

# Multi-Resource Management in Embedded Real-Time Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de  
Technische Universiteit Eindhoven, op gezag van de  
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een  
commissie aangewezen door het College voor  
Promoties in het openbaar te verdedigen  
op woensdag 17 oktober 2012 om 16.00 uur

door

Michał Jakub Holenderski

geboren te Warschau, Polen

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. J.J. Lukkien

Copromotor:

dr.ir. R.J. Bril

Printed by: Eindhoven University Press, Eindhoven, The Netherlands

© Michał Holenderski 2012

All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

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

ISBN: 978-90-386-3237-7

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem statement . . . . .	3
1.2	Contributions . . . . .	3
1.3	Outline . . . . .	4
<b>2</b>	<b>System model</b>	<b>5</b>
2.1	Resource model . . . . .	5
2.2	Application model . . . . .	7
2.3	Mapping . . . . .	11
<b>3</b>	<b>Processor management</b>	<b>17</b>
3.1	Related work . . . . .	19
3.2	System model . . . . .	24
3.3	RELTEQ . . . . .	29
3.4	Periodic tasks . . . . .	33
3.5	Servers . . . . .	35
3.6	Hierarchical scheduling . . . . .	42
3.7	Evaluation . . . . .	44
3.8	Discussion . . . . .	50
<b>4</b>	<b>Memory management</b>	<b>55</b>
4.1	Related work . . . . .	57
4.2	System model . . . . .	62
4.3	Reducing memory requirements . . . . .	68
4.4	Handling overload conditions . . . . .	77
4.5	Bounding the mode change latency . . . . .	79
4.6	Discussion . . . . .	93
<b>5</b>	<b>Multi-resource management</b>	<b>95</b>
5.1	Related work . . . . .	97
5.2	Recap of related synchronization protocols . . . . .	101
5.3	Towards multi-resource sharing . . . . .	106
5.4	System model . . . . .	107
5.5	Parallel-SRP (PSRP) . . . . .	110
5.6	Schedulability analysis for PSRP . . . . .	113
5.7	Evaluation . . . . .	128
5.8	Discussion . . . . .	130
<b>6</b>	<b>Grasp</b>	<b>133</b>
6.1	Related work . . . . .	135

6.2	Grasp overview . . . . .	136
6.3	Multiprocessor scheduling . . . . .	141
6.4	Hierarchical scheduling . . . . .	144
6.5	Hierarchical multiprocessor scheduling . . . . .	146
6.6	Timestamp synchronization . . . . .	147
<b>7</b>	<b>Conclusion</b>	<b>151</b>
	<b>Bibliography</b>	<b>153</b>
	<b>Symbol index</b>	<b>164</b>
	<b>Acronyms</b>	<b>166</b>
	<b>Accomplishments</b>	<b>167</b>
	<b>Summary</b>	<b>171</b>
	<b>Curriculum Vitae</b>	<b>174</b>

# Chapter 1

## Introduction

The history of the modern computer starts in 1944 with the Colossus. It was the world's first electronic, digital programmable computer, used by the British to decrypt German messages during World War II. It used vacuum tubes instead of mechanical or electrical relay switches found in its predecessors. The vacuum tubes were the computational units and were interconnected by simple copper wires. The machine was programmed by manually routing the wires. The first general-purpose electronic computer was the ENIAC, completed in 1946. Like the Colossus, it was colossal, weighing 30 tons, occupying 165 m<sup>2</sup>, and consuming 174kW of power (McCartney, 1999). Its 18 000 vacuum tubes could perform 5000 additions per second, and were initially used by the US army for computing the behavior of chemical reactions inside of a hydrogen bomb for the Manhattan project and later for ballistic analysis.

The invention of the transistor and the integrated circuit in the late 1940s, followed by their mass production in the 1950s, made it possible to shrink computers and to reduce their cost, making them available for many new applications. Initially they found their application in large and powerful mainframe computers, where several users at a time could execute their computations on a shared mainframe computer.

With their ever shrinking size, however, digital computers could also be used for controlling smaller systems. A prominent example is the Apollo Guidance Computer, which was responsible for guidance, navigation and control of the Apollo spacecrafts on their lunar missions in the late 1960s. It weighed 32kg, occupied 1 cubic foot, had 2K of RAM, and 36K of hard-wired core-rope memory with copper wires threaded or not threaded through tiny magnetic cores. It consumed 55 Watts and could perform 40 000 additions per second (O'Brien, 2010). The state of the art in scheduling in those days was to divide the entire control application into individual tasks, and to schedule them according to a round robin, First-In-First-Out, or table driven scheme. However, the safety critical nature of the Apollo Guidance Computer, combined with its limited resources, required new task scheduling methods, giving rise to one of the earliest priority-based schedulers. In the case of an overload, the scheduler would continue executing the highest priority tasks, at the cost of dropping those with a

lower priority (Martin, 1994).

As digital computers were increasingly used for controlling safety critical systems, theories for reasoning about the behavior of these systems started to emerge, in particular, the real-time scheduling theory. Its goal is to analyze the mapping of the digital resources (such as the processor, memory or network) to tasks to ensure that the tasks' timing constraints (derived from the system timing constraints) are met. The seminal paper by Liu and Layland (1973) is often regarded as the beginning of the real-time scheduling theory. It proposed sufficient and necessary utilization bounds for fixed priority preemptive scheduling on uniprocessor systems. Since then, real-time theory has addressed more complicated systems, such as multiprocessor and distributed platforms, with dependencies between tasks, and various other constraints. Even though significant progress has been made, real-time scheduling and analysis still offers many challenging and fundamental open problems, in particular in the multiprocessor domain (Davis and Burns, 2011).

The Apollo Guidance Computer can be regarded as the precursor of modern embedded systems. Unlike large super computers, embedded devices have stringent operational constraints, such as weight, size, and cost. Consequently, embedded systems are characterized by limited processing, storage and energy resources. For example, due to the weight constraints, an airplane will reuse the same computer hardware to run the control software during different modes of operation, such as takeoff, flight, and landing. Switching between the different modes of operation must be efficient and predictable, without disturbing the operation of the entire system.

It is often too costly to develop dedicated hardware for solving a particular problem, which has lead to a wide adoption of general purpose computers, exemplified already by the Colossus vs. ENIAC. General purpose computers offer the flexibility to reuse the same hardware for different applications. As computer platforms and applications become ever more complex, however, the programmers must rely on operating systems providing higher level abstractions for managing the resources. In embedded systems this is often achieved by explicit fine-grained multi-resource management of the various resources comprising the platform. A popular abstraction are resource reservations (Mercer et al., 1994), which aim at providing temporal isolation between independent tasks. They form the basis for virtual platforms, which provide tasks with the illusion of executing on a dedicated platform, often comprised of several different resources.

As applications became increasing complex, comprised of hundreds of tasks, it became more difficult and costly to develop large systems. This gave rise to component-based engineering, which offers a modular approach for designing and developing complex systems by grouping the tasks performing a particular function inside of components. Several component models for real-time systems were proposed since, most notably the periodic resource model (Shin and Lee, 2003), aiming to facilitate independent development and analysis of components. Hierarchical scheduling is then used to schedule components on the global level, and tasks locally within components. Due to the limited resources in embedded systems, it is important that the mechanisms provided by an operating system, such as task scheduling and timer management, are efficient and introduce little overhead.

We conducted our work within the context of multimedia processing systems in the surveillance domain. The hardware platforms in this domain are usually resource-constrained embedded systems, comprised of multiple different resources, such as a processor, memory and digital signal processors. The multimedia applications which are mapped to these platforms have real-time constraints, are data intensive, and experience high variability of resource requirements due to data-dependent workload. For example, an MPEG encoder will require more processing time for a scene with a lot of movement. Consequently, the encoded video frames will vary in size, requiring variable amount of network bandwidth during transmission and memory during decoding. If several video streams are processed on the same platform, and their total resource requirements exceed the available resources, then a tradeoff needs to be maintained between the resource requirements and the quality of the individual streams (expressed in terms of e.g. timing constraints on individual frames), to guarantee a system wide quality of service. We considered multimedia processing applications comprised of several scalable components, which can operate in different modes and are scheduled by a hierarchical scheduling framework.

## 1.1 Problem statement

This thesis addresses the problem of mapping multiple heterogenous resources to tasks in the context of resource constrained embedded real-time systems. It focuses on three research questions:

1. How can we design an efficient hierarchical scheduling framework for supporting independent development and analysis of component based systems, to provide temporal isolation between components?
2. How do we change the mapping of resources to tasks and components during run-time efficiently and predictably, and how do we analyze the latency of such a system mode change in systems comprised of several scalable components?
3. How do we schedule and analyze a set of parallel-tasks which require simultaneous access to several different resources, while guaranteeing that all tasks meet their deadlines?

## 1.2 Contributions

In Chapter 2 we propose a system model, comprised of a resource model, an application model, and a mapping between the two. The novel resource model abstracts the key properties of heterogenous resources from a scheduling perspective, providing a uniform model for different resources, such as a processor, memory or network. The application model is based on the notions of tasks and components, and supports the modeling and analysis of hierarchical and reservation-based systems.

In Chapter 3 we present RELTEQ, which is a general timer management system exhibiting low processor and memory overheads. We then leverage RELTEQ to design and implement an efficient hierarchical scheduling framework, which provides temporal isolation between components. The system overheads are evaluated based on an implementation within  $\mu\text{C}/\text{OS-II}$ , a real-time operating system used in the industry in various domains, such as aerospace, automotive, medical, and surveillance.

In Chapter 4 we investigate scalable applications operating in a memory-constrained system. A scalable application, comprised of scalable components, can operate in one of several predefined system modes, defined in terms of component modes. A component mode defines a trade-off between its resource requirements and its output quality. During runtime the system may reallocate the resources between the components, resulting in a system mode change. The latency of a system mode change, defined as the time duration between a mode change request and the time that it has been effected, should satisfy an upper bound. We first show how to reduce the memory requirements in a streaming multimedia application. We then show how to provide guaranteed resource access in spite of mode changes, while at the same time minimizing the mode change latency bound. We present a novel mode change protocol called Swift Mode Changes, which relies on fixed priority with deferred preemption scheduling. The design, analysis and implementation are presented and evaluated based on a quantitative analysis.

In Chapter 5 we address the problem of scheduling periodic parallel tasks on a multi-resource platform, where tasks have real-time constraints. The goal is to exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources. A new scheduling algorithm called PSRP is presented, together with the accompanying schedulability analysis. The benefits of PSRP are demonstrated by means of simulation results and an example application showing that PSRP indeed exploits the available concurrency in heterogeneous real-time systems.

In Chapter 6 we present a trace visualization toolset called Grasp, which we have used extensively during the design and development of the various real-time operating system extensions described in this thesis. It provides a clear and intuitive user interface and a simple architecture, making it easy to extend Grasp with new visualizations.

### 1.3 Outline

This thesis is structured as follows. Chapter 2 introduces the system model, which is used later in Chapters 3, 4, and 5. Each of these chapters instantiates and extends the model for its particular needs. Chapters 3, 4, 5, and 6 discuss the main contributions outlined in Section 1.2. Chapter 7 concludes this thesis. A list of publications contributing to this thesis can be found in the Accomplishments appendix.



## Chapter 2

# System model

A system consists of applications, resources and a mapping between them. Application workload is expressed in terms of tasks, where a task represents the work which needs to be done in response to an event, such as processing a newly arrived video frame. The mapping of resources to tasks describes how the “ownership” of resources changes during runtime, i.e. which task “owns” a particular resource at a particular time. The ownership may change, e.g. if several tasks need to access the same memory region or the same processor. A mapping must satisfy certain soundness constraints, e.g. every mutually exclusive resource is owned by at most one task at a time. In real-time systems there are additional timeliness constraints, which are often expressed in terms of task deadlines. Embedded systems exhibit additional constraints which address the overheads associated with the mapping due to limited resources, such as scheduling, and context switching overheads.

In this chapter we present an abstraction which allows us to model the scheduling of tasks on *different* resources in a uniform way. The essential abstraction is that any resource (such as a processor, memory space or bus) can be represented as a multi-unit preemptive or non-preemptive resource.

### 2.1 Resource model

The purpose of a resource model is to provide a certain level of abstraction helping to describe the mapping of physical resources to tasks. It has to be simple enough to reason about, while at the same time expressive enough to use the available resources efficiently. The main contribution of our model is the ability to schedule tasks on various different resources, such as processor and memory, in a uniform way. At the core of our resource model is the multi-unit resource.

**Definition 2.1** (Multi-unit resource). *Let  $\mathcal{R}$  be the set of all resources in the system. A multi-unit resource  $r \in \mathcal{R}$  consists of multiple units, where each unit is a serially accessible entity. A resource  $r$  is specified by its capacity  $N_r \geq 1$ , which represents the maximum number of units the resource can provide simultaneously.*

Memory space is an example of a *multi-unit resource*. In this thesis, when talking about the memory space resource we are interested in the memory requirements in terms of memory size, and ignore the specifics of memory allocation and the actual data stored in the memory. A memory, managed as a collection of fixed sized blocks, can be regarded as a multi-unit resource with capacity equal to the number of blocks. In this sense our multi-unit resource is similar to a multi-unit resource discussed by Baker (1991).

The capacity of a multi-unit resource represents essentially the maximum number of tasks which can use the resource simultaneously. A multi-core processor can therefore be modeled as a resource with capacity equal to the number of cores.

Resource management is about managing access to scarce resources, i.e. resources for which at times the demand may exceed their provision. If the total requirement for a resource never exceeds its capacity, then the management is trivial: we can always provide access to the resources. When modeling systems we can therefore ignore resources for which the demand never exceeds their provision.

Single-unit resources are a special case of multi-unit resources.

**Definition 2.2** (Single-unit resource). *A single-unit resource  $r \in \mathcal{R}$  is a multi-unit resource such that  $N_r = 1$*

A single-core processor is an example of a *single-unit resource*, since only a single task can be using it at a time. A memory controller synchronizing the access to a memory space can also be regarded as a single-unit resource<sup>1</sup>. In the remainder of the thesis we assume multi-unit resources with capacity greater than 1, unless explicitly stated otherwise.

### 2.1.1 Preemptive vs. non-preemptive resources

A preemption is the change of ownership of a resource unit before the owner is ready to relinquish the ownership. In terms of the traditional task model, a job (representing the ownership of a resource) may be preempted by another job before it completes. We can classify all resources in one of two categories:

**Definition 2.3** (Preemptive resource). *The usage (or ownership) of a unit of a preemptive resource can be preempted without corrupting the state of the resource. We use  $\mathcal{P} \subseteq \mathcal{R}$  to denote the set of all preemptive resources in the system.*

**Definition 2.4** (Non-preemptive resource). *The usage (or ownership) of a unit of a non-preemptive resource may not be preempted without the risk of corrupting the state of the resource. We use  $\mathcal{N} \subseteq \mathcal{R}$  to denote the set of all non-preemptive resources in the system.*

Every resource is either preemptive or non-preemptive, i.e.

$$(\mathcal{N} \cup \mathcal{P} = \mathcal{R}) \wedge (\mathcal{N} \cap \mathcal{P} = \emptyset). \quad (2.1)$$

---

<sup>1</sup>A memory can therefore be modeled by *two* resources: one representing the memory space and the other representing mutually exclusive access to the memory.

A processor is an example of a preemptive resource, as the processor state of a running task can be saved upon a preemption and later restored. A bus is an example of a non-preemptive resource, as an ongoing message transfer cannot be preempted without losing the message. A logical resource (e.g. a shared variable) is another example of a non-preemptive resource.

Preemption is usually provided on the software level, by the operating system. For example, when an interrupt arrives, the operating system first stores the processor state, then executes the interrupt handler, and finally restores the saved processor state. Note that actually the usage of nearly any resource can be preempted: e.g. memory space (usually considered a non-preemptive resource) can be “switched out” to a memory higher in the memory hierarchy (e.g. data can be moved from the processor cache to the RAM), or the data can be simply overwritten and recomputed later. However, this will come at the cost of a performance penalty (incurred by storing-and-restoring or recomputing the state of the resource). A non-preemptive resource is basically one for which the system designer has decided that its preemption overhead is too large.

We assume mutually exclusive access to resources, meaning that each unit of a multi-unit resource can be accessed by at most one task at a time. For example, each core in a multicore processor can be accessed by at most one task at a time. Notice that preemptiveness is orthogonal to mutual exclusion, as mutual exclusion holds for both preemptive and non-preemptive resources.

### 2.1.2 Physical vs. virtual resources

Physical resources are the hardware resources provided by the platform, such as a processor, memory or bus. As we will see later in Section 2.2.2, application components can provide their own *virtual* resources. We use  $\mathcal{R}$  to denote both physical and virtual resources. Similar to physical resources, each virtual resource is either preemptive or non-preemptive.

## 2.2 Application model

Let  $\mathcal{T} = \mathbb{R}_0^+$  be the time domain (of non-negative real numbers), with  $t \in \mathcal{T}$  representing a time instant or the duration of a time interval.

### 2.2.1 Tasks

We model applications in terms of a set of *tasks*. Each task specifies the resource requirements which are required to do a particular work.

**Definition 2.5** (Task). *A task  $\tau_i$  is specified by its*

- *fixed and unique priority  $\pi_i$  (lower number indicating higher priority),*
- *period  $T_i$ , which specifies the inter-arrival time between two consecutive instances of  $\tau_i$ ,*

- *initial offset (or phasing)  $O_i$ , which specifies the arrival time of the first instance of  $\tau_i$ ,*
- *relative deadline  $D_i$ , with  $D_i \leq T_i$ ,*
- *sequence of segments  $S_i$ , where the  $j$ -th segment  $\tau_{i,j} \in S_i$  is specified by its worst-case execution time  $E_{i,j}$ , and a set of resource requirements  $R_{i,j}$ . Each resource requirement  $(r, n) \in R_{i,j}$  represents a requirement for  $n > 0$  units of resource  $r \in \mathcal{R}$ .*

We use  $\Gamma$  to denote the set of all tasks in the system, and  $\mathcal{S}$  to denote the set of all segments in the system, i.e.

$$\mathcal{S} = \bigcup_{\tau_i \in \Gamma} S_i$$

Our task therefore models programs which can be expressed as a sequence of segments, where each segment  $\tau_{i,j}$  is wrapped between a *lock*( $R_{i,j}$ ) and *unlock*( $R_{i,j}$ ) operation. The semantics of these operations is similar to the primitives used in (Havender, 1968) for locking resources collectively.

Priorities are used for resolving conflicts during runtime when more than one task tries to access a shared resource. All segments  $\tau_{i,j} \in S_i$  share its priority  $\pi_i$ . We use  $\pi_\perp$  to denote a priority lower than the lowest priority, and  $\pi_\top$  to denote a priority higher than the highest priority among all tasks.

**Notation** To keep the notation short, if a segment  $\tau_{i,j}$  requires only single-unit resources, or if the number of required units is not important, we will write  $R_{i,j} = \{r_1, r_2, r_3\}$  instead of  $R_{i,j} = \{(r_1, 1), (r_2, 1), (r_3, 1)\}$ .

