decomposition is performed before abstraction. We propose a new interface model as well as three integration algorithms for this new approach. (iii) We present the result of our extensive simulations comparing the approach based on the MPR abstraction model with our alternative approach.

13.2 System model and development approaches

System and task model. We assume a multiprocessor platform with m homogeneous processors. n components are composed on the multiprocessor platform. The slack bandwidth of the j^{th} processor is denoted using S_j. We assume a constrained deadline periodic task model in which the k^{th} task τ_k is characterized using period $T_k \in \mathbb{N}^+$, deadline $D_k \in \mathbb{N}^+$ and Worst-Case Execution Time (WCET) $C_k \in \mathbb{N}^+$ ($C_k \leq D_k \leq T_k$). We assume that the tasks are independent, i.e., except the processor resource, they do not share any other resources.

Scheduling scheme. We assume a hierarchical scheduling scheme in which the scheduling is performed in two levels. At the global level components are scheduled using a *component-scheduler*. In this paper we use pEDF for scheduling the components. Within the components, however, a *task-scheduler* coordinates the execution of the tasks. In the case that a component is assigned

to one processor we use EDF as the task-scheduler. When a component is spread over multiple processors, we use global EDF (gEDF) [4] as the task-scheduler. From a resource provisioning vantage point, the multiprocessor resource is partitioned in the time domain. Each component is assigned to a multiprocessor partition which indeed provisions a fraction of the multiprocessor resource to the component. The components, then, distribute their fraction of the resource among their inner tasks.

Component-based system development. Component-based development approaches often consider the following two roles for the system development: (i) component developer (ii) system integrator. The component developers develop a task set, and they select a suitable task-scheduler. They also calculate the component interface based on the task set and the task-scheduling algorithm. The component interface indicates the amount and the specifications of the required processor resource fraction. This approach enables independent development of components by different development teams. The system integrator, on the other hand, receives a set of component interfaces. The system integrators use the component interfaces to examine the schedulability of the system. If the component requires a processor fraction more than one, then the integrator divides the component into a number of subcomponents. This step is referred as the transformation of interfaces into interface-tasks in the previous approaches (e.g. [3, 2]). The reason behind performing this transformation is that it is desirable to use the conventional task schedulability analyses for examining the schedulability of the system. The basic assumption in such analyses is that the utilization of tasks is less than or equal to one. Therefore, in order to perform schedulability test using the interfaces, we require interfaces in which their utilization is less than or equal to one. We use the word "decomposition" to refer to the step in which a large component is divided into a number of smaller subcomponents. After the decomposition step, the interfaces of the subcomponents can be used for performing the schedulability test.

Component model. We assume that component $\mathcal{C}_i$ is composed of a set of tasks denoted by $\mathcal{T}_i$. We use $U_{\mathcal{T}_i}$ to denote the task set utilization of $\mathcal{C}_i$. The components are assigned to the processors at the integration phase. We use $\rho_{i,j}$ to denote the amount of the utilization of $\mathcal{C}_i$ that is allocated on the j^{th} processor. In this paper we target components for which their utilizations are more than one $U_{\mathcal{T}_i} > 1$, i.e. they require more than one processor for performing their computations. The following two alternative approaches can be used when dealing with such components. (i) First Abstraction, then Decomposition (FAD): in this approach the component developers first abstract the

resource requirements of the entire component. The system integrators have to divide the component into a number of subcomponents at the integration phase.(ii) First Decomposition, then Abstraction (FDA): the component developers first divide the component into a number of subcomponents such that all subcomponents have utilization less than or equal to one. The subcomponent interfaces are then derived by the component developers.We use $\mathcal{C}_{i,r}$ to denote the r^{th} subcomponent of component i. Similar to the components, we use $\rho_{i,r,j}$ to denote the amount of the utilization of $\mathcal{C}_{i,r}$ that is allocated on the j^{th} processor. In the following we explore the above two alternatives and we compare the implications of using each approach.

The FAD approach. In this approach the processor requirements of a component is abstracted using a single interface. The interface essentially indicates the fraction of the multiprocessor capacity required by the component. In doing so, it is assumed that the tasks within one component are allowed to migrate among processors, i.e., they are scheduled using a global multiprocessor scheduling policy. For instance, in the approach proposed by Easwaran *et al.* [3], the MPR model is used for abstracting the processor requirements of the components. In this model the interface of the i^{th} component is denoted by $\Gamma_i^{m'_i} < \Pi_i, \Theta_i >$ where $\Pi_i \in \mathbb{N}^+$, $\Theta_i \in \mathbb{N}^+$ and $m'_i \in \mathbb{N}^+$ characterize the period, total budget and the parallelism level of the component. The parallelism level indicates the maximum number of processors that can contribute in providing the total budget to $\mathcal{C}_i$. The MPR interface imposes the following constraints by definition: $1 \leq m'_i \leq m$ and $\Theta_i \leq m'_i \times \Pi_i$. The fraction of the required multiprocessor, i.e the interface utilization, is denoted using:

$$U_{\Gamma_i^{m'_i}} = \frac{\Theta_i}{\Pi_i}.$$

This model provides a great deal of flexibility at the integration phase. This is because the total utilization can be provided using any m'_i processors. Therefore, the integrators can use the processors' slack status to decide on how to perform the decomposition. For instance, assume that we have two processors with the following slacks $S_1 = S_2 = 0.55$. Suppose that a new component is being integrated with $U_{\Gamma_i^{m'_i}} = 1.1$ and $m'_i = 2$. The only decomposition that can deem the system schedulable, is to divide the component into two subcomponents each with utilization equal to 0.55. The only problem with this model is that it incurs a considerable amount of abstraction overhead (see Section 13.4). Therefore, in the following we investigate an alternative approach in which we allow the system integrators to trade-off the integration flexibility with the abstraction overhead.

The FDA approach. In this approach the component developers are responsible to decompose the components, for which their utilization is more than one, into a number of subcomponents. The system integrator, then, can directly use the subcomponent interfaces to examine the schedulability of the system. We use the Periodic Resource (PR) [6] model for abstracting the processor requirements of the subcomponents. The PR model can be seen as a special case of the MPR model where $m'_i = 1$. Note that the fact that $m'_i = 1$ allows us to use a different analysis (i.e. single processor EDF schedulability) for deriving the subcomponent interfaces. The FDA approach does not provide any flexibility at the integration phase. This is because the utilization required by one subcomponent has to be provided using exactly one processor ($m'_i = 1$). For instance, assume that, similar to the previous example, we have two processors with the following slacks $S_1 = S_2 = 0.55$. The component developer has decomposed its large component into two subcomponents with utilizations equal to 0.65 and 0.45. Although the total component utilization is equal to the overall processor slack, it is not possible for the integrator to deem the system schedulable. This is because the decomposition is already performed before the abstraction, and the integrator has to perform the integration using the provided subcomponents.

The fact that the FDA approach does not provide flexibility at the integration phase may result in processor utilization loss. In order to mitigate this problem, we propose an altered modeling approach. In the new approach, after decomposing the large components, the component developer uses the PR model to abstract subcomponents' processor requirements. In addition, assuming that it may be impossible to fit one subcomponent in one processor at the integration phase, the component developer derives the MPR model for the subcomponents assuming $m' \in [2, m]$. We refer to this model as the Extended Periodic Resource (EPR) model in the rest of the paper. In the EPR model the component interfaces are denoted using the following matrix:

$$\Omega_i = \begin{bmatrix} \Gamma^1_{i,1} & \Gamma^1_{i,2} & \cdots & \Gamma^1_{i,p_i} \\ \Gamma^2_{i,1} & \Gamma^2_{i,2} & \cdots & \Gamma^2_{i,p_i} \\ \vdots & \vdots & \ddots & \vdots \\ \Gamma^m_{i,1} & \Gamma^m_{i,2} & \cdots & \Gamma^m_{i,p_i} \end{bmatrix},$$

where $\Gamma^j_{i,r}$ denotes the MPR interface of $\mathcal{C}_{i,k}$ given that its parallelism is equal to j. p_i represents the total number of subcomponents of $\mathcal{C}_i$. p_i depends on the decomposition algorithm which is addressed later in this section. The bud-

get and the period of $\Gamma_{i,r}^{j}$ is denoted using $\Theta_{i,r}^{j}$ and $\Pi_{i,r}^{j}$ respectively. Ω_i allows integrators to select an interface which has a suitable parallelism level considering the processor slacks. If the slacks are scattered throughout the processors, then it may be beneficial to use an interface with a large parallelism level. However, this additional flexibility comes at a price. As it is shown in [3], increasing the parallelism level increases the utilization of the MPR interfaces. We use $\Delta_{i,k}^{j}$ to denote the difference of the utilization required by subcomponent $\mathcal{C}_{i,k}$ in parallelism level j and $j-1$, i.e.:

$$\Delta_{i,k}^{j} = \frac{\Theta_{i,r}^{j}}{\Pi_{i,r}^{j}} - \frac{\Theta_{i,r}^{j-1}}{\Pi_{i,r}^{j-1}},$$

where $\forall j < 1 \ \Theta_{i,k}^{j} = 0$. Informally speaking, $\Delta_{i,k}^{j}$ denotes the amount of penalty that needs to be paid for gaining an additional level of flexibility.

The component decomposition algorithm takes one component $\mathcal{C}_i$ and decomposes it into a set of subcomponents $\{\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,p_i}\}$. We use the following three bin packing heuristics for component decomposition: First Fit (FF), Best Fit (BF) and Worst Fit (WF) [4]. In the case of the WF heuristic, we assume that we have $\lceil U_{\mathcal{T}_i} \rceil$ available processors (i.e. $p_i = \lceil U_{\mathcal{T}_i} \rceil$). If the decomposition fails, then we add a new processor and reperform the decomposition.

13.3 Integration

In this section we present a number of algorithms for integrating components with MPR interfaces as well as components with EPR interfaces. The input to the integration problem is a set of component interfaces. A solution to the integration problem is a set of processor allocations such that (i) the sum of all allocations on each processor is less than or equal to one since we use pEDF for scheduling components; (ii) the constraints specified in the component interfaces are met. In the following we explain the integration algorithms corresponding to each interface model in detail.