We use a shorthand notation to refer to the resources which are required by a segment, where  $r \in R_{i,j}$  means that  $\exists(x, n) \in R_{i,j} : x = r$ .

When specifying segments we use a shorthand notation, where  $(e, \{r_1, r_2, \dots\})$  represents a segment  $\tau_{i,j}$  with  $E_{i,j} = e$  and  $R_{i,j} = \{r_1, r_2, \dots\}$ .

When applying set operations to sets of resource requirements we are usually interested only in the identity of the resources (and not the number of units). Therefore, for simplicity, we will use  $R_{i,j} \cap R_{x,y}$  to denote  $\{r \mid (r, n) \in R_{i,j} \wedge (r, m) \in R_{x,y}\}$ , and  $R_{i,j} \cup R_{x,y}$  to denote  $\{r \mid (r, n) \in R_{i,j} \vee (r, m) \in R_{x,y}\}$ .

We use  $E_i = \sum_{\tau_{i,j} \in S_i} E_{i,j}$  to denote the worst-case execution time of task  $\tau_i$ .

**Example 2.1.** Our model distinguishes between 2 single-core processors (modeled as 2 single-unit resources) and a single 2-core processor (modeled as one multi-unit resource). In the 2 single-core processors case, we can specify a task requiring simultaneous access for  $t$  time units to particular single-core processors, e.g.

$$\mathcal{P} = \{p_1, p_2\}, N_{p_1} = 1, N_{p_2} = 1, \Gamma = \{\tau_1\}, S_1 = \langle (t, \{p_1, p_2\}) \rangle.$$

In the 2-core processor case we can only specify a task requiring simultaneous access for  $t$  time units to a particular number of cores, e.g.

$$\mathcal{P} = \{p\}, N_p = 2, \Gamma = \{\tau_1\}, S_1 = \langle (t, \{(p, 2)\}) \rangle.$$

□

### 2.2.2 Components

Next to the notion of physical resources (described in Section 2.1.2) we introduce the notion of *virtual* resources, which are provided by *components*.

**Definition 2.6** (Component). *A component  $c$  is specified by its*

- *fixed and unique priority  $\pi_c$  (lower number indicating higher priority),*
- *period  $T_c$ , which specifies the inter-arrival time between two consecutive instances (or replenishments of the budget) of  $c$ ,*
- *initial offset (or phasing)  $O_c$ , which specifies the arrival time of the first instance (or replenishment of the budget) of  $c$ ,*
- *relative deadline  $D_c$ , with  $D_c \leq T_c$ ,*
- *set of resource requirements  $R_c$ , where each resource requirement  $(r, n) \in R_c$  represents a requirement for  $n > 0$  units of resource  $r \in \mathcal{R}$ ,*
- *set of resource provisions  $P_c$ , where each resource provision  $(r, n) \in P_c$  represents a provision of  $n > 0$  units of resource  $r \in \mathcal{R}$ ,*
- *time capacity (or budget)  $E_c$ , which specifies the amount of time during each period that component  $c$  will provide the resources in  $P_c$ , while requiring the resources in  $R_c$ .*

We use  $\mathcal{C}$  to denote the set of all components in the system.

A component  $c$  provides *virtual* resources, which are specified in  $P_c$ . We include these resources in the set of all resources in the system, i.e.

$$\forall c \in \mathcal{C} : P_c \subseteq \mathcal{R}.$$

The following two examples demonstrate how to express a processor server and a memory buffer in terms of our component model.

**Example 2.2** (A processor server component). A single core processor can be modeled as a preemptive resource  $cpu \in \mathcal{P}$  with  $N_{cpu} = 1$ . A processor server  $s$  provides a share of a processor bandwidth, which is specified by its replenishment period  $\Pi_s$  and capacity  $\Theta_s$  (Shin and Lee, 2003). When a server is running, every time unit its remaining budget is decremented by one. Every  $\Pi_s$  time units its remaining budget is replenished to  $\Theta_s$ . In systems with global fixed-priority scheduling each server also has a fixed priority  $\pi_s$ .

A server  $s$  providing a share of processor  $cpu$ , can be modeled as a component  $c \in \mathcal{C}$  with

$$\begin{aligned}
 \pi_c &= \pi_s \\
 T_c &= \Pi_s \\
 O_c &= 0 \\
 D_c &= \Pi_s \\
 E_c &= \Theta_s \\
 R_c &= \{(cpu, 1)\} \\
 P_c &= \{(cpu_s, 1)\}
 \end{aligned} \tag{2.2}$$

with  $cpu_s \in \mathcal{P}$ . □

**Example 2.3** (A memory buffer component). A memory containing  $M$  bytes can be modeled by a non-preemptive resource  $mem \in \mathcal{N}$  with  $N_{mem} = M$ . A memory buffer component manages a part of the memory space in terms of buffer elements and provides a FIFO access to these elements. Each buffer  $q$  has a finite capacity  $NumElems_q$ , defining the maximum number of elements which can be stored in the buffer, where each element has a fixed size of  $ElemSize_q$  bytes.

A buffer  $q$ , which resides in memory  $mem$  and is created during the initialization of the system and never destroyed, can be modeled as a component  $c \in \mathcal{C}$  with

$$\begin{aligned}
 T_c &= \infty \\
 O_c &= 0 \\
 D_c &= \infty \\
 E_c &= \infty \\
 R_c &= \{(mem, ElemSize_q * NumElems_q)\} \\
 P_c &= \{(elements_q, NumElems_q), (mutex_q, 1)\}
 \end{aligned} \tag{2.3}$$

with  $elements_q, mutex_q \in \mathcal{N}$ .  $T_c = \infty$  means that the component is aperiodic.  $O_c = 0$  and  $D_c = E_c = \infty$  mean that the buffer will be created at the system initialization and live until the system terminates. During that time it will require  $ElemSize_q * NumElems_q$  units of the  $mem$  resource, and in return it will provide a virtual  $elements_q$  resource containing  $NumElems_q$  buffer elements.

A buffer provides interface methods to store and retrieve elements from the buffer. These methods manipulate the buffer's internal data structures and have to execute in a mutually exclusive fashion. A common implementation will guard these methods with a mutex. A call to the buffer's interface methods is modeled by a requirement for one unit of the virtual  $mutex_q$  resource for the duration of the call. Since  $N_{mutex_q} = 1$  and  $mutex_q \in \mathcal{N}$ , only one task may access the buffer's interface methods at a time. Note that a buffer requires mutually exclusive access only to its internal data structures. This means that one task can be reading from the buffer's head element, while another task is writing to the tail element, provided that the internal data structures keeping track of the pointers to the head and tail are updated in a mutually exclusive manner.

Note that since buffers live in the memory for the entire duration of the system execution, during runtime there is no need for arbitration between two buffers com-

peting for memory: all buffers need to fit in the memory or the application cannot start executing. Hence, the priority of the buffer component is irrelevant.  $\square$

### 2.2.3 Applications

We can group tasks and components which together perform a particular function to form an application.

**Definition 2.7.** *An application  $a$  is specified by its*

- *task set  $\Gamma_a \subseteq \Gamma$ ,*
- *component set  $\mathcal{C}_a \subseteq \mathcal{C}$ .*

*We use  $\mathcal{A}$  to denote the set of all applications in our system.*

Every task (or component) belongs to at most one application. We assume that applications are independent and do not share tasks (or components), i.e.

$$\forall a, b \in \mathcal{A} : a \neq b \Rightarrow (\Gamma_a \cap \Gamma_b = \emptyset \wedge \mathcal{C}_a \cap \mathcal{C}_b = \emptyset). \quad (2.4)$$

## 2.3 Mapping

In Sections 2.1 and 2.2 we have defined the static models describing the resources and the applications. Now we move on to the mapping of resources to applications. The mapping is responsible for time sharing the access to resources during runtime, and is defined by *allocation* and *scheduling*. An allocation assigns task segments and components to their required resources, while a schedule describes *when* segments and components gain access to their assigned resources.

### 2.3.1 Allocation

In this thesis we assume static allocation of resources (also referred to as *partitioning*), which is specified in our system model by the resource requirements of task segments and components. In embedded systems, where hardware platforms are usually comprised of several different resources (e.g. CPU, DSP, DMA controller), tasks must be explicitly mapped to the various resources. In such systems static allocation of tasks to resources is often desired. It also simplifies the scheduling of the entire platform during runtime, since the decision of allocation has been taken offline.

#### Resource identity in multi-unit resources

When a task or component requests memory space for storing its data, it assumes that no other task or component will modify and corrupt the data inside of its memory space. This suggests that two parameters should be associated with each granted memory request: its size and its identity. The scheduling function makes sure that a resource request for  $n$  units of a multi-unit resource is granted only if the resource

has at least  $n$  units available. When a request is granted, the requesting task or component has complete control over these units. Protection mechanisms preventing tasks and components from corrupting each others resources are outside the scope of this model.

However, when a task or component accessing a resource is preempted it may be critical that it is later resumed on the *same* resource units. For example, when a job is preempted while it is writing to a particular memory location, once it is resumed it should continue writing to the same location. Our model abstracts from the identity of individual units in a multi-unit resource and therefore it cannot be used to reason about particular resource units. However, we can assume the context switch to be responsible for making sure that a segment is resumed on the appropriate resource units. In the example of a preemption of a task writing to memory, upon preemption the system may store the partially written data to the disk and later restore it to the memory upon resuming.

### 2.3.2 Scheduling

The schedule needs to maintain the timing constraints of tasks and components and mutually exclusive access to shared resources. All these constraints can be resolved *offline*, giving rise to a *time-driven schedule*. Alternatively, the scheduling can be done *online*, where scheduling decisions are *event-driven* and performed according to a predefined scheduling policy. In this thesis we consider the latter, focusing on *Fixed-Priority Preemptive Scheduling* (FPPS) and *Fixed-Priority with Deferred-preemption Scheduling* (FPDS).

We can schedule both tasks and components. When a task segment (or a component) is scheduled on a resource during runtime, we say that the segment (or component) is *using* the resource, or that it *owns* the resource. The ownership of a resource may change. However, at any point in time each unit of a multi-unit resource may be used by at most one segment (or component).

The scheduling of segments on a multi-unit resource  $r$  can be visualized by means of  $N_r$  “tracks”, where each track represents the usage of a single unit of resource  $r$ , as illustrated in Example 2.4.

**Example 2.4.** Figure 2.1 represents a schedule of segments  $b$ ,  $c$ , and  $d$  on resources  $cpu$  and  $dsp$ , with  $b$  scheduled at time 0,  $d$  scheduled at time 1 and  $c$  scheduled at time 5 and preempting segment  $d$ . The arrival pattern of a segment is determined by the task it belongs to, which is discussed in the next section.

A resource manages its units internally, meaning that we cannot specify a requirement for a particular unit of a multi-unit resource. A segment can only specify a requirement for a certain number of arbitrary units within a multi-unit resource. However, we assume that while a segment is “executing” on a resource unit the segment will not be migrated to another unit. For example, in Figure 2.1, once  $d$ ’s requirement for 2 cores of the  $cpu$  is granted,  $d$  will own the same  $cpu$  cores throughout its execution in the time interval  $[1, 5)$ . Note that the  $cpu$  resource may decide to migrate  $d$  to different units when it is preempted at time 5.  $\square$



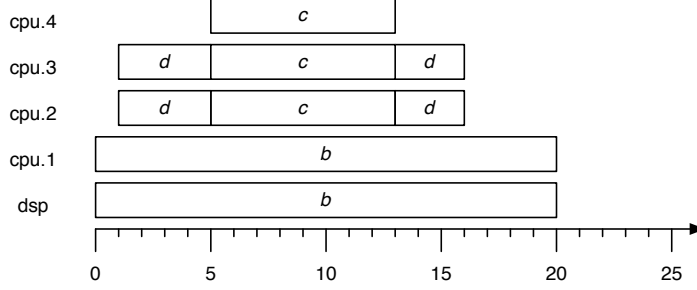


Figure 2.1: Example of a mapping of segments  $\mathcal{S} = \{b, c, d\}$  with  $E_b = 20$ ,  $R_b = \{(cpu, 1), (dsp, 1)\}$ ,  $E_c = 8$ ,  $R_c = \{(cpu, 3)\}$ ,  $E_d = 7$ ,  $R_d = \{(cpu, 2)\}$ , on a platform containing a 4-core *cpu* and a single *dsp*, i.e.  $\mathcal{R} = \{cpu, dsp\}$  with  $N_{cpu} = 4$ ,  $N_{dsp} = 1$ . We use a dot notation to refer to the individual units in a multi-unit resource.

### Scheduling of tasks

**Definition 2.8** (Task schedule). A schedule  $\sigma^{\mathcal{S}} : \mathcal{T} \times \mathcal{S} \times \mathcal{R} \rightarrow \mathbb{N}$  expresses resource ownership by task segments during runtime, where  $\sigma^{\mathcal{S}}(t, s, r)$  is the number of units of resource  $r$  which are owned by segment  $s$  at time  $t$ .

Saying that “at time  $t$  segment  $s$  owns  $n$  units of resource  $r$ ” is synonymous to saying that “at time  $t$  segment  $s$  executes on  $n$  units of resource  $r$ ” or that “at time  $t$  segment  $s$  is scheduled on  $n$  units of resource  $r$ ”.

**Definition 2.9.** We define  $\eta^{\mathcal{S}} : \mathcal{T} \times \mathcal{S} \times \mathcal{R} \rightarrow \{0, 1\}$ , where  $\eta^{\mathcal{S}}(t, s, r)$  returns 1 if at time  $t$  segment  $s$  is scheduled on resource  $r$ , and 0 otherwise, i.e.

$$\eta^{\mathcal{S}}(t, s, r) = \begin{cases} 1 & \text{if } \sigma^{\mathcal{S}}(t, s, r) > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (2.5)$$

The scheduling function  $\sigma^{\mathcal{S}}$  must satisfy the following criteria:

1. A segment is scheduled only on resources which it requires, i.e.

$$\forall t \in T, s \in \mathcal{S}, r \in \mathcal{R} : \sigma^{\mathcal{S}}(t, s, r) > 0 \Rightarrow r \in R_s. \quad (2.6)$$

2. A segment is scheduled on all of its required resources, or not at all, i.e.

$$\forall t \in T, s \in \mathcal{S} : (\exists r \in R_s : \sigma^{\mathcal{S}}(t, s, r) > 0) \Rightarrow (\forall (r, n) \in R_s : \sigma^{\mathcal{S}}(t, s, r) = n). \quad (2.7)$$

3. The scheduling function must satisfy the realtime constraints, by making sure that every segment receives its worst-case execution time on its required resources before its parent task's deadline, i.e.

$$\forall k \in \mathbb{N}, \tau_{i,j} \in \mathcal{S}, r \in R_{i,j} : (k+1)E_{i,j} \leq \int_0^{k \cdot T_i + D_i} \eta^{\mathcal{S}}(x, \tau_{i,j}, r) dx. \quad (2.8)$$

### Scheduling of components

**Definition 2.10** (Component schedule). *A schedule  $\sigma^C : \mathcal{T} \times \mathcal{C} \times \mathcal{R} \rightarrow \mathbb{N}$  expresses resource ownership by components during runtime, where  $\sigma^C(t, c, r)$  is the number of units of resource  $r$  which are owned by component  $c$  at time  $t$ .*

Saying that “at time  $t$  component  $c$  owns  $n$  units of resource  $r$ ” is synonymous to saying that “at time  $t$  component  $c$  executes on  $n$  units of resource  $r$ ” or that “at time  $t$  component  $c$  is scheduled on  $n$  units of resource  $r$ ”.

**Definition 2.11.** *We define  $\eta^C : \mathcal{T} \times \mathcal{C} \times \mathcal{R} \rightarrow \{0, 1\}$ , where  $\eta^C(t, c, r)$  returns 1 if at time  $t$  component  $c$  is scheduled on resource  $r$ , and 0 otherwise, i.e.*

$$\eta^C(t, c, r) = \begin{cases} 1 & \text{if } \sigma^C(t, c, r) > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (2.9)$$

**Definition 2.12.** *We define  $\beta : \mathcal{T} \times \mathcal{C} \rightarrow \mathcal{T}$ , which captures the remaining budget (or time) of components during runtime, where  $\beta(t, c)$  is the remaining budget of component  $c$  at time  $t$ .*

The scheduling function  $\sigma^C$  must satisfy the following criteria:

4. A component is scheduled only on resources which it requires, i.e.

$$\forall t \in T, c \in \mathcal{C}, r \in \mathcal{R} : \sigma^C(t, c, r) > 0 \Rightarrow r \in R_c. \quad (2.10)$$

5. A component is scheduled on all of its required resources, or not at all, i.e.

$$\forall t \in T, c \in \mathcal{C} : (\exists r \in R_c : \sigma^C(t, c, r) > 0) \Rightarrow (\forall (r, n) \in R_c : \sigma^C(t, c, r) = n). \quad (2.11)$$

6. The scheduling function must satisfy the realtime constraints, by making sure that every component receives its required time capacity on its required resources before its deadline. The exact specification may differ for different components. For the periodic-idling server component (see Section 3.2.2), the real-time requirement can be formalized as

$$\forall k \in \mathbb{N}, c \in \mathcal{C}, r \in R_c : (k+1)E_c = \int_0^{k \cdot T_c + D_c} \eta^C(x, c, r) dx. \quad (2.12)$$

7. Only components with remaining budget are scheduled, i.e.

$$\forall t \in \mathcal{T}, c \in \mathcal{C} : (\exists r \in \mathcal{R} : \sigma^C(t, c, r) > 0) \Rightarrow \beta(t, c) > 0. \quad (2.13)$$

8. The remaining budget should never exceed component’s time capacity, i.e.

$$\forall t \in \mathcal{T}, c \in \mathcal{C} : 0 \leq \beta(t, c) \leq E_c. \quad (2.14)$$

9. The remaining budget of a component  $c$  is replenished periodically, with period  $T_c$ . Each budget is consumed at a uniform rate whenever it is scheduled. The exact specification may differ for different components. For the periodic-idling server component (see Section 3.1.2), the periodic replenishment and uniform consumption requirement can be formalized as

$$\forall k \in \mathbb{N}, c \in \mathcal{C}, t \in [kT_c, (k+1)T_c), r \in R_c : \beta(t, c) = E_c - \int_{kT_c}^t \eta^c(x, c, r) dx. \quad (2.15)$$

### Scheduling of tasks and components

The scheduling functions  $\sigma^{\mathcal{S}}$  and  $\sigma^{\mathcal{C}}$  must satisfy the following criteria:

10. The scheduling functions never schedule more units than a resource can provide, i.e.

$$\forall t \in \mathcal{T}, r \in \mathcal{R} : \sum_{s \in \mathcal{S}} \sigma^{\mathcal{S}}(t, s, r) + \sum_{c \in \mathcal{C}} \sigma^{\mathcal{C}}(t, c, r) \leq N_r. \quad (2.16)$$

Note that several segments and component may be scheduled on a resource at the same time, as long as together they do not require more units than the resource can provide.