13.3.1 MPR composition

In the following we formally define the integration problem of the FAD approach in which the components are abstracted using the MPR interface model. This problem is similar to that of what is found in the problem of *bin packing*

with fragmentable items. In this variation of the bin packing problem, a number of items have to be packed into a set of bins. It is possible to divide items into smaller chunks. The item division does not incur any overhead, i.e., the sizes of the items do not increase by dividing them. The objective is to minimize the number of fragments when placing the items into the bins. In [7] Bertrand *et al.* proved that this variation of the bin packing problem is strongly NP-complete. In our MPR integration problem the items are the MPR interfaces and the processors are the bins. However, our problem is slightly more complex than the above bin packing problem in the following aspects. Instead of minimizing the number of fragments, our objective is to find an allocation in which the number of fragments of all items is less than or equal to their corresponding parallelism level m'_i. In the following we present a mathematical formulation of the constraints of the MPR integration problem:

$$\sum_{i=1}^{n} \rho_{i,j} \leq 1 \qquad \forall j \in [1 \dots m], \tag{13.1a}$$

$$\sum_{j=1}^{m} \rho_{i,j} = U_{\Gamma_i^{m'_i}} \qquad \forall i \in [1 \dots n], \tag{13.1b}$$

$$\sum_{j=1}^{m} f_{i,j} \leq m'_i \qquad \forall i \in [1 \dots n], \forall j \in [1 \dots m], \tag{13.1c}$$

$$f_{i,j} \in \{0,1\}, \rho_{i,j} \in \mathbb{Z}_{\geq 0}, \tag{13.1d}$$

where $\rho_{i,j}$ is the amount of $U_{\Gamma_i^{m'_i}}$ allocated on processor j. $f_{i,j}$ is equal to one when $\rho_{i,j} > 0$, i.e., $\Gamma_i^{m'_i}$ is partially allocated to processor j.

The FAD approach postpones the decomposition to the integration phase. Therefore, we provide two algorithms in which decomposition and allocation is performed simultaneously. In these algorithms each component is treated separately. The component decomposition is performed based on the current status of the slack utilizations on the multiprocessor. We present two algorithms referred as *compact integration* and *balanced integration*. In the compact integration algorithm, the objective at each step is to (i) use a minimum number of the processors (ii) use processors that already have other components assigned on them. The compact integration algorithm is presented in Algorithm. 10. We first sort processors based on increasing slacks. The next step is to find the first m''_i processors which can accommodate the current component, where $m''_i \leq m'_i$. Line 2 returns the index of the first processor in the set that can accommodate the component. Once the processors are sorted it is easy to find m''_i. Function `findFirstProcessors`(m'_i, $U_{\Gamma_i^{m'_i}}$) loops through the processors starting from the first processors. At each iteration, the following sum

Algorithm 10: MPR compact integration.

Require: Γ_i
Ensure: matrix of processor allocations $\{\rho\}$ or failure.

```
1: sortProcessorsIncreasingSlack();
2: j = findFirstProcessors(m'_i, U_{Γ_i^{m'_i}});
3: if j < 0 then
4:    return FALSE;
5: end if
6: 𝔘 = U_{Γ_i^{m'_i}};{Unallocated utilization}
7: while 𝔘 > 0  and  j ≤ m do
8:    ρ_{i,j} = max(S_j, 𝔘);
9:    𝔘 -= ρ_{i,j};
10:   j++;
11: end while
12: if 𝔘 = 0 then
13:   return {ρ};
14: else
15:   return FALSE;
16: end if
```

is calculated: $\sum_{j}^{j+m'_i} S_j$. If the above sum is more than the utilization of the current component being integrated $\mathcal{C}_i$, then `findFirstProcessors` returns the current processor index j. If this function fails to find the candidate set of processors, then it returns -1 and the algorithm returns failure. Otherwise, the algorithm starts filling each processor until either the processor is full or the component is completely allocated. This algorithm is called for all components. The while loop Lines 7 to 11 has at most m iterations. Therefore, since sorting the processors and finding the first processor can be done in polynomial time, the entire algorithm runs in polynomial time. The exact complexity, however, depends on the particular implementations of the sort algorithm.

We present an alternative algorithm for integrating components with MPR interfaces in Algorithm 11. This algorithm is referred as balanced integration. The objective in this approach is to evenly distribute the slack at each step. The algorithm first sorts the processors based on decreasing slack. Thereafter, it finds the first m''_i processors that can fit the components, where $m''_i \leq m'_i$. Once the m''_i target processors are selected, the algorithm calculates the tar-

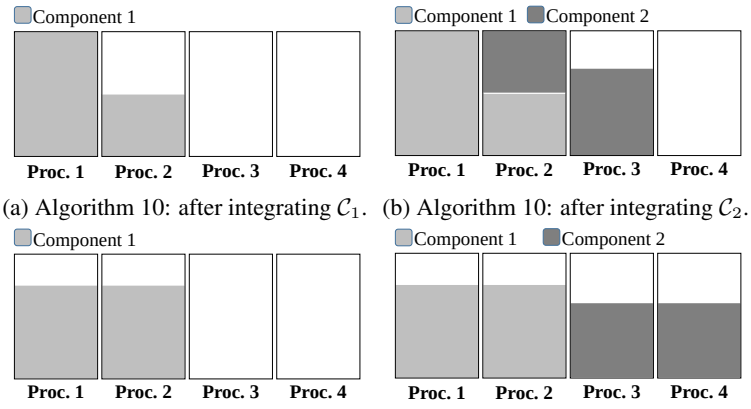
Algorithm 11: MPR balanced integration.