11. A task or component is scheduled on a virtual resource provided by another component only when that component is itself scheduled on its required resources, i.e.

$$\forall t \in \mathcal{T}, s \in \mathcal{S}, c \in \mathcal{C}, p \in P_c : \sigma^{\mathcal{S}}(t, s, p) > 0 \Rightarrow (\forall q \in R_c : \sigma^{\mathcal{C}}(t, c, q) > 0) \quad (2.17)$$

$$\forall t \in \mathcal{T}, c_1, c_2 \in \mathcal{C}, p \in P_{c_2} : \sigma^{\mathcal{C}}(t, c_1, p) > 0 \Rightarrow (\forall q \in R_{c_2} : \sigma^{\mathcal{C}}(t, c_2, q) > 0) \quad (2.18)$$



## Chapter 3

# Processor management

Modern real-time systems have become exceedingly complex. A typical car is controlled by over 100 million lines of code executing on close to 100 Electronic Control Units (ECU). With more and more functions being implemented in software, the traditional approach of implementing each function (such as engine control, ABS, windows control) on a dedicated ECU is no longer viable, due to increased manufacturing costs, weight, power consumption, and decreased reliability and serviceability (Nolte et al., 2009). With the ECUs having increasingly more processing power, it has become feasible to integrate several functions on a single ECU. However, this introduces the challenge of supporting independent and concurrent development and analysis of individual functions which are later to be integrated on a shared platform. A popular approach in the industry and literature is component-based engineering, where the complete system is divided into smaller software components which can be developed independently. The Automotive Open System Architecture (AUTOSAR) (AUTOSAR, 2011) standard is an example of such an approach in the automotive domain. It relies on a formal specification of component interfaces to verify the functional properties of their composition. Many functions in automotive systems, however, also have real-time constraints, meaning that their correct behavior is not only dependent on their functional correctness but also their temporal correctness. AUTOSAR does not provide temporal isolation between components. Verifying the temporal properties of an integrated system requires complete knowledge of all functions comprising the components mapped to the same ECU, and therefore violates the requirement for independent development and analysis.

In this chapter we address the problem of providing temporal isolation to components in an integrated system. Ideally, temporal isolation allows to develop and verify the components independently (and concurrently), and then to seamlessly integrate them into a system which is functioning correctly from both a functional and timing perspective (Shin and Lee, 2008; Nolte, 2011). The question is how to provide true temporal isolation when components execute on a shared processor. We address this problem by means of an hierarchical scheduling framework (HSF).

An HSF provides the means for the integration of independently developed and

analyzed components into a predictable real-time system. A component is defined by a set of tasks, a local scheduler and a *server*, which defines the component's time budget (i.e. its share of the processing time) and its replenishment policy.

## Problem description

An HSF-enabled platform should provide the following general functionalities:

1. Interface for the creation of servers and assigning tasks to servers.
2. *Virtual timers*, which are relative to a components's budget consumption, as well as *global timers*, which are relative to a fixed point in time.
3. Local scheduling of tasks within a component, and global scheduling of components on the system level.

In this chapter we focus on providing temporal isolation and preventing interference between components. We aim at satisfying the following additional requirement:

4. Expiration of events local to a component, such as the arrival of periodic tasks, should not interfere with other components. In particular, the handling of the events local to inactive components should be deferred until the corresponding component is activated. The time required to handle them should be accounted to the corresponding component, rather than the currently active one.

These requirements should be met by a modular and extensible design, with low performance overhead and minimal modifications to the underlying RTOS. It should exhibit predictable overhead, while remaining efficient to support resource-constrained embedded systems in the automotive domain.

Real-time applications will often require support for periodic task arrival. Periodic tasks rely on timers to represent their arrival time. For servers, we also need timers representing the replenishment and depletion of a budget. Vital, and a starting point for our design, is therefore the support for simple timers (or timed events), i.e. the assumption that an event can be set to arrive at a certain time. This simple timer support is typically available in an off-the-shelf Real-Time Operating System (RTOS) (Labrosse, 2002). Some RTOSes provide much more functionality (for which our work then provides an efficient realization) but other systems provide just that. As a result, the emphasis lies with the management of timers. The timer management should support long event inter-arrival times and long lifetime of the system at a low overhead.

## Contributions

We first present the design of a general timer management system, which is based on Relative Timed Event Queues (RELTEQ), an efficient timer management system targeted at embedded systems. Pending timers are stored in a queue sorted on the expiration time, where the expiration time of each timer is stored relative to the

previous timer in the queue. This representation makes it possible to reduce the memory requirements for storing the expiration times, making it ideal for resource constrained embedded systems. We have implemented RELTEQ within  $\mu C/OS-II$ , and showed that it also reduces the processor overhead compared to the existing timer implementation.

We then leverage RELTEQ to implement periodic tasks and design an efficient HSF. The proposed HSF extension of RELTEQ supports various servers (including the polling, idling-periodic, deferrable and constant-bandwidth servers), and provides access to both virtual and global timers. It supports independent development of components by separating the global and local scheduling, and allowing each server to define a dedicated scheduler. The HSF design provides a mechanism for tasks to monitor their server's remaining budget, and addresses the system overheads inherent to an HSF implementation. It provides temporal isolation and limits the interference of inactive servers on the system level. Moreover, it avoids recalculating the expiration of virtual events upon every server switch and thus reduces the worst-case scheduler overhead.

The proposed design is evaluated based on an implementation within  $\mu C/OS-II$ , a commercial operating system used in the automotive domain. The results demonstrate low overheads of the design and minimal interference between the components.

In this chapter we focus on the means for implementing a HSF. The corresponding analysis falls outside of the scope.

## Publications

We have introduced RELTEQ in (Holenderski et al., 2009c). In (Holenderski et al., 2010a, 2012a) we have presented an HSF extension of RELTEQ supporting fixed-priority servers. We have extended the design with the constant bandwidth server in (van den Heuvel et al., 2011), and applied it to priority processing in multimedia systems.

## 3.1 Related work

In this section we summarize the existing work related to the processor management discussed in this chapter.

### 3.1.1 Processor reservations

Resource reservations have been introduced by (Mercer et al., 1994), aiming at providing temporal isolation for individual components comprising a real-time system, to guarantee resource provisions in a system with dynamically changing resource requirements. They focused on the processor and specified the reservation budget by a tuple  $(C, T)$ , with capacity  $C$  and period  $T$ . The semantics is as follows: a reservation will be allocated  $C$  units of processor time every  $T$  units of time. When a reservation

uses up all of its  $C$  processor time within a period  $T$  it is said to be *depleted*. Otherwise it is said to be *undepleted*. At the start of the period  $T$  the reservation capacity is *replenished*.

(Rajkumar et al., 1998) identify four ingredients for guaranteeing resource provisions:

1. *Admission*: When a reservation is requested, the system has to check if granting the reservation will not affect any timing constraints.
2. *Scheduling*: The reservations have to be scheduled on the global level, and tasks have to be scheduled within the reservations.
3. *Accounting or monitoring*: Processor usage of tasks has to be monitored and accounted to their assigned reservations.
4. *Enforcement*: A reservation, once granted, has to be enforced by preventing other components from “stealing” the granted budget.

(Rajkumar et al., 1998) aim at a uniform resource reservation model and extended the concept of processor reserves to other resources, in particular the disk bandwidth. They schedule processor reservations according to FPPS and EDF, and disk bandwidth reservations according to EDF. They extend the reservation model to  $(C, T, D, S, L)$ , with capacity  $C$ , period  $T$ , deadline  $D$ , starting time  $S$  and the life-time  $L$  of resource reservation, meaning that the reservation guarantees of  $C$ ,  $T$  and  $D$  start at  $S$  and terminate at  $S + L$ . Note that (Rajkumar et al., 1998) apply their uniform resource reservation model only to *single-unit resources*, such as processor or disk-bandwidth. They do not show how their methods can be applied to *multi-unit resources*, such as memory.

### 3.1.2 Hierarchical scheduling frameworks

A hierarchical scheduling framework (HSF) facilitates independent analysis and development of components, which can be integrated into a complete system based only on the information provided on their interfaces. A compositional system model is therefore critical.

Shin and Lee (2003) introduce the periodic resource model, allowing the integration of independently analyzed components in compositional hard real-time systems. Their resource is specified by a pair  $(\Pi_i, \Theta_i)$ , where  $\Pi_i$  is its replenishment period and  $\Theta_i$  is its capacity. They also describe the schedulability analysis for a HSF based on the periodic resource model under the Earliest Deadline First and Rate Monotonic scheduling algorithms on local and global level. While the periodic-idling (Davis and Burns, 2005) and deferrable (Strosnider et al., 1995) servers conform to the periodic resource model, the polling (Lehoczky et al., 1987) server does not. The HSF presented in this chapter supports various two-level hierarchical processor scheduling mechanisms, including the polling, periodic idling, deferrable servers, and constant-bandwidth (Abeni and Buttazzo, 1998) servers. We have reported on the benefits of our constant-bandwidth server implementation in (van den Heuvel et al., 2011).



In this chapter we focus on the underlying timer management and illustrate it with fixed-priority servers.

Åsberg et al. (2009) make first steps towards using hierarchical scheduling in the AUTOSAR standard. They sketch what it would take to enable the integration of software components by providing temporal isolation between the AUTOSAR components. In (Nolte et al., 2009) they extend their work to systems where components share logical resources, and describe how to apply the SIRAP protocol (Behnam et al., 2007) for synchronizing access to resources shared between tasks belonging to different components. In this work we consider independent components and focus on minimizing the interference between components due to them sharing the timer management system.

### HSF implementations

Saewong et al. (2002) present the implementation and analysis of an HSF based on deferrable and sporadic servers using an hierarchical rate-monotonic and deadline-monotonic scheduler, as used in systems such as the Resource Kernel (Rajkumar et al., 1998).

Inam et al. (2011) present a FreeRTOS implementation of an HSF, which is based on our earlier work in (Holenderski et al., 2010a). It supports temporal isolation for fixed-priority global and local scheduling of independent tasks, including the support for the idling-periodic and deferrable servers. Their goal is to minimize the changes to the underlying OS. Consequently they rely on absolute timers provided by FreeRTOS. They do not address virtual timers. The HSF presented in this chapter relies on relative times, which allow for an efficient implementation of virtual timers. Also, our HSF implementation is modular and supports both fixed-priority as well as EDF scheduling on both global and local levels, as well as constant-bandwidth servers.

Kim et al. (2000) propose a two-level HSF called the SPIRIT  $\mu$ Kernel, which provides a separation between components by using partitions. Each partition executes a component, and uses the Fixed-Priority Scheduling (FPS) policy as a local scheduler to schedule the component's tasks. An offline schedule is used to schedule the partitions on a global level.

Behnam et al. (2008) present an implementation of a HSF based on the periodic resource model in the VxWorks operating system. They keep track of budget depletion by using separate event queues for each server in the HSF by means of absolute times. Whenever a server is activated (or switched in), an event indicating the depletion of the budget, i.e. the current time plus the remaining budget, is added to the server event queue. On preemption of a server, the remaining budget is updated according to the time passed since the last server release and the budget depletion event is removed from the server event queue. When the server's budget depletion event expires, the server is removed from the server ready queue, i.e. it will not be rescheduled until the replenishment of its budget.

Oikawa and Rajkumar (1999), describe the design and implementation of the Linux/RK, an implementation of a resource kernel (Portable RK) within the Linux kernel. They minimize the modifications to the Linux kernel by introducing a small

number of call back hooks for identifying context switches, with the remainder of the implementation residing in an independent kernel module. Linux/RK introduces the notion of a resource set, which is a set of processor reservations. Once a resource set is created, one or more processes can be attached to it to share its reservations. Although reservations are periodic, periodic tasks inside reservations are not supported. The system employs a replenishment timer for each processor reservation, and a global enforcement timer which expires when the currently running reservation runs out of budget. Whenever a reservation is switched in the enforcement timer is set to its remaining budget. Whenever a reservation is switched out, the enforcement timer is cancelled, and the remaining budget is recalculated.

AQuoSA (Palopoli et al., 2009) also provides the Linux kernel with EDF scheduling and various well-known resource reservation mechanisms, including the constant bandwidth server. Processor reservations are provided as servers, where a server can contain one or more tasks. Periodic tasks are supported by providing an API to sleep until the next period. Similar to Oikawa and Rajkumar (1999) it requires a kernel patch to provide for scheduling hooks and updates the remaining budget and the enforcement timers upon every server switch.

Faggioli et al. (2009) present an implementation of the Earliest Deadline First (EDF) and constant bandwidth servers for the Linux kernel, with support for multi-core platforms. It is implemented directly into the Linux kernel. Each task is assigned a period (equal to its relative deadline) and a budget. When a task exceeds its budget, it is stopped until its next period expires and its budget is replenished. This provides temporal protection, as the task behaves like a hard reservation. Each task is assigned a timer, which is activated whenever a task is switched in, by recalculating the deadline event for the task.

Eswaran et al. (2005) describe Nano-RK, a reservation-based RTOS targeted for use in resource-constrained wireless sensor networks. It supports fixed-priority pre-emptive multitasking, as well as resource reservations for processor, network, sensor and energy. Only one task can be assigned to each processor reservation. Nano-RK also provides explicit support for periodic tasks, where a task can wait for its next period. Each task contains a timestamp for its next period, next replenishment and remaining budget. A one-shot timer drives the timer ISR, which (i) loops through all tasks, to update their timestamps and handle the expired events, and (ii) sets the one-shot timer to the next wakeup time.

Unlike the work presented in (Behnam et al., 2008), which implements a HSF on top of a commercial operating system, and in (Oikawa and Rajkumar, 1999; Faggioli et al., 2009; Palopoli et al., 2009), which implement reservations within Linux, our design for HSF is integrated within a RTOS targeted at embedded systems. Kim et al. (2000) describe a micro-kernel with a two-level HSF and time-triggered scheduling on the global level.

Our design aims at efficiency, in terms of memory and processor overheads, while minimizing the modifications of the underlying RTOS. Unlike Oikawa and Rajkumar (1999); Behnam et al. (2008); Palopoli et al. (2009) it avoids recalculating the expiration of local server events, such as budget depletion, upon every server switch. It also limits the interference of inactive servers on system level by deferring the handling

of their local events until they are switched in. While Behnam et al. (2008) present an approach for limiting interference of periodic idling servers, to the best of our knowledge, our work is the first to also cover deferrable servers.

## Monitoring

Run-time monitoring of the consumed resources is intrinsic to realizing correct implementation of the scheduling and enforcement rules. Monitoring of real-time systems can be classified as synchronous or asynchronous (Chodrow et al., 1991). In the synchronous case, a constraint (e.g. worst-case execution time) is examined by the task itself. In the asynchronous case, a constraint is monitored by a separate task. The approaches in (Chodrow et al., 1991) are based on program annotations and, hence, are synchronous. In reservation-based systems, however, monitoring should be asynchronous to guarantee enforcement without relying on cooperation from tasks. Moreover, monitoring should not interfere with task execution, but should be part of the operating system or middleware that hosts the real-time application. Our HSF takes the asynchronous monitoring approach.

### 3.1.3 Timer management

The two most common ways to represent the timestamps of pending timers are: *absolute* timestamps are relative to a fixed point in time (e.g. January 1st, 1900), while *relative* timestamps are relative to a variable point in time (e.g. the last tick of a periodic timer).

In (Oikawa and Rajkumar, 1999; Palopoli et al., 2009) each timer consists of a 64-bit absolute timestamp and a 32-bit overflow counter. The timers are stored in a sorted linked list. A timer Interrupt Service Routine (ISR) checks for any expiring timers, and performs the actual enforcement, replenishment, and priority adjustments. In (Oikawa and Rajkumar, 1999) the timer ISR is driven by a one-shot high resolution timer which is programmed directly. Palopoli et al. (2009) use the Linux timer interface, and therefore their temporal granularity and latency depend on the underlying Linux kernel.

The Eswaran et al. (2005) implementation is based on the POSIX time structure `timeval`, with two 32-bit numbers to represent seconds/nanoseconds. The authors assume the absolute timestamp value is large enough such that it will practically not overflow.

Carlini and Buttazzo (2003) present the Implicit Circular Timers Overflow Handler (ICTOH), which is an efficient time representation of absolute deadlines in a circular time model. It assumes a periodic timer and absolute time representation. Its main contribution is handling the overflow of the time due to a fixed-size bit representation of time. It requires managing the overflow at every time comparison and is limited to timing constraints which do not exceed  $2^{n-1}$ , where  $n$  is the number of bits of the time representation. Buttazzo and Gai (2006) present an implementation of an EDF scheduler based on ICTOH for the ERIKA Enterprise kernel (Evidence, 2010) and focus on minimizing the tick handler overhead.

The  $\mu\text{C}/\text{OS-II}$  (Labrosse, 2002) real-time operating system stores timestamps relative to the current time. The timers are stored in an unordered queue. It assumes a periodic timer, and at every tick it decrements the timestamp of all pending timers. A timer expires when its timestamp reaches 0. Timestamps are represented as 16-bit integers. The lifetime of their queue is therefore  $2^{16}$  ticks.

In (Holenderski et al., 2009c) we introduced the Relative Timed Event Queues (RELTEQ), which is a timed event management component targeted at embedded operating systems. It supports long event interarrival time (compared to the size of the bit representation for a single timestamp), long lifetime of the event queue, and low memory and processor overheads. By using extra “dummy” events it avoids the need to handle overflows at every comparison due to a fixed bit-length time representation, and allows to vary the size of the time representation to trade the processor overhead for handling dummy events for the memory overhead due to time representation. Similar to (Engler et al., 1995; Kim et al., 2000), our RELTEQ implementation is tick based, driven by a periodic hardware timer.

## 3.2 System model

In this section we specialize the system model presented in Chapter 2 and extend it with notions which are specific to this chapter.

### 3.2.1 Resource model

In this chapter we consider a uniprocessor platform and assume that the task or component which has access to the processor has access to the complete platform. Also, we assume no blocking between tasks and components. Hence, we have  $\mathcal{R} = \{cpu\}$ , with  $N_{cpu} = 1$ ,  $\mathcal{P} = \{cpu\}$  and  $\mathcal{N} = \emptyset$ .

### 3.2.2 Application model

In this chapter we assume a system is composed of independently developed and analyzed subsystems. A subsystem consists of a set of tasks which implement the desired application, a local scheduler, and a server. There is a one-to-one mapping between subsystems and servers.

#### Servers

We consider a set of server components (or simply servers)  $\Sigma \subseteq \mathcal{C}$ , where each server  $s \in \Sigma$  is specified according to Example 2.2.

#### Tasks

We assume that tasks are preemptive and independent, and that each task  $\tau_i$  is mapped to exactly one server.

**Definition 3.1.** We define  $\gamma : \mathcal{C} \rightarrow 2^\Gamma$ , where  $\gamma(c)$  is the set of tasks requiring component  $c$ , i.e.

$$\gamma(c) = \{\tau_i \in \Gamma \mid \exists \tau_{i,j} \in S_i : c \in R_{i,j}\}$$

Since we assume that each task is mapped to exactly one server, we have

$$\bigcup_{s \in \Sigma} \gamma(s) = \Gamma \quad \wedge \quad \bigcap_{s \in \Sigma} \gamma(s) = \emptyset. \quad (3.1)$$

**Definition 3.2.** We define  $\lambda : \Gamma \rightarrow 2^\mathcal{C}$ , where  $\lambda(\tau_i)$  is the set of components required by task  $\tau_i$ , i.e.

$$\lambda(\tau_i) = \{c \in \mathcal{C} \mid \exists \tau_{i,j} \in S_i : c \in R_{i,j}\}$$

**Notation** We will use  $\rho(\tau_i)$  to denote the server which task  $\tau_i$  is mapped to, i.e.  $\rho(\tau_i) = s \in \Sigma \mid \tau_i \in \gamma(s)$ .

Since tasks are independent and mapped to a single server, it is sufficient to model the resource requirements of each task  $\tau_i$  with a single segment  $S_i = \langle \tau_{i,1} \rangle$ , with  $E_{i,1}$  representing the worst-case execution time of  $\tau_i$ , and  $R_{i,1} = \{(\text{cpu}_{\rho(\tau_i)}, 1)\}$ , where  $\text{cpu}_{\rho(\tau_i)}$  is the virtual processor provided by server  $\rho(\tau_i)$  (see Example 2.2).

### Component states during runtime

During runtime, a component may be in one of five states: *running*, *ready*, *blocked*, *depleted* or *waiting*. In this section we define these states in terms of our system model.

**Definition 3.3.** We define  $\nu : \mathcal{T} \times \mathcal{R} \rightarrow \mathbb{N}$ , where

$$\nu(t, r) = N_r - \sum_{b \in \mathcal{S}} \sigma(t, b, r)$$

is the number of units of resource  $r$  which are available at time  $t$ .

**Definition 3.4.** We define  $\alpha : \mathcal{T} \times \Gamma \rightarrow \mathcal{S}$ , where  $\alpha(t, \tau_i)$  is the segment of task  $\tau_i$  which is active at time  $t$ .

Note that  $\alpha$  is a partial function, as during the time interval between the completion of a task and its next arrival, the task will not have an active segment.

We will use  $\alpha(t, \tau_i)$  mainly for identifying the resources which are required by task  $\tau_i$  at time  $t$ .

**Definition 3.5.** We define  $R_{\alpha(t, \tau_i)} : \mathcal{T} \times \Gamma \rightarrow 2^\mathcal{R}$ , where  $R_{\alpha(t, \tau_i)}$  is the set of resources required by task  $\tau_i$  at time  $t$ . If  $\alpha(t, \tau_i) = \tau_{i,j}$ , then  $R_{\alpha(t, \tau_i)} = R_{i,j}$ . For those values of  $t$  for which  $\alpha(t, \tau_i)$  is not defined, we assume  $R_{\alpha(t, \tau_i)} = \emptyset$ .

**Definition 3.6.** We say that there is demand for component  $c$  at time  $t$ , referred to by the predicate  $\text{demand}(t, c)$ , iff at time  $t$  there is (i) a segment which is active and is requiring a resource provided by  $c$ , or (ii) a component which has a remaining budget and is requiring a resource provided by  $c$ , i.e.

$$\begin{aligned} \text{demand}(t, c) \equiv & (\exists \tau_i \in \Gamma : R_{\alpha(t, \tau_i)} \cap P_c \neq \emptyset) \vee \\ & (\exists d \in \mathcal{C} : \beta(t, d) > 0 \wedge R_d \cap P_c \neq \emptyset) \end{aligned} \quad (3.2)$$

**Definition 3.7.** A component  $c$  is said to be running at time  $t$ , referred to by the predicate  $\text{running}(t, c)$ , iff at time  $t$  it is scheduled on any of its required resources<sup>1</sup>, i.e.

$$\text{running}(t, c) \equiv \exists r \in R_c : \sigma^C(t, c, r) > 0. \quad (3.3)$$

**Definition 3.8.** A component  $c$  is said to be blocked at time  $t$ , referred to by the predicate  $\text{blocked}(t, c)$ , iff at time  $t$  it requires a resource  $r$  which is owned by a task or component with a lower priority and there are insufficient units of  $r$  available for  $c$ , i.e.

$$\begin{aligned} \text{blocked}(t, c) \equiv & \beta(t, c) > 0 \wedge (\exists (r, n) \in R_c : \nu(t, r) < n \wedge \\ & (\exists \tau_i \in \Gamma : (\sigma^S(t, \tau_i, r) > 0 \wedge \pi_c < \pi_i) \vee \\ & (\exists d \in \mathcal{C} : (\sigma^C(t, d, r) > 0 \wedge \pi_c < \pi_d))). \end{aligned} \quad (3.4)$$

**Definition 3.9.** A component  $c$  is said to be ready at time  $t$ , referred to by the predicate  $\text{ready}(t, c)$ , iff at time  $t$  it requires a resource  $r$  which is owned only by tasks or components with a higher priority and there are insufficient units of  $r$  available for  $c$ , i.e.

$$\begin{aligned} \text{ready}(t, c) \equiv & \beta(t, c) > 0 \wedge (\exists (r, n) \in R_c : \nu(t, r) < n \wedge \\ & (\forall \tau_i \in \Gamma : \sigma^S(t, \tau_i, r) > 0 \Rightarrow \pi_c > \pi_i) \wedge \\ & (\forall d \in \mathcal{C} : \sigma^C(t, d, r) > 0 \Rightarrow \pi_c > \pi_d))). \end{aligned} \quad (3.5)$$

**Definition 3.10.** A component  $c$  is said to be depleted at time  $t$ , referred to by the predicate  $\text{depleted}(t, c)$ , iff its remaining budget has been exhausted, i.e.

$$\text{depleted}(t, c) \equiv \beta(t, c) = 0. \quad (3.6)$$

**Definition 3.11.** A component  $c$  is said to be waiting on demand at time  $t$ , referred to by the predicate  $\text{waiting}(t, c)$ , iff at time  $t$  it has remaining budget but there is no demand for it, i.e.

$$\text{waiting}(t, c) \equiv \beta(t, c) > 0 \wedge \neg \text{demand}(t, c). \quad (3.7)$$

---

<sup>1</sup>According to the scheduling condition 5 in Section 2.3.2, a running component will be scheduled on all of its required resources.

**Deferrable server** The deferrable server by Strosnider et al. (1995) is bandwidth preserving. This means that when a server is switched out because none of its tasks are ready, it will preserve its budget to handle tasks which may become ready later. A deferrable server can be in one of the states shown in Figure 3.1. A server in the *running* state is said to be *active*, and in either *ready*, *waiting* or *depleted* state is said to be *inactive*. A change from inactive to active or vice-versa is accompanied by the server being *switched in* or *switched out*, respectively.

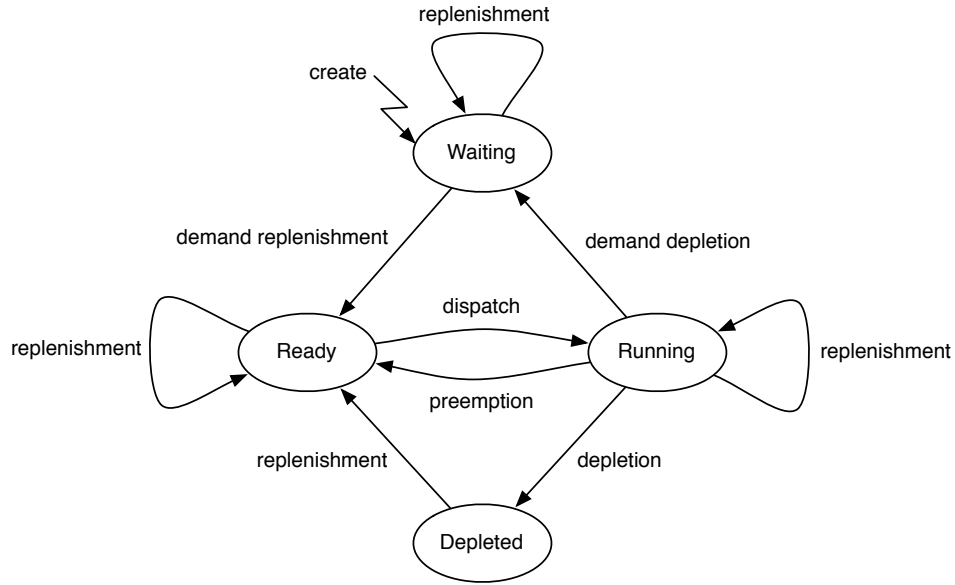


Figure 3.1: State transition diagram for the deferrable server.

A deferrable server  $s \in \mathcal{C}$  is created in the waiting state, with its remaining budget equal to its capacity, i.e.  $\beta(O_s, s) = \Theta_i$ . As soon as there is demand for it, it moves to the ready state. When it is dispatched by the scheduler it moves to the running state. A running server  $s$  may become inactive for one of three reasons:

- It has been preempted by a higher priority server, upon which it preserves its budget and moves to the ready state.
- It has remaining budget, but none of its tasks in  $\gamma(s)$  are ready to run, upon which it preserves its budget and moves to the waiting state.
- Its budget has become depleted, upon which it moves to the depleted state.

When a depleted server is replenished it moves to the ready state and becomes eligible to run. A waiting server may be woken up by a newly arrived periodic task or a delay event.

**Idling periodic server** When the idling periodic server by Davis and Burns (2005) is replenished and none of the tasks in  $\gamma(s)$  is ready, then it idles its budget away until either a task arrives or the budget depletes. An idling periodic server follows the state transition diagram in Figure 3.2. It resembles the state transition diagram

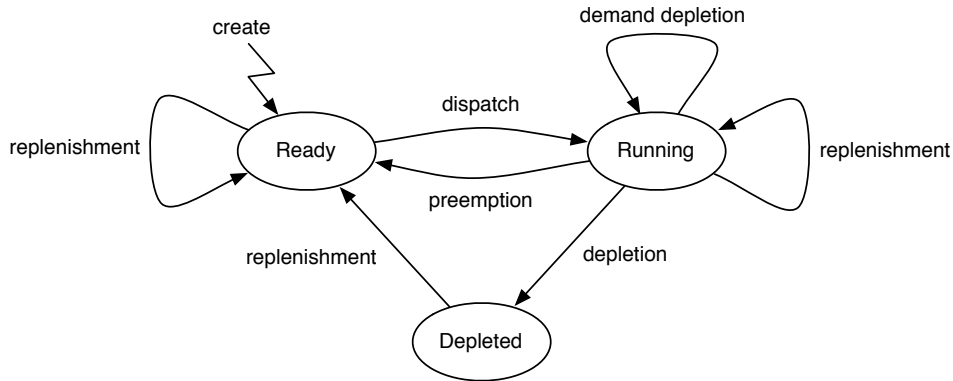


Figure 3.2: State transition diagram for the periodic-idling server.

of a deferrable server, however, due to its idling nature the periodic-idling server will never reach the waiting state. The idling property can be regarded as artificial demand for it when there is no proper demand from its tasks.

**Polling server** The polling server by Lehoczky et al. (1987) is not bandwidth preserving. Therefore, when it is replenished and none of the tasks in  $\gamma(s)$  is ready, or when its workload is exhausted, then its budget is immediately depleted.

A polling server can be in one of three states, shown in Figure 3.3. The difference

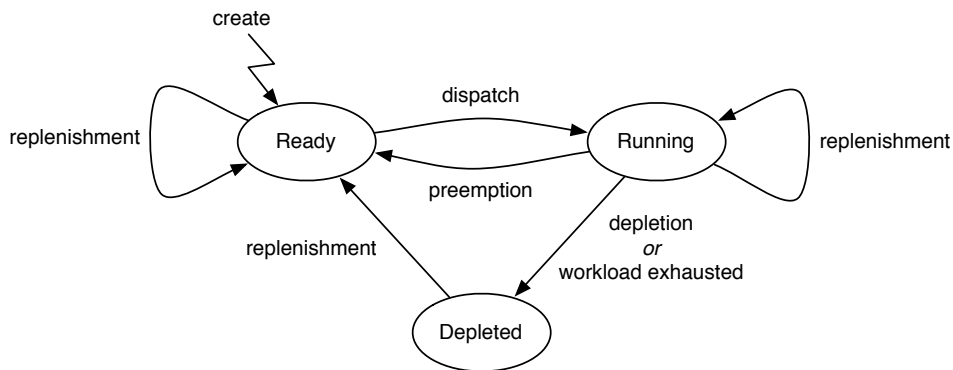


Figure 3.3: State transition diagram for the polling server.

between a polling and a deferrable server lies in what happens when the workload of



a running server is exhausted. Rather than moving to the waiting state (and thus preserving its budget), the polling server discards any remaining budget and moves to the depleted state.

### Hierarchical scheduling

In two-level hierarchical scheduling one can identify a global scheduler which is responsible for selecting a server component. The server is then free to use any local scheduler to select a task to run.

In order to facilitate the reuse of existing server components when integrating them to form larger systems, the platform should support (at least) fixed-priority preemptive scheduling at the local level within servers (since it is a de-facto standard in the industry). To give the system designer the most freedom it should support arbitrary schedulers at the global level. In this chapter we will focus on a fixed-priority scheduler on both local and global level.

### Timed events

The platform needs to support at least the following timed events: task delay, arrival of a periodic task, server replenishment and server depletion, which are generated by the timer handler.

Events local to server  $s$ , such as the arrival of periodic tasks in  $\gamma(s)$ , should not interfere with other servers, unless they wake a server, i.e. the time required to handle them should be accounted to  $s$ , rather than the currently running server. In particular, handling the events local to inactive servers should not interfere with the currently active server and should be deferred until the corresponding server is switched in.

## 3.3 RELTEQ

Our goal in this chapter is to extend the real-time operating system  $\mu C/OS-II$  with an HSF. To implement the desired extensions in  $\mu C/OS-II$  we needed a general mechanism for different kinds of timed events, exhibiting low runtime overheads. This mechanism should be expressive enough to easily implement higher level primitives, such as periodic tasks, fixed-priority servers and two-level fixed-priority scheduling.

### 3.3.1 RELTEQ time model

RELTEQ stores the arrival times of future events relative to each other, by expressing their time *relative to their previous event*. The arrival time of the head event is relative to the current time<sup>2</sup>, as shown in Figure 3.4.

---

<sup>2</sup>Later in this chapter we will use RELTEQ queues as an underlying data structure for different purposes. We will relax the queue definition: all event times will be expressed relative to their previous event, but the head event will not necessarily be relative to “now”.

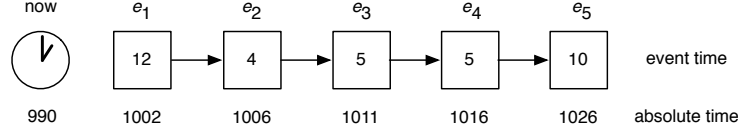
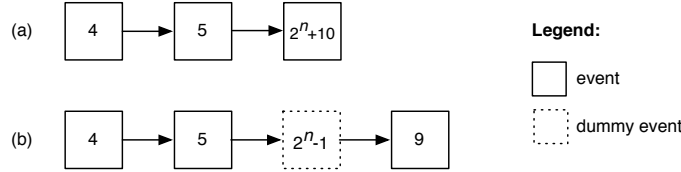


Figure 3.4: Example of the RELTEQ event queue.

### Unbounded interarrival time between events

One of our requirements is for long event interarrival times with respect to time representation. In other words, given  $d$  as the largest value that can be represented for a fixed bit-length time representation, we want to be able to express events which are  $kd$  time units apart, for some parameter  $k > 1$ .

For an  $n$ -bit time representation, the maximum interval between two consecutive events in the queue is  $2^n - 1$  time units<sup>3</sup>. Using  $k$  events, we can therefore represent event interarrival time of at most  $k(2^n - 1)$ . RELTEQ improves this interval even further and allows for an *arbitrarily long interval* between *any* two events by inserting “dummy” events, as shown in Figure 3.5.

Figure 3.5: Example of (a) an overflowing relative event time (b) RELTEQ inserting a dummy event with time  $2^n - 1$  to handle the overflow.

If  $t$  represents the event time of the last event in the queue, then an event  $e_i$  with a time larger than  $2^n - 1$  relative to  $t$  can be inserted by first inserting dummy events with time  $2^n - 1$  at the end of the queue until the remaining relative time of  $e_i$  is smaller or equal to  $2^n - 1$ .

In general, dummy events act as placeholders in queues and can be assigned any time in the interval  $[0, 2^n - 1]$ .

### 3.3.2 RELTEQ data structures

A RELTEQ *event* is specified by the tuple  $(kind, time, data)$ . The *kind* field identifies the event kind, e.g. a delay or the arrival of a periodic task. *time* is the event time. *data* points to additional data that may be required to handle the event and depends on the event kind. For example, a delay event will point to the task which is to be resumed after the delay event expires. Decrementing an event means decrementing its event time and incrementing an event means incrementing its event time. We will

<sup>3</sup>With  $n$  bits we can represent  $2^n$  distinct numbers. Since we start at 0, the largest one is  $2^n - 1$ .

use a dot notation to represent individual fields in the data structures, e.g.  $e_i.time$  is the event time of event  $e_i$ .

A RELTEQ *queue* is a list of RELTEQ events.  $Head(q_i)$  represents the head event in queue  $q_i$ .

### 3.3.3 RELTEQ tick handler

While RELTEQ is not restricted to any specific hardware timer, in this chapter we assume a periodic timer which invokes the *tick handler*, outlined in Figure 3.6.

The tick handler is responsible for managing the *system queue*, which is a RELTEQ queue keeping track of all the timed events in the system. At every tick of the periodic timer the time of the head event in the queue is decremented. When the time of the head event is 0, then the events with time equal to 0 are popped from the queue and handled.

The scheduler is called at the end of the tick handler, but only in case an event was handled. If no event was handled the currently running task is resumed.

The behavior of a RELTEQ tick handler is summarized in Figure 3.6.

```

Head(system).time := Head(system).time - 1;
if Head(system).time = 0 then
  repeat
    HandleEvent(Head(system));
    PopEvent(system);
  until Head(system).time > 0
  Schedule();
end if

```

Figure 3.6: Pseudocode for the RELTEQ tick handler.

How an event is handled by *HandleEvent()* depends on its kind. E.g. a *delay event* will resume the delayed task. In general, the event handler will often use the basic RELTEQ primitives, as described in the following sections.

Note that the tick granularity dictates the granularity of any timed events driven by the tick handler: e.g. a server's budget can be depleted only upon a tick. High resolution one-shot timers (e.g. High Precision Event Timers) provide a fine grained alternative to periodic ticks. In case these are present, RELTEQ can easily take advantage of the fine time granularity by setting the timer to the expiration of the earliest event among the active queues. The tick based approach was chosen due to lack of hardware support for high resolution one-shot timers on our example platform. In case such a one-shot timer is available, our RELTEQ based approach can be easily modified to take advantage of it.

### 3.3.4 Basic RELTEQ primitives

A new event can be created using the following method:

*Event* **NewEvent**(Kind ***k***, Time ***t***, Data ***d***) Creates and returns a new event  $e_i$  with  $e_i.kind = k$ ,  $e_i.time = t$ , and  $e_i.data = d$ .

Three operations can be performed on an event queue: a new event can be inserted, the head event can be popped, and an arbitrary event in the queue can be deleted:

*void* **InsertEvent**(Queue ***q<sub>i</sub>***, Event ***e<sub>j</sub>***) Inserts event  $e_j$  into queue  $q_i$ .