Require: Γ_i.
Ensure: matrix of processor allocations $\{\rho\}$ or failure.

```
1: sortProcessorsDecreasingSlack();
2: j = findFirstProcessors(m'_i, U_{Γ_i^{m'_i}});
3: if j < 0 then
4:    return FALSE;
5: end if
6: S^T = Σ_{i=j}^{j+m'_i} S_i;
7: 𝔘 = U_{Γ_i^{m'_i}};{Unallocated utilization}
8: while 𝔘 > 0  and  j ≤ m do
9:    ρ_{i,j} = max(S_j − S^T, 𝔘);
10:   𝔘 -= ρ_{i,j};
11:   j++;
12: end while
13: if 𝔘 = 0 then
14:   return {ρ};
15: else
16:   return FALSE;
17: end if
```

get slack S^T on each processor. Finally, it fills each processor until its target slack is reached. Similar to the compact integration algorithm, the balanced integration algorithm also runs in polynomial time.

We present an example for elaborating the above two algorithms. Assume that we want to integrate two components with the following interface utilizations: $U_{\Gamma_1^2} = 1.5$ and $U_{\Gamma_2^2} = 1.2$. Assuming that we start with $\mathcal{C}_1$, the compact integration algorithm decomposes the interface into two subcomponents with utilizations equal to one and 0.5. The result of this step is illustrated in Figure 13.1a. Thereafter, $\mathcal{C}_2$ is integrated. At this stage the `findFirstProcessors` function returns processor 2 because $\mathcal{C}_2$ fits in the slack utilization of processor two and three. $\mathcal{C}_2$ is decomposed into two subcomponents with utilizations equal to 0.5 and 0.7 (Figure 13.1b). On the other hand, the balanced integration divides $\mathcal{C}_1$ into two identical subcomponents with utilizations equal to 0.75, and it allocates them on the first two processors. The result of this step is illustrated in Figure 13.1c. When integrating $\mathcal{C}_2$, the `findFirstProcessors` function returns three because the overall slack



(a) Algorithm 10: after integrating $\mathcal{C}_1$. (b) Algorithm 10: after integrating $\mathcal{C}_2$.

(c) Algorithm 11: after integrating $\mathcal{C}_1$. (d) Algorithm 11: after integrating $\mathcal{C}_2$.

Figure 13.1: The steps of the two MPR integration algorithms.

on processors $\{1,2\}$ and $\{2,3\}$ is not enough for integrating $\mathcal{C}_2$. The algorithm, then, divides $\mathcal{C}_2$ into two subcomponents with identical utilizations 0.6, and allocates them on the third and forth processors (Figure 13.1d). As illustrated in Figure 13.1, the balanced integration algorithm resulted in fragmented slacks, while the compact integration resulted in one entirely free processor and one partially free processor.

13.3.2 EPR integration

In the following we formally define the integration problem of the FDA approach in which the components are abstracted using the EPR interface model. This problem is analogous to the problem of *bin packing with size-increasing fragmentation*. In this variation of the bin packing problem the items are allowed to be fragmented while fragmenting an item is associated with a cost. Menakerman and Rom [8] showed that this problem is also NP-hard. The EPR integration problem is more complex because instead of a fixed fragmentation cost, the fragmentation cost varies for different subcomponents. For components with parallelism level equal to one, the integration algorithm only has to allocate subcomponents on the multiprocessor. This problem is equivalent to partitioning implicit deadline periodic tasks on multiprocessors. However, if the allocation fails, the integration algorithm can fragment a subcomponent while adding a fragmentation cost. The fragmentation cost for $\Gamma_{i,r}^{j}$ is denoted

using $\Delta_{i,r}^{j}$. Since we treat each subcomponent separately, solving the EPR integration problem for one component is equivalent to solving this problem for all components in the systems. For notational convenience, we drop the component index when referring to the EPR interfaces in the rest of this section. Let q_r be the parallelism level of subcomponent $\mathcal{C}_r$, and let $\mathcal{Q}$ be the set of parallelism levels $\mathcal{Q} = \{q_1, \ldots, q_p\}$. Also, assume that the total number of subcomponents is represented using n'. The EPR integration problem is to find $\mathcal{Q}$ and allocations such that:

$$\sum_{r=1}^{n'} \rho_{r,j} \leq 1 \qquad \forall j \in [1 \ldots m], \tag{13.2a}$$

$$\sum_{j=1}^{m} \rho_{r,j} = U_{\Gamma_r^{q_r}} \qquad \forall r \in [1 \ldots n'], \tag{13.2b}$$

$$\sum_{j=1}^{m} f_{r,j} \leq q_r \qquad \forall r \in [1 \ldots n'], \forall j \in [1 \ldots m], \tag{13.2c}$$

$$f_{r,j} \in \{0,1\}, \rho_{r,j} \in \mathbb{Z}_{\geq 0}, \tag{13.2d}$$

The EPR integration algorithm is presented in Algorithm 12. First the subcomponents are sorted based on decreasing first parallelism utilizations ($m'_r = 1$). The algorithm assigns the parallelism levels of all subcomponents to one in Line 2. The `isfeasible(`$\mathcal{Q}$`)` function is called in Line 4. This function performs the following utilization test based on the current parallelism levels, i.e. $\mathcal{Q}$:

$$\sum_{r=1}^{n'} U_{\Gamma_r^{q_r}} \leq m. \tag{13.3}$$

The algorithm loops through all subcomponents in Line 5. In Line 6, the algorithm calls the `allocate` function. The following two cases may happen: (i) parallelism level equal to one; (ii) parallelism level more than one. In the case of parallelism level equal to one, the allocation is similar to allocating implicit deadline periodic tasks on multiprocessors. We use different versions of the allocation function, each version implementing a different bin packing heuristic. In the evaluations we present the result of using the following three heuristics: FF, BF and WF. In the case of parallelism more than one, however, we use the MPR integration algorithms for allocating the subcomponents on the multiprocessor. If the allocation fails, then the algorithm calls the `IncreaseFlexibility(`$\mathcal{Q}$`)` function. This function selects one subcomponent, and it increments its parallelism level. It selects the subcomponent which has the smallest $\Delta_r^{q_r}$. In other words, it selects a subcomponent that provides one extra level of flexibility with a minimum overhead penalty. Since the `IncreaseFlexibility` function only increases the parallelism levels,