*void* **PopEvent**(Queue ***q<sub>i</sub>***) Removes the earliest event from  $q_i$ .

*void* **DeleteEvent**(Queue ***q<sub>i</sub>***, Event ***e<sub>j</sub>***) Removes event  $e_j$  from queue  $q_i$ .

### 3.3.5 Event queue implementation

The most straightforward RELTEQ queue implementation is probably a doubly linked list:

*void* **InsertEvent**(Queue ***q<sub>i</sub>***, Event ***e<sub>j</sub>***) When a new event  $e_j$  with absolute time  $t_j$  is inserted into the event queue  $q_i$ , the queue is traversed accumulating the relative times of the events until a later event  $e_k$  is found, with absolute time  $t_k \geq t_j$ . When such an event is found, then (i)  $e_j$  is inserted before  $e_k$ , (ii) its time  $e_j.time$  is set relative to the previous event, and (iii) the arrival time of  $e_k$  is set relative to  $e_j$  (i.e.  $t_k - t_j$ ). If no later event was found, then  $e_j$  is appended at the end of the queue, and its time is set relative to the previous event.

*void* **PopEvent**(Queue ***q<sub>i</sub>***) When an event is popped from a queue it is simply removed from the head of the queue  $q_i$ .

*void* **DeleteEvent**(Queue ***q<sub>i</sub>***, Event ***e<sub>j</sub>***) Since all events in a queue are stored relative to each other, the time  $e_j.time$  of any event  $e_j \in q_i$  is critical for the integrity of the events later in the queue. Therefore, before an event  $e_j$  is removed from  $q_i$ , its event time  $e_j.time$  is added to the following event in  $q_i$ .

Note that the addition could overflow. In such case, instead of adding  $e_j.time$  to the following event in  $q_i$ , the kind of  $e_j$  is set to a dummy event and the event is not removed. If  $e_j$  is the last event in  $q_i$  then it is simply removed, together with any dummy events preceding it.

The time complexity of the *InsertEvent()* operation is then linear in the number of events in the queue, while the complexity of the *DeleteEvent()* and *PopEvent()* operations is constant.

The linear time complexity of the insert operation may be inconvenient for large event queues. An alternative implementation based on a heap or a balanced binary

tree may seem more appropriate, as it promises logarithmic time operations. However, as the following theorem states, the relative time representation coupled with the requirement for long event interarrival times (compared to the time representation) make such an implementation impossible.

**Theorem 3.1.** *Assume that the maximum value we can represent in the time representation is  $d$  and also assume that we store times in a tree using relative values no greater than  $d$ . Finally, assume that any two events in the tree are at most  $kd$  apart in real time, for some parameter  $k$ . Then a logarithmic time retrieval of an event from a tree is not possible.*

*Proof.* If there are  $k$  events, the largest time span these  $k$  events can represent is  $kd$  time units, i.e., the time difference between the first and last event can be at most  $kd$  units. If we are to obtain this value by summing over a path this path has to be of length  $k$  which leads to a linear representation. This argument pertains to any representation that sums contributions over a path.  $\square$

We can illustrate Theorem 3.1 using dummy events: assuming that we start at time 0, the real time of a newly inserted event is at most  $kd$ . We would need to insert dummy events until a root path can contain this value. This means we would need to add dummy events until there is a root path of length  $k$ .

Conversely, if we assume a tree representation, then we would like to obtain  $kd$  as a sum of  $\log(k)$  events. If we assume an even distribution over all events, which is the best case with respect to the number of bits required for the time representation, then each event time will be equal to  $\frac{k}{\log(k)}d$ . This means that  $\left\lceil \log \left( \frac{k}{\log(k)} \right) \right\rceil$  extra bits are needed. Therefore, in a tree implementation one cannot limit the time representation to a given fixed value, independent of  $kd$  (i.e. the tree span).

In order to satisfy our initial requirement for long event interarrival time, we chose for a linked-list implementation of RELTEQ queues. In future work we look into relaxing this requirement.

### 3.4 Periodic tasks

The task concept is an abstraction of a program text. There are roughly three approaches to periodic tasks, depending on the primitives the operating system provides. Figure 3.7 illustrates the possible implementations of periodic tasks, where function  $f_i()$  represents the body of task  $\tau_i$  (i.e. the actual work done during each job of task  $\tau_i$ ).

In Figure 3.7.a, the periodic behavior is programmed explicitly while in Figure 3.7.b this periodicity is implicit. The first syntax is typical for a system without support for periodicity, like  $\mu\text{C}/\text{OS-II}$ . It provides two methods for managing time: *GetTime()* which returns the current time, and *DelayFor( $t$ )* which delays the execution of the current task for  $t$  time units relative to the time when the method was called. As an important downside, the approach in Figure 3.7.a may give rise to jitter, when the task is preempted between  $\text{now} := \text{GetTime}()$  and *DelayFor()*.

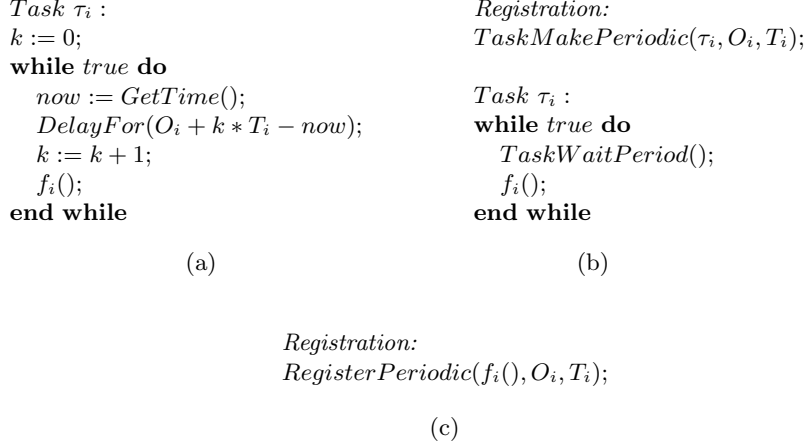


Figure 3.7: Possible implementations of a periodic task.

In order to go from Figure 3.7.a to 3.7.c we extract the periodic timer management from the task in two functions: a registration of the task as periodic and a synchronization with the timer system. A straightforward implementation of *TaskWaitPeriod()* is a suspension on a semaphore. Note that we wait at the beginning of the while loop body (rather than at the end) in case  $O_i > 0$ . Going from interface in Figure 3.7.b to 3.7.c is now a simple implementation issue.

Note that the task structure described in Figure 3.7.b guarantees that a job will not start before the previous job has completed, and therefore makes sure that two jobs of the same task will not overlap if the first job's response time exceeds the task's period.

## RELTEQ primitives for periodic tasks

In order to provide the periodic task interface in 3.7.b, we need to implement a timer which expires periodically and triggers the task waiting inside the *TaskWaitPeriod()* call.

To support periodic tasks we introduce a new kind of RELTEQ events: a *period event*. Each period event  $e_i$  points to a task  $\tau_i$ . The expiration of a period event  $e_i$  indicates the arrival of a periodic task  $\tau_i$  upon which (i) the event time of the  $e_i$  is set to  $T_i$  and reinserted into the system queue using *InsertEvent()*, and (ii) the semaphore blocking  $\tau_i$  is raised.

To support periodic tasks we have equipped each task with three additional attributes: *TaskPeriod*, expressed in the number of ticks, *TaskPeriodSemaphore*, pointing to the semaphore guarding the release of the task, and *TaskPeriodEvent*, pointing to a RELTEQ period event. For efficiency reasons we have added these directly to the Task Control Block (TCB), which is the  $\mu C/OS-II$  structure storing the state in-

formation about a task. Our extensions could, however, reside in a separate structure pointing back to the original TCB.

A task  $\tau_i$  is made periodic by calling  $TaskMakePeriodic(\tau_i, O_i, T_i)$ , which

1. sets the *TaskPeriod* to  $T_i$ ,
2. removes the *TaskPeriodEvent* from the system queue using  $DeleteEvent()$ , in case it was already inserted by a previous call to  $TaskMakePeriodic()$ , otherwise creates a new period event using  $NewEvent(period, T_i, \tau_i)$  and assigns it to *TaskPeriodEvent*.
3. sets the event time of the *TaskPeriodEvent* to  $O_i$  if  $O_i > 0$  or  $T_i$  if  $O_i = 0$ , and inserts it into the system queue.

## 3.5 Servers

A server  $s$  is created using  $ServerCreate(\Pi_i, \Theta_i, kind)$ , where *kind* specifies whether the server is *idling periodic* or *deferrable*. A task  $\tau_i$  is mapped to server  $s$  using  $ServerAddTask(s, \tau_i)$ .

In Section 3.3.3 we have introduced a system queue, which keeps track of pending timed events. For handling periodic tasks assigned to servers we could reuse the system queue. However, this would mean that the tick handler would process the expiration of events local to inactive servers within the budget of the running server.

In order to limit the interference from inactive servers we would like to separate the events belonging to different servers. For this purpose we introduce additional RELTEQ queues for each server. We start this section by introducing additional primitives for manipulating queues, followed by describing how to use these in order to implement fixed-priority servers.

### 3.5.1 RELTEQ primitives for servers

We introduce the notion of a pool of queues, and define two pools: *active queues* and *inactive queues*. They are implemented as lists of RELTEQ queues. Conceptually, at every tick of the periodic timer the heads of all active queues are decremented. The inactive queues are left untouched.

To support servers we extend RELTEQ with the following methods:

*void **ActivateQueue**(Queue  $q_i$ )* Moves queue  $q_i$  from the inactive pool to the active pool.

*void **DeactivateQueue**(Queue  $q_i$ )* Moves queue  $q_i$  from the active pool to the inactive pool.

*void **IncrementQueue**(Queue  $q_i$ )* Increments the head event in queue  $q_i$  by 1. Time overflows are handled by setting the overflowing event to  $2^n - 1$  and inserting a new dummy event at the head of the queue with time equal to the overflow (i.e. 1).

*void SyncQueueUntilEvent(Queue  $q_i$ , Queue  $q_j$ , Event  $e_k$ )* Synchronizes queue  $q_i$  with queue  $q_j$  until event  $e_k \in q_j$ , by conceptually computing the absolute time of  $e_k$ , and then popping and handling all the events in  $q_i$  which have occurred during that time interval.

### 3.5.2 Limiting interference of inactive servers

To support servers, we add an additional *server queue* for *each* server  $s$ , denoted by  $s.sq$ , to keep track of the events local to the server, i.e. delays and periodic arrival of tasks  $\tau_j \in \gamma(s)$ . At any time at most one server can be active; all other servers are inactive. The additional server queues make sure that the events local to inactive servers do not interfere with the currently active server.

When the active server is switched out (e.g. because a higher priority server is resumed, or the active server gets depleted) then the active server queue is deactivated by calling *DeactivateQueue( $s.sq$ )*. As a result, the queue of the switched out server is “paused”. When a server  $s$  is switched in then its server queue is activated by calling *ActivateQueue( $s.sq$ )*. As a result, the queue of the switched in server is “resumed”. The system queue is never deactivated.

In this new configuration the hardware timer drives two event queues:

1. the *system queue*, keeping track of system events, i.e. the replenishment of periodic servers,
2. the *server queue* of the *active* server, keeping track of the events local to a particular server, i.e. task delays and the arrival of periodic tasks belonging to the server.

To keep track of the time which has passed since the last server switch, we introduce a *stopwatch*. The stopwatch is basically a counter, which is incremented with every tick. In order to handle time overflows discussed in Section 3.3.1, we represent the stopwatch as a RELTEQ queue and use *IncrementQueue(stopwatch)* to increment it.

During the time when a server is inactive, several other servers may be switched in and out. Therefore, next to keeping track of time since the last server switch, for each server we also need to keep track of how long it was inactive, i.e. the time since that particular server was switched out. Rather than storing a separate counter for each server, we multiplex the stopwatches for all servers onto the single stopwatch which we have already introduced, exploiting the RELTEQ approach. We do this by inserting a *stopwatch event*, denoted by  $s.se$ , at the head of the stopwatch queue using *InsertEvent(stopwatch,  $s.se$ )* whenever server  $s$  is switched out. The event points to the server and its time is initially set to 0. The behavior of the tick handler with respect to the stopwatch remains unchanged: upon every tick the head event in the stopwatch queue is incremented using *IncrementQueue(stopwatch)*.

During runtime the stopwatch queue will contain one stopwatch event for every inactive server (the stopwatch event for the currently active server is removed when the server is switched in). The semantics of the stopwatch queue is defined as follows:



the accumulated time from the head of the queue until (and including) a stopwatch event  $s.se$  represents the time the server  $s$  was switched out.

When a server  $s$  is switched in, its server queue is synchronized with the stopwatch using *SyncQueuesUntilEvent*( $s.sq$ , *stopwatch*,  $s.se$ ), which handles all the events in  $s.sq$  which might have occurred during the time the server was switched out. It accumulates the time in the stopwatch queue until the stopwatch event  $s.se$  and handles all the events in  $s.sq$  which have expired during that time. Then  $s.se$  is removed from the stopwatch queue. When  $s$  is switched out,  $s.se$  with time 0 is inserted at the head of the stopwatch queue.

**Example 3.1.** The stopwatch queue is a great example of RELTEQ's strength. It provides an efficient and concise mechanism for keeping track of the inactive time for *all* servers. Figure 3.8 demonstrates the behavior of the stopwatch queue for an example system consisting of three servers  $A$ ,  $B$  and  $C$ . It illustrates the state of the stopwatch queue at different moments during execution, before the currently running server is switched out and after the next server is switched in.

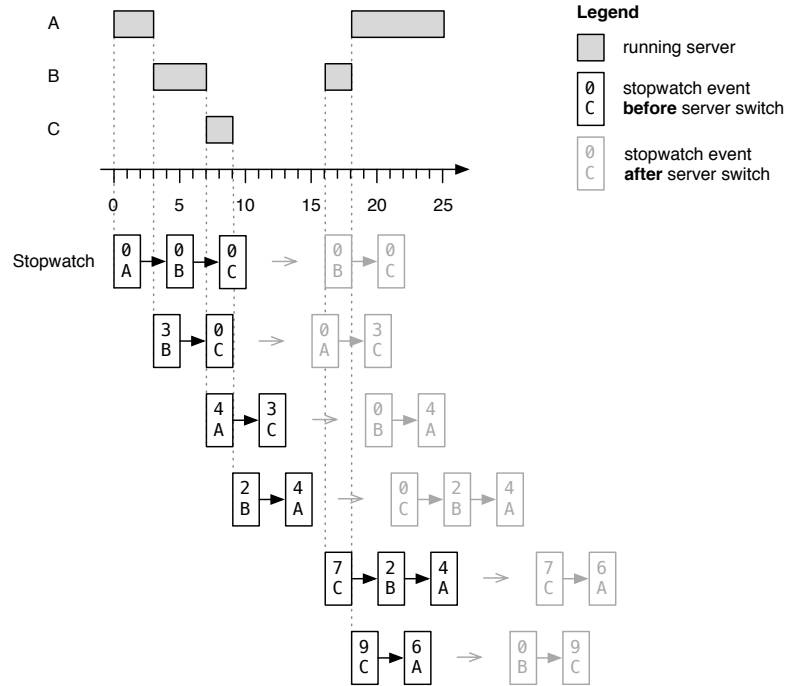


Figure 3.8: Example of the stopwatch queue.

Initially, when server  $s$  is created, a stopwatch event  $s.se$  with time 0 is inserted into the stopwatch queue. At time 0 server  $A$  is switched in and its stopwatch event is removed. While server  $A$  is running, the tick handler increments the head of the stopwatch queue, which happens to be the stopwatch event of server  $B$ . At time 3,

when server  $A$  is switched out and server  $B$  is switched in, server  $B$  synchronizes its absolute queue with the stopwatch queue until and including  $B.se$ ,  $B.se$  is deleted, and  $A.se$  with time 0 is inserted. Note that when  $B.se$  is deleted, its time is added to  $C.se$ .

At time 7 server  $C$  is switched in, its absolute queue is synchronized with time  $4 + 3 = 7$ , after which  $C.se$  is deleted, and  $B.se$  with time 0 is inserted.

At time 9, since no server is switched in, no synchronization is taking place and no stopwatch event is deleted. Only stopwatch event  $C.se$  with time 0 is inserted, since server  $C$  is switched out.

At time 16, when server  $B$  is switched in and its stopwatch event  $B.se$  is deleted, the time of  $B.se$  is added to  $C.se$ .  $\square$

### Deferrable server

When the workload of a deferrable server  $s$  is exhausted, i.e. there are no ready tasks in  $\gamma(s)$ , then the server is switched out and its server queue  $s.sq$  is deactivated. Consequently, any periodic task which could wake up the server to consume any remaining budget cannot be noticed. One could alleviate this problem by keeping its server queue active when  $s$  is switched out. This, however, would make the tick handler overhead linear in the number of deferrable servers, since a tick handler decrements the head events in all active queues (see Section 3.5.6).

Instead, in order to limit the interference of inactive deferrable servers, when a deferrable server  $s$  is switched out and it has no workload pending (i.e. no tasks in  $\gamma(s)$  are ready), we deactivate  $s.sq$ , change its state to waiting, and insert a *wakeup event*, denoted as  $s.we$ , into the system queue. The wakeup event has its *data* pointing to  $s$  and time equal to the arrival of the first event in  $s.sq$ . When the wakeup event expires, the state of  $s$  is set to the ready state. This way handling the events inside  $s.sq$  is deferred until  $s$  is switched in.

### Idling periodic server

An idling periodic server is a special kind of a deferrable server containing an idle task (with lowest priority). The idle task is switched in if no higher priority task is ready, effectively idling away the remaining capacity. In order to save memory needed for storing the task control block and the stack of the idle task, one idle task is shared between all idling periodic servers in the system.

### 3.5.3 Virtual timers

When the server budget is depleted an event must be triggered to guarantee that a server does not exceed its budget. In the next section we present a general approach for handling server depletion. It relies on the notion of *virtual timers*, which are events relative to server's budget consumption. These complement the *global timers*, which operate in the absolute time domain, used for triggering events such as server replenishment or the arrival of periodic tasks.

We can implement virtual timers by adding a *virtual server queue* for *each* server, denoted by  $s.vq$ . Similarly to the server queues introduced earlier, when a server is switched in, its virtual server queue is activated. The difference is that the virtual server queue is not synchronized with the stopwatch queue, since during the inactive period a server does not consume any of its budget. When a server is switched out, its virtual server queue is deactivated.

The relative time representation by RELTEQ allows for a more efficient virtual queue activation than an absolute time representation does. An absolute time representation (e.g. in (Behnam et al., 2008; Inam et al., 2011)) requires to recompute the expiration time for *all* the events in a virtual server queue upon switching in the corresponding server, which is linear in the number of events. In our RELTEQ-based virtual queues the events are stored relative to each other and their expiration times do not need to be recomputed upon queue activation. Note that it will never be necessary to handle an expired virtual event upon queue activation, since such an event would have been already handled before the corresponding server was switched out. Therefore, our HSF design exhibits a constant time activation of a virtual server queue.