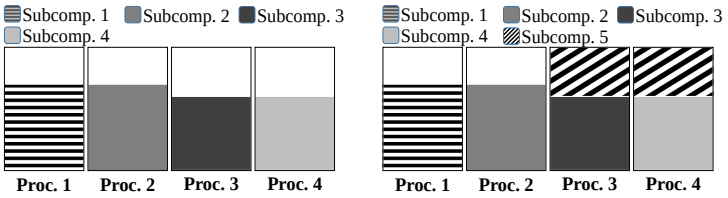
Algorithm 12: EPR integration.

Require: An EPR interface Ω_i.
Ensure: matrix of processor allocations $\{\rho\}$ or failure.

```
1: sortInterfaces(); {Based on U_{Γ_r^1}}
2: ∀r ∈ [1, n'] q_r ← 1;
3: FLAG ← FALSE;
4: while FLAG = FALSE and isfeasible(Q) do
5:    for r = 1; r < n'; k++ do
6:       FLAG ← allocate(Γ_k^{q_r});
7:       if FLAG = FALSE then
8:          Q ← IncreaseFlexibility(Q);
9:          break;
10:      end if
11:   end for
12: end while
13: return FLAG;
```

in the worst-case the algorithm tries $n' \times m$ different $\mathcal{Q}$. However, in our evaluations, we observed that the `isfeasible` function detects the infeasibility in the early stages and it terminates the algorithm. For each $\mathcal{Q}$, the algorithm calls an allocation heuristic which runs in polynomial time. Thus, the EPR integration algorithm runs in polynomial time.

We present an example to further elaborate the EPR integration algorithm. Suppose we want to integrate five subcomponents with the following utilization for their first parallelism level: $U_{\Gamma_1^1} = U_{\Gamma_2^1} = 0.7$, $U_{\Gamma_3^1} = U_{\Gamma_4^1} = 0.6$ and $U_{\Gamma_5^1} = 0.5$. Also, assume that the second parallelism utilizations are as follows: $U_{\Gamma_1^2} = U_{\Gamma_2^2} = 0.9$, $U_{\Gamma_3^2} = U_{\Gamma_4^2} = 0.85$ and $U_{\Gamma_5^2} = 0.8$. We call Algorithm 12 for all subcomponents, starting from $\mathcal{C}_1$. $\mathcal{C}_1$ to $\mathcal{C}_4$ are allocated to processor one to four respectively. The result of integrating the first four subcomponents is illustrated in Figure 13.2a. When integrating $\mathcal{C}_5$, the allocation cannot be performed using the first level parallelism. Therefore, the algorithm calls the `IncreaseFlexibility` function, and it increases the parallelism level of $\mathcal{C}_5$ to two. Thereafter, $\mathcal{C}_5$ is divided into two chunks and it is allocated on the third and forth processors (Figure 13.2b).



(a) Algorithm 12: after integrating $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ and $\mathcal{C}_4$.

(b) Algorithm 12: after integrating $\mathcal{C}_5$.

Figure 13.2: The steps of the EPR integration algorithm.

13.4 Evaluations

In this section we present two types of evaluations. In the first set of evaluations our aim is to compare the abstraction overhead of the MPR model against the abstraction overhead of the EPR model. We used the interface calculation method presented in [3] to calculate optimal MPR interfaces. Note that the optimal MPR interfaces have the minimum possible parallelism level. The abstraction overhead of $\mathcal{C}_i$ abstracted using the MPR model is calculated using the following equation:

$$O_i^{\Gamma} = 100 \times \frac{U_{\Gamma_i^{m_i'}} - U_{\mathcal{T}_i}}{U_{\mathcal{T}_i}}, \tag{13.4}$$

where O_i^{Γ} represents the percentage of abstraction overhead of $\mathcal{C}_i$ abstracted using the MPR model. In addition, for the EPR model, we calculate the abstraction overhead only for parallelism level equal to one because the EPR integration algorithm tries to use the first level parallelisms. We have:

$$O_i^{\Omega} = 100 \times \frac{\sum_{r=1}^{p_i} \Gamma_{i,r}^1 - U_{\mathcal{T}_i}}{U_{\mathcal{T}_i}}, \tag{13.5}$$

where O_i^{Ω} represents the percentage of the first parallelism level abstraction overhead of $\mathcal{C}_i$ abstracted using the EPR model. In the second set of simulations we intend to answer the following question: "given a set of components, which combination of the abstraction models and integration techniques requires the lowest number of processors for composing the component set"?