### 3.5.4 Server replenishment and depletion

We introduce two additional RELTEQ event kinds to support servers: *server replenishment* and *server depletion*. When a server  $s$  is created, a replenishment event  $e_j$  is inserted into the *system queue*, with  $e_j.data$  pointing to  $s$  and  $e_j.time$  equal to the server's replenishment period  $\Pi_i$ . When  $e_j$  expires,  $e_j.time$  is updated to  $\Pi_i$  and it is inserted into the system queue.

Upon replenishment, the server's depletion event  $e_j$  is inserted into its *virtual server queue*, with  $e_j.data$  pointing to  $s$  and  $e_j.time$  equal to the server's capacity  $\Theta_i$ . If the server was not depleted yet, then the old depletion event is removed from the virtual server queue using  $DeleteEvent(s.vq, e_j)$ .

### 3.5.5 Switching servers

The methods for switching servers in and out are summarized in Figures 3.9 and 3.10.

```

SyncQueuesUntilEvent( $s.sq$ ,  $stopwatch$ ,  $s.se$ );
ActivateQueue( $s.sq$ );
ActivateQueue( $s.vq$ );
if  $s.we \neq \emptyset$  then
    DeleteEvent( $system$ ,  $s.we$ );
     $s.we = \emptyset$ ;
end if

```

Figure 3.9: Pseudocode for  $ServerSwitchIn(s)$ .

```

DeleteEvent(stopwatch, s.se);
s.se = NewEvent(stopwatch, 0, s);
InsertEvent(stopwatch, s.se);
DeactivateQueue(s.sq);
DeactivateQueue(s.vq);
if s.readyTasks =  $\emptyset$  then
    s.we = NewEvent(wakeup, Head(s.sq).time, s);
    InsertEvent(system, s.we);
end if

```

Figure 3.10: Pseudocode for *ServerSwitchOut(s)*.

### 3.5.6 RELTEQ tick handler with support for servers

An example of the RELTEQ queues managed by the tick handler in the proposed RELTEQ extension with servers is summarized in Figure 3.11. Conceptually, every

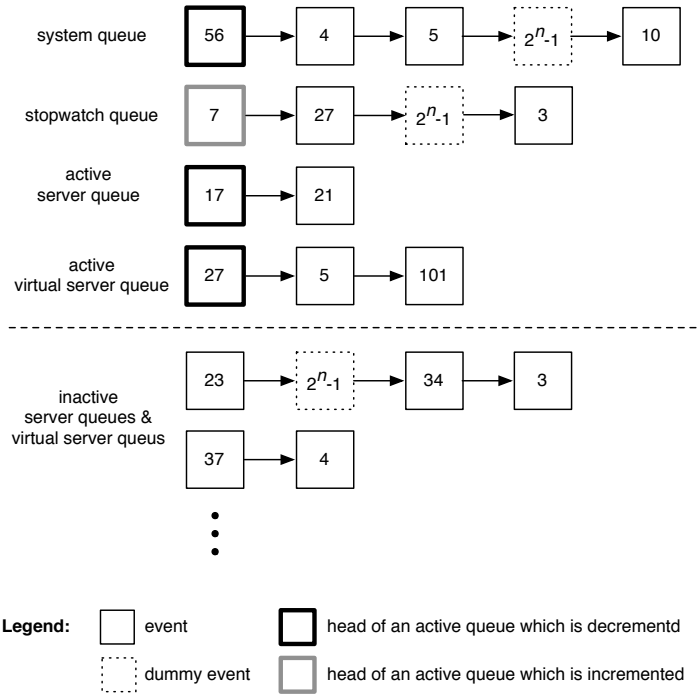


Figure 3.11: Example of the RELTEQ queues managed by the tick handler.

tick the stopwatch queue is incremented and the heads of the system queue, the active server queue and the active virtual server queue are decremented. If the head of any queue becomes 0, then their head event is popped and handled until the queue is

exhausted or the head event has time larger than 0.

Actually, rather than decrementing the head of each active queue and checking whether it is 0, a *CurrentTime* counter is incremented and compared to the *EarliestTime*, which is set whenever the head of an active queue changes. If they are equal, then (i) the *CurrentTime* is subtracted from the heads of all the active queues, (ii) any head event with time 0 is popped and handled, (iii) *CurrentTime* is set to 0, and (iv) *EarliestTime* is set to the earliest time among the heads of all active queues<sup>4</sup>.

The behavior of a RELTEQ tick handler supporting servers is summarized in Figure 3.12.

```

IncrementQueue(stopwatch);
CurrentTime := CurrentTime + 1;
if EarliestTime = CurrentTime then
  for all  $q \in \text{activequeues}$  do
    Head( $q$ ).time := Head( $q$ ).time - CurrentTime;
    while Head( $q$ ).time = 0 do
      HandleEvent(Head( $q$ ));
      PopEvent( $q$ );
    end while
  end for
  CurrentTime := 0;
  EarliestTime := Earliest(activequeues);
  Schedule();
end if

```

Figure 3.12: Pseudocode for the RELTEQ tick handler supporting hierarchical scheduling.

Note that at any moment in time there are at most four active queues, as shown in Figure 3.11. Also, the tick handler is executed with disabled interrupts, guaranteeing mutually exclusive access to the queue data structures.

### 3.5.7 Summary

We have described a generic framework for supporting servers, which is tick based (Section 3.3.3) and limits the interference of inactive servers on system level (Section 3.5.2). The interference of inactive servers which are either ready or depleted was limited by means of a combination of inactive server queues and a stopwatch queue. Deactivating server queues of waiting servers was made possible by inserting a wakeup event into the system queue, in order to wake up the server upon the arrival of a periodic task while the server is switched out.

A large part of our design concerns manipulating events in different event queues. Table 3.1 summarizes which events are stored in which queues. Note that each queue

---

<sup>4</sup>Note that the *time* of any event will never become negative.

can contain dummy events to prolong the event inter-arrival times.

Queue	Events
System queue	Server replenishment, server wakeup
Stopwatch queue	Stopwatch event
Server queue	Periodic task arrival, task delay
Server virtual queue	Server depletion

Table 3.1: Mapping between events and queues.

## 3.6 Hierarchical scheduling

Rajkumar et al. (1998) identified four ingredients necessary for guaranteeing resource provisions: admission, monitoring, scheduling and enforcement. In this section we describe how our implementation of HSF addresses each of them.

### 3.6.1 Admission

We allow admission of new components only during the integration, not during run-time. The admission testing requires analysis for hierarchical systems, which is outside the scope of this chapter.

### 3.6.2 Monitoring

There are two reasons for monitoring the budget consumption of servers: (i) handle the server depletion and (ii) allow the assigned tasks to track and adapt to the available budget.

In order to notice the moment when a server becomes depleted we have introduced a virtual *depletion event* for every server, which is inserted into its virtual server queue. When the depletion event expires, then (i) the server's capacity is set to 0, (ii) its state is set to depleted, and (iii) the scheduler is called.

We would like to allow tasks  $\tau_j \in \gamma(s)$  to track the remaining server budget  $\beta_s$ . Upon such a request, we could sum up the event times of the depletion event of server  $s$  and all the previous events in the virtual server queue. However, this approach would incur an overhead linear in the number of events in the virtual server queue upon every request for the remaining budget. We therefore equipped each server with a *budget counter*. Upon every tick the budget counter of the currently active server is decremented by one. The depletion event will make sure that a depleted server is switched out before the counter becomes negative. We also added the *ServerBudget()* method, which can be called by any task.

*Ticks* **ServerBudget**(*Server s*) Returns the current value of  $\beta_i$ , which represents the lower bound on the processor time that server  $s$  will receive within the remainder of its current period.

### 3.6.3 Scheduling

The  $\mu\text{C}/\text{OS-II}$  scheduler does two things: (i) select the highest priority ready task, and (ii) in case it is different from the currently running one, do a context switch. Our hierarchical scheduler replaces the original *OS\_SchedNew()* method, which is used by the  $\mu\text{C}/\text{OS-II}$  scheduler to select the highest priority ready task.

It first uses the *global* scheduler *HighestReadyServer()* to select the highest priority ready server, and then the server's *local* scheduler *HighestReadyTask()*, which selects the highest priority ready task belonging to that server. This approach allows to implement different global and local schedulers (such as fixed-priority or EDF), and also different schedulers in each server. Our fixed-priority global scheduler is shown in Figure 3.13.

```

highestServer := HighestReadyServer();
if highestServer  $\neq$  currentServer then
    if currentServer  $\neq$   $\emptyset$  then
        ServerSwitchOut(currentServer);
    end if
    if highestServer  $\neq$   $\emptyset$  then
        ServerSwitchIn(highestServer);
    end if
    currentServer := highestServer;
end if
if currentServer  $\neq$   $\emptyset$  then
    return currentServer.HighestReadyTask();
else
    return idleTask;
end if

```

Figure 3.13: Pseudocode for the hierarchical scheduler.

The *currentServer* is a global variable referring to the currently active server. Initially *currentServer* =  $\emptyset$ .

The scheduler first determines the highest priority ready server. Then, if the server is different from the currently active server, a server switch is performed, composed of 3 steps:

1. If there is a currently active server, then it is switched out, using *ServerSwitchOut()* described in Section 3.5.5.
2. If there is a ready server, then it is switched in, using *ServerSwitchIn()* described in Section 3.5.5.
3. The *currentServer* is updated.

Finally the highest priority task in the currently active server is selected, using the current server's local scheduler *HighestReadyTask()*. If no server is active, then the idle task is returned.

### 3.6.4 Enforcement

When a server becomes depleted during the execution of one of its tasks (i.e. if a depletion event expires), the task will be preempted and the server will be switched out. This is possible, since we assume preemptive and independent tasks.

## 3.7 Evaluation

Figure 3.14 shows a trace of an example application consisting of two servers, each serving one task. We ran the setup within the OpenRISC simulator (OpenCores, 2010), with 1 tick set to 1ms<sup>5</sup>. The task execution is shown on top, with the server capacities illustrated underneath. The application was traced and visualized using the Grasp toolset (see Chapter 6).

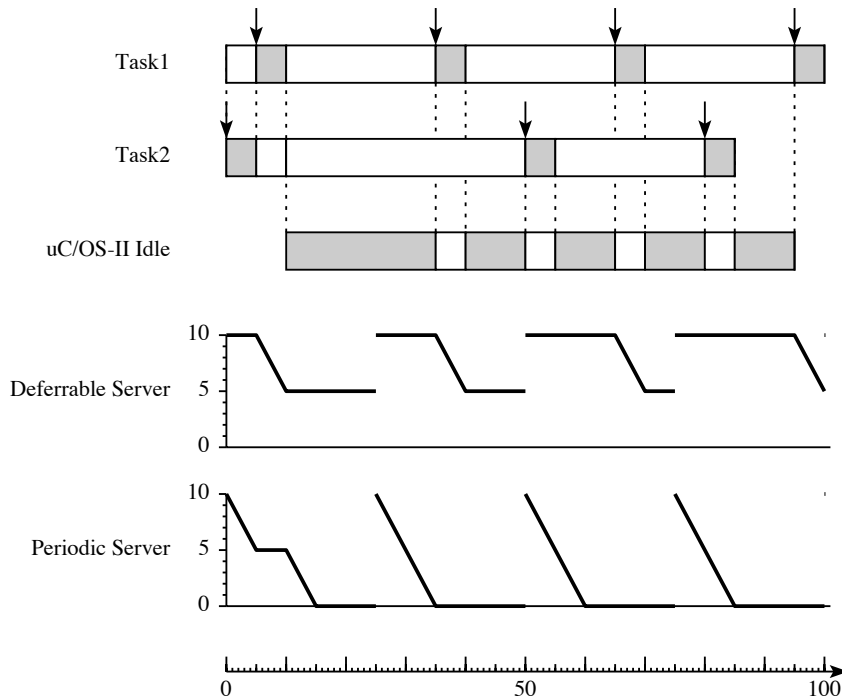


Figure 3.14: Example trace of an application with one deferrable and one periodic idling server.

In this particular example Task1 is assigned to the Deferrable Server, and Task2 is assigned to the Periodic Server. Both tasks have a period of 30ms and execution

<sup>5</sup>The complete simulation setup and the source code of our HSF realization are available at <http://www.win.tue.nl/~mholende/ucos/>.



time of 5ms. Task1 has an offset of 5ms. Both servers have a replenishment period of 25ms and capacity of 10ms. The figure demonstrates, how the idling periodic server idles away its capacity when there is no workload pending (time interval 10-15ms), while the deferrable server preserves its capacity until Task1 arrives (time interval 25-35ms). The figure also illustrates how the handling of period events (indicated by the arrows) of Task2 is postponed until the Periodic Server is switched in (at time 50ms).

In the remainder of this section we evaluate the modularity, memory footprint and performance of the HSF extension for RELTEQ. We chose a linked-list as the data structure underlying our RELTEQ queues and implemented the proposed design within  $\mu C/OS-II$ .

### 3.7.1 Modularity and memory footprint

The design of RELTEQ and the HSF extension is modular, allowing to enable or disable the support for HSF and different server types during compilation with a single compiler directive for each extension.

The complete RELTEQ implementation including the HSF extension is 1610 lines of code (excluding comments and blank lines), compared to 8330 lines of the original  $\mu C/OS-II$ . 105 lines of code were inserted into the original  $\mu C/OS-II$  code, out of which 60 were conditional compilation directives allowing to easily enable and disable our extensions. No original code was deleted or modified. Note that the RELTEQ+HSF code can replace the existing timing mechanisms in  $\mu C/OS-II$ , and that it provides a framework for easy implementation of other scheduler and servers types.

The 105 lines of code represent the effort required to port RELTEQ+HSF to another operating system. Such porting requires (i) redirecting the tick handler to the RELTEQ handler, (ii) redirecting the method responsible for selecting the highest priority task to the HSF scheduler, and (iii) identifying when tasks become ready or blocked.

The memory footprint of RELTEQ+HSF is summarized in Table 3.2. It lists the additional memory required by an application consisting of 6 servers with 6 tasks each using  $\mu C/OS-II$ +RELTEQ+HSF, compared to the memory requirements of an application consisting of 36 tasks (with a stack of 128B each) using the original  $\mu C/OS-II$ .

	$\mu C/OS-II$	RELTEQ+HSF
Code	32KB	8KB
Data	47KB	5KB

Table 3.2: Memory overhead of RELTEQ+HSF.

### 3.7.2 Performance analysis

In this section we evaluate the system overheads of our extensions, in particular the overheads of the scheduler and the tick handler. We express the overhead in terms of the maximum number of events inside the queues which need to be handled in a single invocation of the scheduler or the tick handler, times the maximum overhead for handling a single event.

#### Handling a single event

Handling different events will result in different overheads.

- When a dummy event expires, it is simply removed from the head of the queue. Hence, handling it requires  $O(1)$  time.
- When a task period event expires, an event representing the next periodic arrival is inserted into the corresponding server queue. In this section we assume a linked-list implementation, and consequently insertion is linear in the number of events in a queue. Note that we could delay inserting the next period event until the task completes, as at most one job of a task may be running at a time. This would reduce the handling of a periodic arrival to constant time, albeit at the additional cost of keeping track for each task of the time since its arrival, which would be taken into account when inserting the next period event. However, if we would like to monitor whether tasks complete before their deadline, then we will need to insert a deadline event into  $s.sq$  anyway. Hence the time for handling an event inside of a server queue is linear in the number of events in a server queue. Since there are at most two events per task in a server queue (period and deadline events), handling a period event is linear in the maximum number of tasks assigned to a server, i.e.  $O(m(s))$ .
- When a task deadline event expires, it is simply removed from the head of the queue and the system is notified that a deadline was missed. Hence, handling it requires  $O(1)$  time.
- When a server replenishment event expires, an event representing the next replenishment is inserted into the system queue<sup>6</sup>. Since there are at most two events in the system queue per server (replenishment and wakeup event), handling a replenishment event is linear in the number of servers, i.e.  $O(|\Sigma|)$ .
- When a server depletion event expires, it is simply removed from the queue. Hence, handling it requires  $O(1)$  time.

---

<sup>6</sup>Inserting the next replenishment event could be deferred until the server is depleted, at a similar cost and benefit to deferring the insertion of the task period event.

### Scheduler

Our HSF supports different global and local schedulers, by means of the methods *HighestReadyServer()* and *HighestReadyTask()*. For the sake of a fair comparison with the  $\mu\text{C}/\text{OS-II}$  which implements a fixed-priority scheduler, we also assume fixed-priority global and local schedulers in this section. For both global and local scheduling we can reuse the bitmap-based approach implemented by  $\mu\text{C}/\text{OS-II}$ , which has a constant time overhead for selecting the highest priority ready task as well as indicating whether a task is ready or not (Labrosse, 2002). Consequently, in our HSF we can select the highest priority server and task within a server in constant time.

Once a highest priority server  $s$  is selected, the overhead of switching in the server depends on the number of events inside the stopwatch queue and  $s.sq$  (which needs to be synchronized with the stopwatch), and the overhead of selecting the highest priority task.

The stopwatch queue contains one stopwatch event for each inactive server. The length of the stopwatch queue is therefore bounded by  $|\Sigma| + d_s$ , where  $|\Sigma|$  is the number of servers, and  $d_s = \max_{s \in \Sigma} \left\lfloor \frac{t_s(s)}{2^n - 1} \right\rfloor$  is the maximum number of dummy events inside the stopwatch queue.  $t_s(s)$  is the longest time interval that a server can be switched out, and  $2^n - 1$  is the largest relative time which can be represented with  $n$  bits.

The only local events are a task delay and the arrival of a periodic task. Also, each task can wait for at most one timed event at a time. The number of events inside the server queue is therefore bounded by  $m(s) + d_l(s)$ , where  $m(s)$  is the maximum number of tasks assigned to server  $s$ , and  $d_l(s) = \left\lfloor \frac{t_l(s)}{2^n - 1} \right\rfloor$  is the maximum number of dummy events local to the server queue  $s.sq$ .  $t_l(s)$  is the longest time interval between any two events inside of  $s.sq$  (e.g. the longest task period or the longest task delay).

The complexity of the scheduler is therefore  $O(|\Sigma| + d_s + m(s) + d_l(s))$ . Note that the maximum numbers of dummy events  $d_s$  and  $d_l(s)$  can be determined at design time.

### Tick handler

The tick handler synchronizes all active queues with the current time, and (in case an event was handled) calls the scheduler. The active queues are comprised of the system queue and two queues for the server  $s$  which is active at the time the tick handler is invoked (its server queue  $s.sq$  and virtual server queue  $s.vq$ ).

The system queue contains only replenishment and wakeup events. Its size is therefore proportional to  $|\Sigma| + d_g$ , where  $d_g = \left\lfloor \frac{t_g}{2^n - 1} \right\rfloor$  is the maximum number of dummy events inside the global system queue.  $t_g$  is the longest time interval between any two events inside the global system queue (i.e. the longest server period).

The size of  $s.sq$  is linear in the number of tasks assigned to the server. Similarly, since  $s.vq$  contains one depletion event and at most one virtual timer for each task,

its size is linear in the number of tasks assigned to the server.