Simulation setup. We generated components with specific task set utilizations. Each task is assigned to a random period between 100 and 200. The

utilization of τ_i is selected randomly using a uniform distribution between zero and the maximum allowed task utilization. Except one evaluation in which we varied the maximum allowed task utilization, in the rest of the experiments this parameter was set to 0.9. The execution time of tasks is derived by multiplying the period and the utilization. We assigned deadlines equal to the periods for all evaluations. For generating task sets with a target utilization, we kept generating tasks until the remainder utilization was less than the maximum allowed task utilization 0.9. Then we generated the last task with the remainder utilization. Except one evaluation in which we varied the component periods, we set $\Pi_i = 50$ for components in the rest of the evaluations.

13.4.1 Abstraction overhead

We evaluated the influence of increasing task set utilization on the interface overhead. In this experiment, we generated components with task set utilizations from 1.5 to 8 with step size 0.1. For each utilization, we generated 1000 random task sets. Figure 13.3a shows the result of our evaluation. Note that in this figure the y-axis shows the abstraction overhead. Using the same data, we plotted the relation between the number of tasks and the interface overhead in Figure 13.3b. These results show that (i) in average O_i^{Ω} is significantly lower the O_i^{Γ}; (ii) O_i^{Γ} increases with respect to the task set utilization. Recall that we use gEDF for the MPR interfaces and we use single processor EDF for the first level EPR interfaces. The reason behind the above result is the following. Firstly, the fixed-job priority algorithms (e.g. gEDF) are not optimal for multiprocessors while EDF is optimal for single processors. Secondly, the analysis used for deriving the MPR interfaces are based on sufficient schedulability tests in global algorithms. For the class of partitioned algorithms, however, the exact schedulability tests are available. Therefore, the first parallelism level EPR interface calculation is based on the exact tests.

In order to evaluate the impact of the interface period on the interface overhead, we performed another experiment. In this experiment we fixed the task set utilization to 1.2, and we generated random tasks as explained above. We varied the component period from 10 to 200 with step size equal to 10. We generated 10000 task sets for each period. The result is illustrated in Figure 13.3c. This figure suggests that increasing the interface period has a larger impact on the EPR interfaces than on the MPR interfaces. However, even for a very large interface period, the EPR interfaces still incur smaller overhead than the MPR interfaces.

We performed another experiment to evaluate the impact of individual task

utilizations on the interface overhead. In other words, we wanted to understand whether or not heavyweight tasks and lightweight tasks have different impact on the interface overhead. We fixed the task set utilization to 2.5, and we varied the maximum task utilization from 0.3 to 0.9. We generated 1000 random components for each maximum task utilization. The results, presented in Figure 13.3d, show that (i) the MPR interfaces are more sensitive to the task utilizations; (ii) components with heavyweight tasks incur more abstraction overhead.

13.4.2 Integration

In this part we generated random systems composed of a number of components. Each component is generated randomly using the method explained above. For each system we had a target task set utilization. The task set utilization of each component $U_{\mathcal{T}_i}$ was selected randomly using a uniform distribution between 1.5 and 3. We kept generating new components until the remaining system utilization was less than 1.5 in which we generated a component with the remaining utilization. Note that by target utilization we refer to the task set utilizations as the interfaces were not derived at the system generation phase. We generated systems with target utilization from 5 to 10. We generated 10000 random systems for each target utilization. Once we generated a system, we calculated the MPR and EPR interfaces. We then ran the integration algorithms presented in the previous section. We ran the compact (CP) and the balanced (BL) algorithms for the MPR integration. Note that in the legends of the figures we use the abbreviation, i.e., CP and BL. For the EPR integration, on the other hand, we examined different combinations of decompositions and integration algorithms. Since we used the FF, BF and WF algorithms, there are nine possible combinations. We denote each combination by combining the decomposition algorithm with the integration algorithm in the legend of the figures. For instance, FFBF means that we used FF for the decomposition and BF for the integration.

In the next evaluation, we studied the performance of the two integration algorithms comparing the number of required processors by each algorithm against the minimum number of processors. We define the ratio of extra required processors by algorithm $\mathcal{A}$ as follows:

$$R^{\mathcal{A}} = 100 \times \frac{\text{\# required processors by } \mathcal{A} - \Psi^{\text{MPR}}}{\Psi^{\text{MPR}}}, \tag{13.6}$$

where Ψ^{MPR} is the minimum number of processors required for integrating a set

of MPR interfaces, and it is calculated using the following equation:

$$\Psi^{\text{MPR}} = \Big\lceil \sum_{i=1}^{n} U_{\Gamma_i^{m_i'}} \Big\rceil.$$

Figure 13.3e presents R^{BL} and R^{CP}. Each point in the figure is representing the average of 10000 samples. This figure illustrates that both algorithms perform very closely to an optimal algorithm. This shows that the flexibility provided by the MPR interfaces has been exploited well by the two algorithms. Also, the CP algorithm performs better than the BL algorithm.

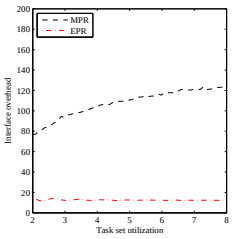
Let us define a new metric for evaluating the performance of different approaches. We define the ratio of extra required processors by approach $\mathcal{A}$ as follows:

$$R'^{\mathcal{A}} = 100 \times \frac{\text{\# required processors by } \mathcal{A} - \Psi^{\mathcal{T}}}{\Psi^{\mathcal{T}}}, \tag{13.7}$$

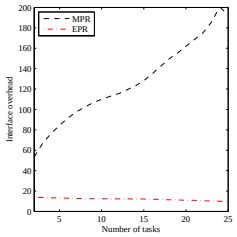
where $\Psi^{\mathcal{T}}$ represents the minimum number of processors required based on the overall task set utilizations, and it is calculated using the following equation:

$$\Psi^{\mathcal{T}} = \Big\lceil \sum_{i=1}^{n} U_{\mathcal{T}_i} \Big\rceil.$$

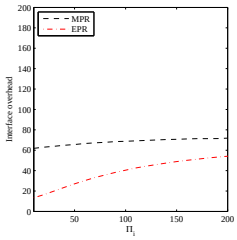
Note that we overloaded symbol $\mathcal{A}$, and it refers to a combination of the abstraction and integration techniques in the FDA approach. Since we have 11 combinations in total, and to keep the figures readable, we present the MPR integration algorithms with three EPR algorithms in one single figure. Figure 13.3f presents $R'^{\mathcal{A}}$ against the task set utilization for the case where we used FF decomposition. Similarly, Figure 13.3g and Figure 13.3h present the cases in which we used BF and WF decompositions respectively. We took the best algorithms of the above three figures and we plotted them in Figure 13.3i to make the comparison easier. This figure shows that the BFBF combination provided the best result among the studied combination of the algorithms, although it has a very close performance to BFFF. The two algorithms based on WF decomposition considered in this figure performed better than the two algorithms that are based on FF decomposition. The best MPR algorithm (i.e. CP) required 19.26 processors in average when the collective task set utilization was equal to 10. While, the best EPR algorithm combination (i.e. BFBF) required 13.87 processors in average for the same collective task set utilization. In other words, the FDA approach, in average, incurred 53.92 % less overhead than the FAD approach for this particular target task set utilization.



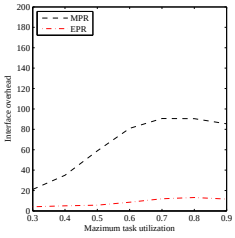
(a) Interface overhead (O_i^{Γ} and O_i^{Ω}) against task set utilization. The step size was set to 0.1.



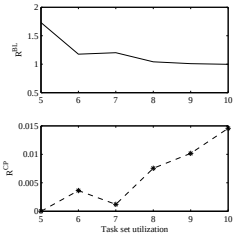
(b) Interface overhead against the number of tasks.



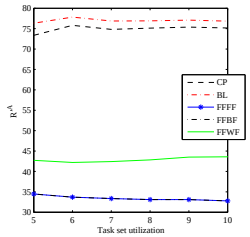
(c) Interface overhead against Π_i. The step size was set to 10.



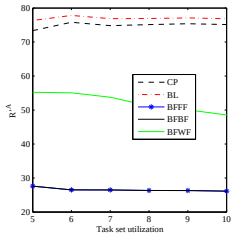
(d) Max task utilization against interface overhead. The step size was set to 0.1.



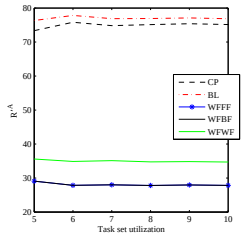
(e) R^{BL} and R^{CL} (Equation 13.6) versus task set utilization.



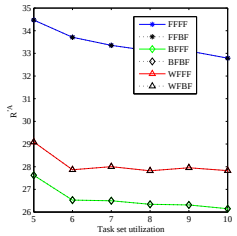
(f) $R'^{\mathcal{A}}$ versus task set utilization using the FF decomposition algorithm.



(g) $R'^{\mathcal{A}}$ versus task set utilization using the BF decomposition algorithm.



(h) $R'^{\mathcal{A}}$ versus task set utilization using the WF decomposition algorithm.



(i) $R'^{\mathcal{A}}$ versus task set utilization for the best six algorithms.

Figure 13.3: Evaluation of the interface overheads as well as integration algorithms. In all figures, the y-axis indicates the percentage of imposed overhead.

13.5 Related work

Component-based development approaches have been the subject of several studies in the real-time time scheduling community. The basic idea behind most of these approaches is to abstract the processor requirements of the components, composed of multiple real-time tasks, in an interface. The schedulability of the real-time systems, composed of multiple components, are examined using the component interfaces. These approaches are also referred to as hierarchical scheduling frameworks since the component scheduling and task scheduling are performed in two different levels. For realization of such component based systems, the processors can be time partitioned, while each partition is assigned to a single component. The processor partitions have to be compliant with the requirements specified in the component interfaces. In doing so, the components are isolated from each other with respect to their timing behavior. A timing anomaly in one component will not be propagated to the other components. Several modeling techniques have been proposed for abstracting the processor requirements of the components. In the following we review a subset of such modeling techniques related to our work.

Single processor platforms. The bounded delay abstraction, introduced in [9], specifies the bandwidth along with the maximum blackout time of the processor supply. The maximum blackout time indicates the largest time interval that the processor may be unavailable. The component schedulability test under fixed-priority scheduling and EDF, based on the bounded delay model, is presented in [10]. Shin *et al.* [6] presented another abstraction model for the processor supply of single processors, namely the Periodic Resource (PR) model. The PR model specifies a budget and a replenishment period in its interface. Easwaran *et al.* [11] proposed using a deadline in the component interface to minimize the abstraction overhead. In the case of single processor components abstracted using a periodic model, the component integration problem is equivalent to the task scheduling problem. Therefore, the schedulability analyses previously developed for examining the schedulability of the periodic tasks, can be directly applied to the components assuming that the component budget is equal to the task execution time.