Therefore, when server  $s$  is active at the time the tick handler is invoked, the tick handler will need to handle  $t_t(s) = |\Sigma| + d_g + m(s) + d_l(s)$  events. The complexity of the tick handler is therefore  $O(\max_{s \in \Sigma} m(s)t_t(s))$ . Note that the tick handler overhead depends only on tasks belonging to the server  $s$  which is active at the time of the tick. It does not depend on tasks belonging to other servers.

The tick handler keeps track of the remaining budget of servers by decrementing the counter corresponding to the server which is active at the moment of the tick. Consequently, the server that is active at the moment of the tick is charged the entire previous tick. Therefore, any scheduler invocations during a tick interval (including those triggered by the tick handler itself) are executed within the budget of the server which is active at the moment of the following tick.

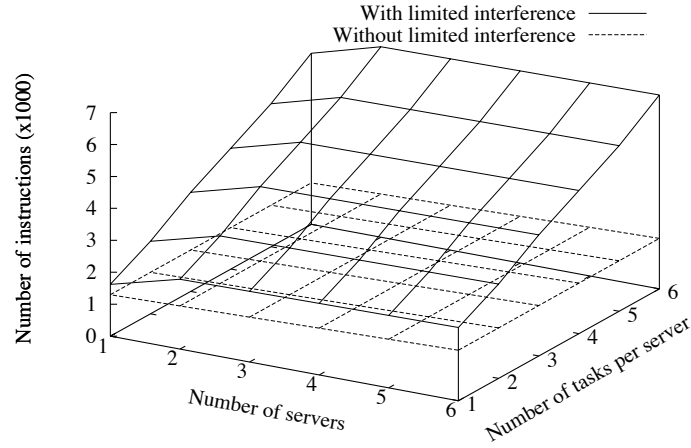
## Experimental results

In Section 3.5.2 we introduced wakeup events in order to limit the interference due to inactive servers. In order to validate this approach we have also implemented a variant of the HSF scheduler which avoids using wakeup events and instead, whenever a deferrable server  $s$  is switched out, it keeps the server queue  $s.sq$  active. Consequently, the scheduler does not need to synchronize the server queue when switching in a server. However, this overhead is shifted to the tick handler, which needs to handle the expired events in all the server queues from inactive deferrable servers. In the following discussion we refer to this approach as *without limited interference*, as opposed to *with limited interference* based on wakeup events.

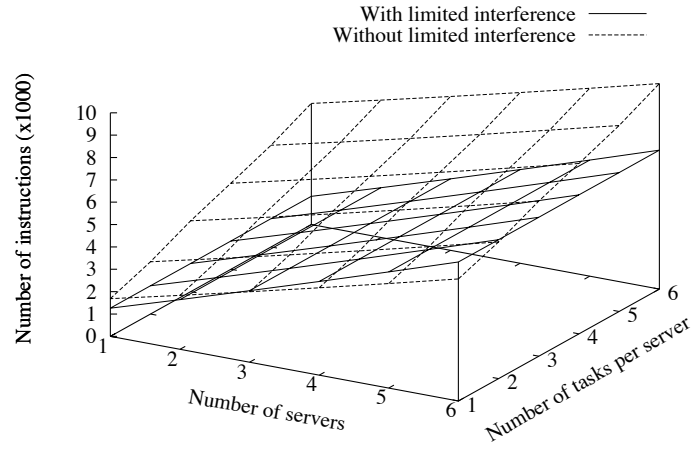
Figures 3.15, 3.16 and 3.17 compare the two variants. We have varied the number of deferrable servers and the number of tasks assigned to each server (all servers had the same number of tasks). The server replenishment periods and the task periods were all set to the same value (100ms), to exhibit the maximum overhead by having all tasks arrive at the same time. Each task had a execution time of 1ms and each server had a capacity of 7ms. We have run the setup within the cycle-accurate hardware simulator for the OpenRISC 1000 architecture (OpenCores, 2010). We have set the processor clock to 345MHz and the tick to 1KHz, which is inline with the platform used by our industrial partner. Each experiment was run for an integral number of task periods.

Figure 3.15 shows the maximum measured overheads of the scheduler and the tick handler, while Figure 3.16 shows the total overhead of the scheduler and the tick handler in terms of processor utilization. The figures demonstrate that wakeup events reduce the tick overhead, at the cost of increasing the scheduler overhead, by effectively shifting the overhead of handling server's local events from the tick handler to the scheduler. Since the scheduler overhead is accounted to the server which is switched in, as the number of servers and tasks per server increase, so does the advantage of the limited interference approach. Figure 3.17.a combines Figures 3.16.a and 3.16.b and demonstrates that the additional overhead due to the wakeup events in the limited interference approach is negligible.

Figure 3.17.b compares the system overheads of our HSF extension to the standard



(a)



(b)

Figure 3.15: (a) maximum overhead of the (local + global) scheduler, (b) maximum overhead of the tick handler.

$\mu C/OS-II$  implementation. As the standard  $\mu C/OS-II$  does not implement hierarchical scheduling, we have implemented a flat system containing the same number of tasks with the same parameters as in Figure 3.17.a. The results demonstrate the temporal isolation and efficiency of our HSF implementation. While the standard

$\mu C/OS-II$  scans through all tasks on each tick to see if any task delay has expired, in the HSF extension the tick handler needs to consider only head event in the server queue of the currently running server.

Figures 3.18 and 3.19 compare the best-case and worst-case measured overheads of the scheduler and tick handler between  $\mu C/OS-II$  with our extensions, compared to the standard  $\mu C/OS-II$ , for which we have implemented a flat system containing the same number of tasks with the same parameters as for the  $\mu C/OS-II+HSF$  case.

Figures 3.18 and 3.19 show that both scheduler and tick handler suffer larger execution time jitter under  $\mu C/OS-II+HSF$ , than the standard  $\mu C/OS-II$ . In the best case the  $\mu C/OS-II+HSF$  tick handler needs to decrement only the head of the system queue, while in  $\mu C/OS-II$  the tick handler traverses all the tasks in the system and for each one it checks whether its timer has expired.

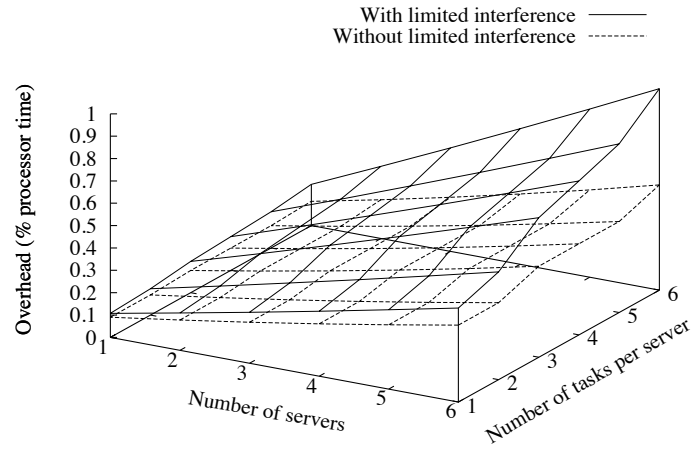
In a system with small utilization of individual tasks and servers (as was the case in our experiments), most local events will arrive while the server is switched out. Since handling local events is deferred until the server is switched in and its server queue synchronized with the stopwatch queue, it explains why the worst-case tick handler overhead is increasing with the number of servers and the worst-case scheduler overhead is increasing with the number of tasks per server.

### 3.8 Discussion

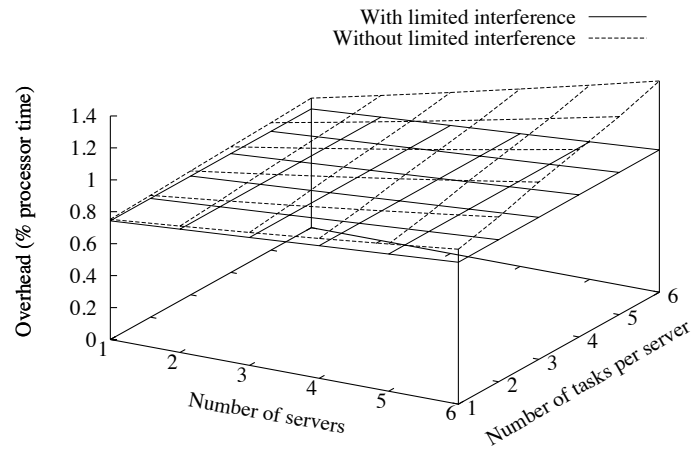
We have presented an efficient, modular and extensible design for enhancing a real-time operating system with a two-level HSF. It relies on Relative Timed Event Queues (RELTEQ), a general timer management system targeting embedded systems. Our design supports various server types (including polling, idling periodic, and deferrable servers), and global and virtual timers. It supports fixed-priority and EDF schedulers on both local and global level. It provides temporal isolation between components and limits the interference of inactive servers on the active server, by means of wakeup events and a combination of inactive server queues with a stopwatch. Enforcement is provided by means of depletion events, which preempt any running task in the depleted server. This is possible since we assume independent tasks. Monitoring is provided by means of an interface which returns a lower bound on the processor time that a server will receive during the following time interval equal to its period.

We have evaluated a fixed-priority based implementation of our RELTEQ and HSF within the  $\mu C/OS-II$  real-time operating system used in the automotive domain. The results demonstrate that our approach exhibits low performance overhead and limits the necessary modifications of the underlying operating system.

We have assumed a linked-list implementation of our RELTEQ queues, and indicated the challenges of a tree-based implementation due to the relative time representation. In the future we want to investigate in more detail other advanced data structures for implementing RELTEQ queues.

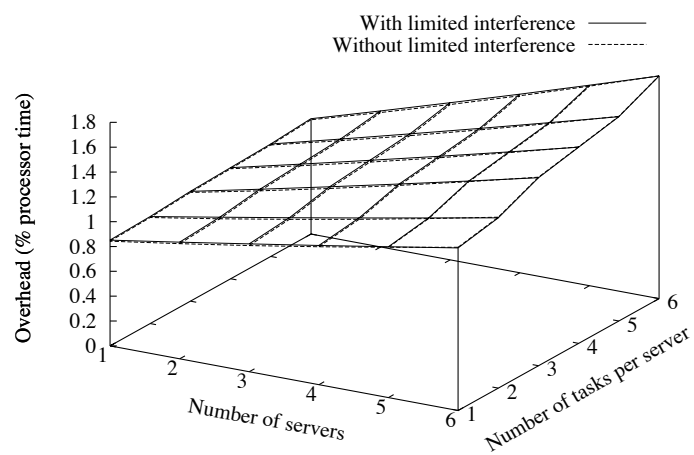


(a)

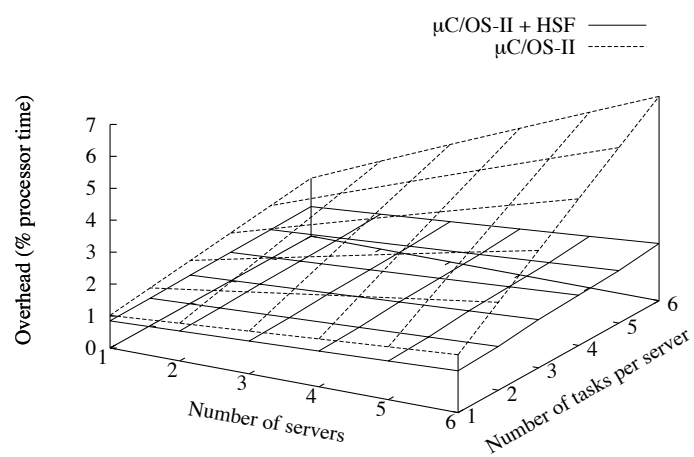


(b)

Figure 3.16: (a) total overhead of the scheduler, (b) total overhead of the tick handler.



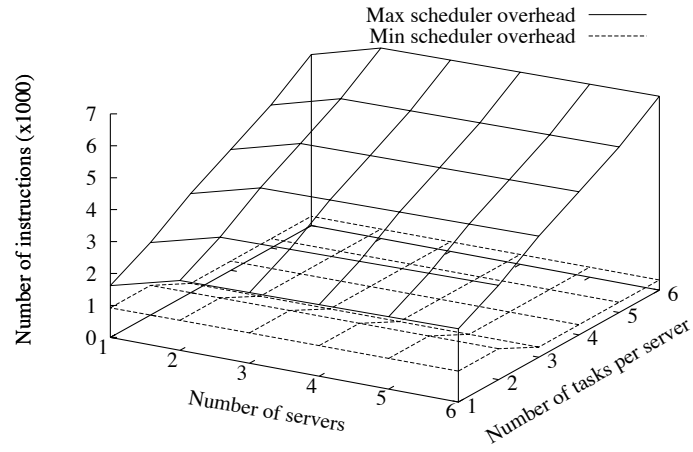
(a)



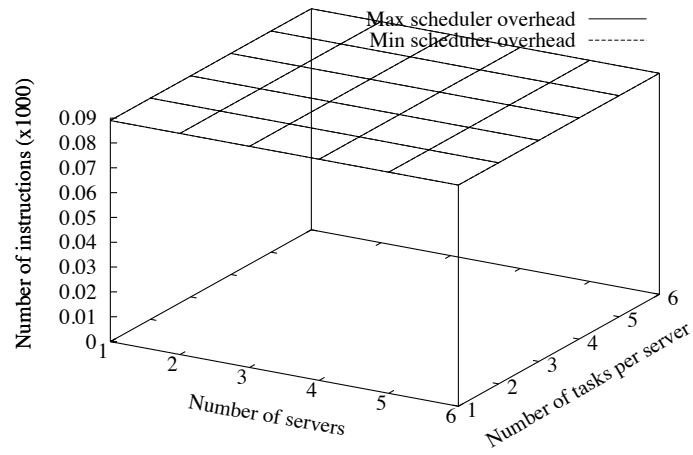
(b)

Figure 3.17: Total overhead of the tick handler and the scheduler.



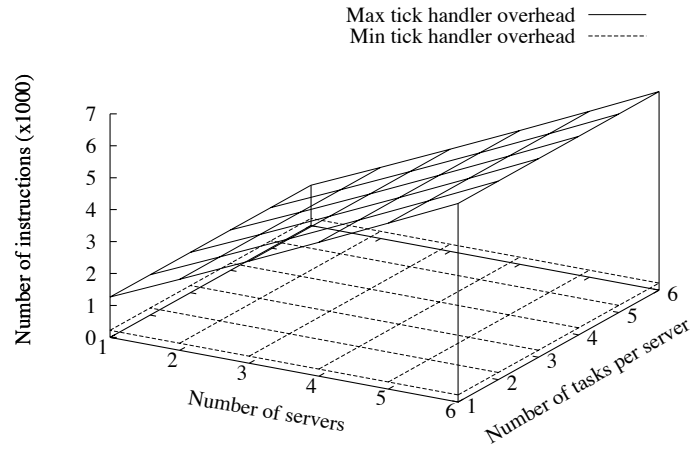


(a)

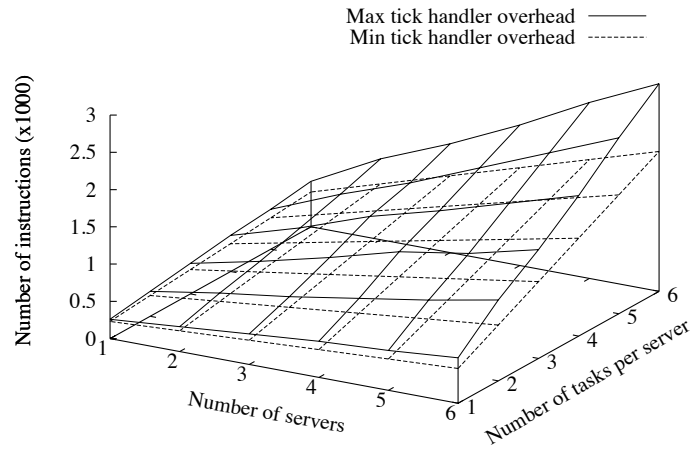


(b)

Figure 3.18: (a) scheduler overhead in  $\mu C/OS-II+HSF$ , (b) scheduler overhead in  $\mu C/OS-II$ .



(a)



(b)

Figure 3.19: (a) tick handler overhead in  $\mu C/OS-II+HSF$ , (b) tick handler overhead in  $\mu C/OS-II$ .

## Chapter 4

# Memory management

In this chapter, we focus on memory management in resource constrained embedded systems, where several applications execute on a single processor. We present a set of general techniques targeted at streaming, or data driven, systems, and use example multimedia processing applications for illustration and motivation.

### Problem statement

In this chapter we address three Quality of Service (QoS) related problems in resource constrained multimedia systems:

- Multimedia applications are known to be data intensive. Many of these applications, especially in the consumer electronics domain, are implemented on resource-constrained embedded systems where the memory space is scarce (Yim et al., 2004; Menichelli and Olivieri, 2009; Ahn et al., 2009). Reducing the memory requirements of these applications is therefore crucial.
- Hentschel et al. (2003) present the theory and practice of video QoS for Consumer Terminals. They describe a pipelined multimedia processing application, where a multiplexed audio and video input stream is demultiplexed into two buffers and processed independently by two processing subchains, as shown in Figure 4.1. They observe that fluctuation in the processing time for decoding different video frames may fill up the buffers along the video processing subchain, leading to the blocking of the demultiplexer and consequently the dropping of audio frames, manifested by sound artifacts. The problem arises due to temporal dependencies between component subchains sharing a common predecessor.
- The third problem we address in this chapter is bounding the mode change latency in scalable multimedia applications, distinguishing between the *application latency* and the *system latency*. From the application perspective, a mode change may interfere with the processing of a video stream. A large

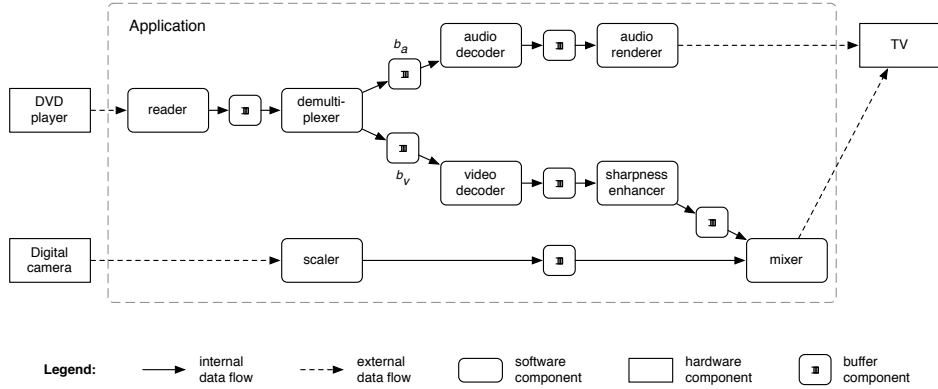


Figure 4.1: Block diagram of the system implementation on a multimedia processor presented in Hentschel et al. (2003). The demultiplexer component writes to two output buffers: the audio output buffer  $b_a$  and the video output buffer  $b_v$ .

mode change latency will delay the rendering of frames and lead to artifacts in the output video. From the system perspective, during a mode change, resources can be allocated to applications in the new mode only *after* they have been released by the applications in the old mode. Therefore the mode change latency will impact when the resources can be reallocated between applications and when these applications can resume operating in the new mode.