Multiprocessor platforms. With the advent of multiprocessors, it became possible to develop components that require more than one processor for their computations. Therefore, researchers proposed abstraction techniques that can abstract the processor demand of such components. Bini *et al.* presented the Multi Supply Function (MSF) model in [12] for modeling the resource supply of multiprocessor platforms. The Parallel Supply Function (PSF) model [13]

is also proposed as an alternative for modeling the resource supply of hierarchical multiprocessor systems. This model indicates a set of supply functions where each of them represent the minimum available supply at a certain parallelism level (from 1 to m). Leontyev and Anderson [14] proposed a model that only specifies bandwidth w in the component interface. In this model $\lfloor w \rfloor$ of a dedicated processor is assigned to the components and the remaining $w - \lfloor w \rfloor$ bandwidth is provided using a periodic server. This model provides limited flexibility at the integration stage for the system integrator as it requires $\lfloor w \rfloor$ dedicated processors. Lipari and Bini proposed the Bounded Delay Multipartition (BDM) abstraction model in [5]. This model specifies the maximum blackout time and a bandwidth for each parallelism level in its interface. They also provided an algorithm for allocating the interfaces on multiprocessors. In our work, we addressed periodic interface models. In addition, we proposed a new approach in which component decomposition is performed before interface abstraction.

Periodic interface models for multiprocessor platforms. Zhu *et al.* [15] presented an approach in which a Deferrable Server (DS) is attached to each processor. They provided response time analysis for tasks assigned to the DSs which can migrate across the multiprocessor platform. In this work, the authors assumed that there can exist at most one DS per processor. Thus, their approach is not suitable for complex systems composed of several components. Shin *et al.* proposed the MPR model [1]. The MPR model specifies a budget, a replenishment period and a parallelism level in its interface. Easwaran *et al.* [3] proposed an optimal component scheduling algorithm for the MPR interfaces assuming that all components have identical periods. Xu *et al.* proposed the Deterministic MPR (DMPR) model in [16]. This model is different from the MPR model in the following aspect. The DMPR model, similar to [14], allows at most one partial processor allocation. Xi *et al.* [17] have investigated the application of the MPR modeling technique in the Xen virtual machine manager. The authors have also reported some benefits of using partitioned task-scheduling over global task-scheduling. On the other hand, the Generalized MPR (GMPR) model [2] specifies a budget for each parallelism level in the interfaces. This additional information in the interface make it possible to reduce the abstraction overhead.

Our work is different from the aforementioned works in the following aspects. (i) All of the aforementioned approaches perform component decomposition after the abstraction phase, while in this paper we presented an approach for performing the decomposition before the abstraction. Recall that, in order to examine the schedulability of the systems using the task schedulability

analyses, the components with utilization more than 100 % of a single processor have to be decomposed to a number of smaller components. (ii) We have quantitatively studied the overhead of using the MPR model considering the whole compositional development processes, i.e., both component abstraction and system integration.

13.6 Conclusions and future work

In this paper we investigated two alternative approaches for developing real-time software components on multiprocessor platforms. The two approaches vary in the following aspect. The first approach abstracts the component interfaces before decomposing them at the integration phase. The second one, however, first decomposes the components and then abstracts their interfaces. Through extensive simulations, we showed that the second approach utilizes the processor resource significantly better than the first approach. For instance, we showed that given a total task set utilization equal to 10, the second approach in average incurs around 53 % less overhead compared to the first approach.

In the future, we intend to propose an integration algorithm for the GMPR interface model, and we propose to evaluate the GMPR model against the two approaches presented in this paper. We only considered pEDF for scheduling the components in this paper. It is interesting to consider other algorithms including global scheduling algorithms for component-scheduling, and to compare their performances against the partitioned component-scheduling algorithms. Finally, we would like to incorporate resource sharing in our approach and compare the abstraction overhead of our approach with the current state-of-the-art (e.g. [18]).

References

[1] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems, (ECRTS'08)*, pages 181–190, July 2008.

[2] A. Burmyakov, E. Bini, and E. Tovar. Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376, 2014.

[3] A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on Scheduling Algorithms, Methods, and Models*, 2004.

[5] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*, pages 249–258, December 2010.

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

[7] B. Lecun, T. Mautor, F. Quessette, and M. Weisser. Bin packing with fragmentable items: Presentation and approximations, January 2013.

[8] N. Menakerman and R. Rom. Bin packing with item fragmentation. *Algorithms and Data Structures*, 2125:313–324, 2001.

[9] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, May 2001.

[10] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 57–67, December 2004.

[11] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium, (RTSS'07)*, pages 129–138, December 2007.

[12] E. Bini, G. Buttazzo, and M. Bertogna. The multi supply function abstraction for multiprocessors. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, (RTCSA'09)*, pages 294–302, August 2009.

[13] E. Bini, M. Bertogna, and S. Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, (RTSS'09)*, pages 437–446, December 2009.

[14] H. Leontyev and J. H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 191–200, July 2008.

[15] H. Zhu, S. Goddard, and M. B. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In *32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 239–248, December 2011.

[16] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium (RTSS'13)*, pages 1–10, December 2013.

[17] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the International Conference on Embedded Software (EMSOFT'14)*, pages 1–10, Oct 2014.

[18] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23rd EUROMICRO Conference on Real-Time Systems (ECRTS'11)*, July 2011.