## Contributions

The contribution of this chapter is three fold.

- Section 4.3 shows how a shared memory pool can reduce the total memory requirements of an application comprised of a data-driven chain of components communicating via bounded buffers, with a time-driven head and tail, and a bounded end-to-end latency. The necessary buffer capacities and component priorities along the component chain are derived. Subsequently it is shown how a shared memory pool can reduce the total memory requirements of the whole application. The general technique targeted at memory-constrained streaming systems is demonstrated with a video encoding example, showing memory savings of about 19%. The impact of a shared memory pool is also evaluated in the context of scalable applications.
- In Section 4.4, it is shown how additional access to the buffer, in particular the support for dropping selected frames, allows to guarantee the QoS of the application during overload conditions, preventing congestion in one buffer to propagate across the whole application. In particular, we present an approach which prevents the demultiplexer in Figure 4.1 from blocking.

- In Section 4.5, two approaches are presented for reducing the mode change latency bound in scalable multimedia processing applications, which can adapt their memory requirements during runtime according to a set of predefined modes. The first approach relies on fixed-priority scheduling with deferred preemption to shorten the time between a mode change request and the time when the mode change starts. The second approach applies in-buffer scaling to reduce the duration of the mode change itself.

## Publications

In (Holenderski et al., 2011b,c) we describe how to reduce the memory requirements in a multimedia streaming application. In (Holenderski et al., 2009a,b, 2010b) we describe how to reduce the mode change latency in scalable applications.

## 4.1 Related work

### 4.1.1 Memory management in embedded systems

In Geelen (2005), systems are investigated which can change their functionality during runtime, by varying their sets of components. Their notion of dynamic functionality is similar to the concept of scalability discussed in Section 4.1.4, where the total memory requirements are determined by the current application mode. They propose a method for dynamic loading of memory in real-time systems. They present a case study of a DVD player platform, with a memory management unit present but *disabled*, which uses static memory partitioning for memory management.

They propose a method with the memory management unit *enabled* for dividing the memory requirements into static overlays, created during the initialization phase. An overlay groups together the memory requirements of several components belonging to a particular mode (e.g. dvd playback or tv recording). They also show how to manage the loading and storing of overlays between RAM and hard disk by directly controlling the entries in a TLB, and implement their approach in a particular DVD player product. They assume soft deadlines on the mode changes, aiming at providing QoS in terms of “smooth” transitions.

In this thesis we present an approach for dynamic reallocation of memory between components during runtime, on a platform without a memory management unit. We organize the memory requirements of components into memory budgets, which components can request, resize and discard during runtime. We show how the components can resize their budgets during a mode change, before the memory can be reallocated to other components.

Our approach is well suited for platforms with explicitly managed local memory, such as *scratch-pad memory*. A scratch-pad provides low latency data storage, similar to on-chip caches, but under explicit software control. The simple design and predictable nature of scratchpad memories has seen them incorporated into a number of embedded and real-time system processors Brash (2002). McIlroy et al. (2008) present a dynamic run-time heap management algorithm for scratch-pad memory.

Their approach is based on a combination of fixed-sized macro blocks, which are later subdivided into variable sized allocations.

In this thesis, we assume the memory is managed in terms of fixed sized blocks (e.g. single bytes or multiple of bytes), allowing to easily reallocate memory between components. We use the notion of *memory reservations*, which are defined as containers for memory allocations, allowing components to request memory and guarantee granted requests during runtime, avoiding static memory allocation in the absence of paging support. Memory reservations are granted to components only if there is enough space in the common memory pool, and memory allocations are granted only if there is enough space within the corresponding reservation.

#### 4.1.2 Memory reservations

Nakajima (1998) apply memory reservations in the context of continuous media processing applications. They aim at reducing the number of memory pages wired in the physical memory at a time, by wiring only those pages which are used by threads currently processing the media data in the physical memory. Unlike the traditional approaches which avoid page faults by wiring all code, data and stack pages, they introduce an interface to be used by the real-time applications to request reservations for memory pages. During runtime, upon a page fault, the system will load the missing page only if it does not exceed the application's reservation. Thus the number of wired pages is limited.

Eswaran and Rajkumar (2005) implement memory reservations in Linux to limit the time penalty of page faults within the reservation, by isolating the memory management of different applications from each other. They distinguish between *hard* and *firm* reservations, which specify the *maximum* and *minimum* number of pages used by the application, respectively. Their reservation model is hierarchical, allowing child reservations to request space from a parent reservation. Their energy-aware extension of memory reservations allows to maximize the power savings, while minimizing the performance penalty, in systems where different hardware components can operate in different power levels and corresponding performance levels. They also provide an algorithm which calculates the optimal reservation sizes for the task set such that the sum of the task execution times is minimized.

In this thesis we use (hard) memory reservations as containers for memory allocations, allowing components to request memory and guarantee granted requests during runtime, avoiding static memory allocation in the absence of paging support. Memory reservations are granted to components only if there is enough space in the common memory pool, and memory allocations are granted only if there is enough space within the corresponding reservation.

#### 4.1.3 Reducing memory requirements

Optimizing memory usage in resource-constrained devices is becoming increasingly important with the increasing number of mobile and multimedia applications. Yim et al. (2004) present an approach for reducing the internal memory fragmentation in

flash memory for mobile devices. Ahn et al. (2009) consider memory-constrained portable media players and propose a memory allocation scheme for multimedia stream buffers, which allows reducing the number of page faults in heap and thus helps multimedia players perform with a consistent quality. Kim et al. (2010) propose a region reuse technique, based on storing objects in upper local regions to the disk and reusing the reclaimed space for new object allocations, and hence reducing heap memory usage in mobile consumer devices with very limited memory. In contrast to (Yim et al., 2004; Ahn et al., 2009; Kim et al., 2010), which try to manage the memory requirements dictated by the applications, in this thesis we focus on actually reducing the memory requirements.

Weffers-Albu (2008) explores how the assignment of task priorities and buffer capacities impact the behavior of multimedia streaming applications comprised of a task chain. The author shows that a task chain with a time-driven tail exhibiting varying task execution times (where processing a window of  $M$  consecutive frames is bounded by  $MT$ ) will meet its real-time constraints if the last buffer has capacity for  $M$  frames. They also show that a task chain with a time-driven head *and* tail, and  $M = 2$ , will meet its real-time constraints if all buffers have capacity 1. In this thesis we derive the buffer capacities for a chain consisting of a time-driven head *and* tail and an *arbitrary*  $M$ .

Goddard 0 studies the real-time properties of PGM dataflow graphs, which closely resemble our media processing graphs. Given a periodic input and the dataflow attributes of the graph, exact node execution rates are determined for all nodes. The periodic tasks corresponding to each node are then scheduled using a preemptive Earliest Deadline First algorithm. For this implementation of the graph, the author shows how to bound the response time of the graph and the buffer requirements. However, it is limited to task sets with deadlines equal to the period, i.e. without self-interference. This approach provided valuable insights, but no rigorous support for analyzing and steering system behavior and associated resource needs has resulted.

Though we consider variations in execution time, we do not consider overload problems resulting from the inability to process the input in time. Approaches aiming at that situation can be used to prevent this from happening (Isovic and Fohler, 2004; Wüst et al., 2005), or to deal with it when it happens, through scaling or skipping the media content (Lan et al., 2001; Wüst et al., 2005; Jarnikov et al., 2004).

In (Holenderski et al., 2010b) we investigated scalable applications, which can operate in one of several predefined modes, where each *mode* specifies the resource requirements in terms of  $M$  for all the buffers belonging to the application. We showed how to use in-buffer scaling to reduce the memory requirements of each buffer. Upon a mode change request for a mode with a smaller  $M$  and smaller memory requirements, we would drop the least significant frames from the chain and/or reduce the number of blocks for certain frames, and show how this could reduce the mode change latency. In this thesis we concentrate on how a shared memory pool will affect the memory requirements of a scalable application in different modes.

#### 4.1.4 Scalable applications and mode changes

Multimedia processing systems are characterized by few but resource intensive tasks, communicating in a pipeline fashion. The tasks require a processor for processing the video frames (e.g. encoding, decoding, content analysis), and memory for storing intermediate results between the pipeline stages. The pipeline nature of task dependencies poses restrictions on the lifetime of the intermediate results in the memory. The large gap between the worst-case and average-case resource requirements of the tasks is compensated by buffering and their ability to operate in one of several predefined modes, allowing to trade off their *processor* requirements for quality (Wüst et al., 2005), or *network bandwidth* requirements for quality (Jarnikov et al., 2004). In this thesis we investigate trading *memory* requirements for quality. We assume there is no memory management unit available.

(Real and Crespo, 2004) present a comprehensive survey of mode change protocols for FPPS on a single processor, where modes express the application's processor requirements, and propose several new protocols along with their corresponding schedulability analysis and configuration methods. They consider a standard fixed priority sporadic task model, with task  $\tau_i$  specified by its priority  $i$  (lower  $i$  meaning higher priority), minimum interarrival period  $T_i$ , worst-case execution time  $C_i$ , and deadline  $D_i$ , with  $0 < C_i \leq D_i \leq T_i$ .

They classify existing mode change protocols according to three dimensions:

1. Ability to abort old-mode tasks upon a mode change request:
  - (a) All old-mode tasks are immediately aborted.
  - (b) All old-mode tasks are allowed to complete normally.
  - (c) Some tasks are aborted.
2. Activation pattern of unchanged tasks during the transition:
  - (a) Protocols with periodicity, where unchanged tasks are executed independently from the mode change in progress.
  - (b) Protocols without periodicity, where the activation of unchanged periodic tasks may be delayed until the mode change has completed (limiting thus the interference with the tasks involved in the mode change).
3. Ability of the protocol to combine the execution of old- and new- mode tasks during the transition
  - (a) Synchronous protocols, where new-mode tasks are never released until all old-mode tasks have completed their last activation in the old mode.
  - (b) Asynchronous protocols, where a combination of both old- and new-mode tasks are allowed to be executed at the same time during the transition.

(Real, 2000) present a synchronous mode change protocol without periodicity, where upon a mode change request the active old-mode tasks are allowed to complete,



but are not released again (if their next periodic invocation falls within the mode change), classified in the three dimensions as (1b, 2b, 3a). Their algorithm bounds the mode change latency by

$$\mathcal{L} = \sum_{\tau_i \in \Gamma_{old} \cup \Gamma_{unch}} E_i \quad (4.1)$$

where  $\Gamma_{old}$  is the set of tasks in the old mode and  $\Gamma_{unch}$  is the set of unchanged tasks.  $\mathcal{L}$  is the upper bound on the latency of a mode change, i.e. the time interval between a *mode change request* and the time when all the old-mode tasks have been deleted and all the new-mode tasks, with their new parameters, have been added and resumed.

(Sha et al., 1988) consider periodic task executing according to the Rate Monotonic schedule and sharing resources according to the priority ceiling protocol (Sha et al., 1990). Their (1b, 2a, 3b) mode change protocol provides a complicated set of rules for determining when the priority ceilings of shared resources can be changed without conflicting with the delayed release of the new-mode tasks.

The (1b, 2a, 3a) mode change protocol by (Tindell and Alonso, 1996) is applicable when tasks access shared resources according to any resource access policy. Upon a mode change request the protocol waits until an idle instant. When an idle instant is detected, the activation of old-mode tasks is suspended, the mode is changed (e.g. in case of priority ceiling protocol the ceilings of the shared resources are adjusted) and the new-mode tasks are released. This protocol is very simple, however it suffers from a large mode change latency, bounded by the worst-case response time of the lowest priority task.

In this thesis we present a (1b, 2b, 3a) mode change protocol. We assume that the old-mode jobs cannot be aborted at an arbitrary moment, but only at subtask boundaries. Our protocol is as simple as (Tindell and Alonso, 1996), yet improves the mode change latency bound of (Real, 2000). The presented latency bound takes into account resource sharing between tasks according to FPDS or the kernelized monitoring approach (Mok, 1983) for FPPS.

#### 4.1.5 Fixed-Priority Scheduling with Deferred Preemption

On the two sides of the Fixed Priority Scheduling spectrum we have Fixed Priority Non-preemptive Scheduling (FPNS) and Fixed Priority Preemptive Scheduling (FPPS) Liu and Layland (1973). While FPNS favors the lower priority tasks, by postponing preemption by higher priority tasks until a lower priority running task completes, FPPS focuses on the schedulability of higher priority tasks. However, by allowing preemptions at arbitrary moments in time, FPPS ignores the cost of such preemptions. This overhead may become especially significant when tasks share multiple resources, e.g. cache, local or main memory.

Fixed Priority Scheduling with Deferred Preemption (FPDS) was described by Burns (1994); Burns et al. (1994); Gopalakrishnan and Parulkar (1996); Burns (2001); Bril et al. (2007). FPDS, also known as cooperative scheduling (Burns and Wellings, 2001), allows task preemption only at predefined *preemption points*. It finds a middle ground between FPNS and FPPS:

- It reduces the cost of arbitrary preemptions in FPPS, by allowing them only at times convenient for the system (referred to as *preemption points*), e.g. at times where the context switch overhead due to preemption will be smallest. If FPDS is used as a guarding mechanism for critical sections, then there is also no need for access protocols to the shared resources (other than the processor), further reducing the system overheads.
- It can improve the schedulability of systems compared to FPNS by allowing shorter non-preemptive subjobs and thus improves the response-time of higher priority tasks.

FPDS is a generalization of FPPS and FPNS, where FPPS can be modeled by FPDS with arbitrarily short subjobs (ignoring context switch and scheduling overheads), and FPNS by FPDS with tasks consisting of a single subjob.

(Bril et al., 2009) provide a worst case response time analysis for FPDS. They also model a task as a directed acyclic graph of subtasks, where nodes represent subtasks and edges represent the successor relationship between subtasks. The interference of a task on lower priority tasks is bounded by the longest execution time among all paths through its subtask graph.

FPDS can be used to reduce the cost of context switches of preemptive resources, and provide simple access protocol to non-preemptive resources. It offers a simple implementation of critical sections, compared to the intricate priority inheritance protocols used in FPPS, as Pollock and Zöbel (2000); Zöbel et al. (2005) reveal wrong implementation of these protocols in the existing real-time operating systems. In FPDS, subjobs simply execute non-preemptively.

## 4.2 System model

In this section we instantiate the system model introduced in Chapter 2 with a multimedia streaming platform.

### 4.2.1 Resource model

In this chapter we consider a uniprocessor platform and focus on managing two resources: the processor and the memory. Hence, we have  $\mathcal{R} = \{cpu, mem\}$ , where  $N_{cpu} = 1$  and  $N_{mem} \geq 1$ , and  $\mathcal{P} = \{cpu\}$  and  $\mathcal{N} = \{mem\}$ .

### 4.2.2 Application model

An application consists of a chain of  $n$  tasks  $\tau_1, \dots, \tau_n$  communicating via  $n-1$  shared buffers  $q_1, \dots, q_{n-1}$ . Figure 4.2 shows an example of such an application, which is based on the Nexperia TSSA platform investigated by Weffers-Albu (2008).

We assume several applications in our system which contend for the available resources. In the remainder of this chapter we will focus on the behavior of a single application and model other applications by restricting the available resources, either

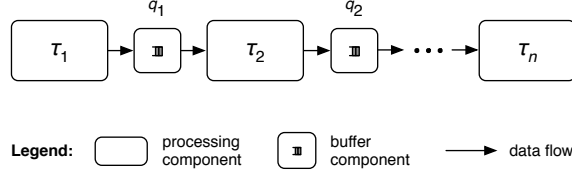


Figure 4.2: A linear chain of media processing tasks communicating via shared buffers.

statically or dynamically during runtime. This will simplify the notation, as we will not need to distinguish between tasks and components belonging to different applications.

### Buffers

A frame buffer  $q$  has a finite initial capacity  $Cap_q$ , defining the maximum number of frames which can be stored in the buffer. To accommodate different frame sizes at different stages in the processing pipeline (e.g. raw video frames are likely to be larger than the encoded frames), we assume frames may span across several buffer elements. To keep things simple, however, we assume that each buffer element contains at most one video frame, i.e. video frames do not share buffer elements. Therefore we assume an  $n$ -to-1 relationship between buffer elements and video frames. A frame buffer  $q$  therefore provides  $NumElems_q = MaxFrameSize * Cap_q$  elements, where  $MaxFrameSize$  is the maximum size of a frame in terms of buffer elements.

Following Example 2.3, a frame buffer, which resides in memory  $mem$  and is created during the initialization of the system and never destroyed, can be modeled as a component  $q \in \mathcal{C}$  with

$$\begin{aligned}
 T_q &= \infty \\
 O_q &= 0 \\
 D_q &= \infty \\
 E_q &= \infty \\
 R_q &= \{(mem, ElemSize_q * NumElems_q)\} \\
 P_q &= \{(elements_q, NumElems_q), (mutex_q, 1)\}
 \end{aligned} \tag{4.2}$$

where  $ElemSize_q$  is the fixed size of each buffer element in terms of bytes.

The interface provides methods for reading and writing, which provide a handshake protocol allowing to do in-memory processing:

**Buffer** *NewBuffer*(Size  $s$ ) allocates the memory for the buffer, i.e. requests a memory budget containing  $s$  units, and returns a reference to the buffer.

**void** *ReadAcquire*(Buffer  $q$ ) removes a full element from the head of buffer  $q$  and returns a reference to that element. This operation blocks if no full frame is available.

*void ReadRelease(Buffer  $q$ , Element  $e$ )* adds the element referred to by  $e$  to buffer  $q$  as an empty element.

*Buffer WriteAcquire(Buffer  $q$ )* removes an empty element from tail of buffer  $q$  and returns a reference to that element. This operation blocks if no empty element is available.

*void WriteRelease(Buffer  $q$ , Element  $e$ )* adds the element referred to by  $e$  to buffer  $q$  as a full element.

*void DeleteBuffer(Buffer  $q$ )* frees the memory allocated to buffer  $q$ , i.e. discards the corresponding memory budget.

### Tasks

The first and the last tasks in the chain are time-driven, with periods  $T_1$  and  $T_n$ , phasings  $O_1$  and  $O_n$ , and relative deadlines  $D_1$  and  $D_n$ , respectively. The time between activations and deadlines represent the times that the application may access the input frame buffer and the output frame buffer respectively. All other tasks in the chain are data-driven, meaning that their arrival time is determined by the availability of data, rather than being determined by their period and offset (as is the case for time-driven tasks). For example, in Figure 4.2, if task  $\tau_2$  is data-driven, then at the moment that it completes processing a frame it will immediately arrive again to process the next frame, unless either  $q_1$  is empty or  $q_2$  is full, in which case it will block until  $q_1$  is not empty or  $q_2$  is not full, respectively.

Clearly, the periods of the head and tail tasks must be consistent with the frame counts since this system can only be expected to work if the inflow equals the outflow. In this chapter we do not consider variations in the number of frames produced or consumed by a task nor any other data dependent behavior than varying execution times. In particular, we assume that every instance of every task along the chain consumes and produces exactly one frame (with the head task only producing one frame and the tail task only consuming one frame). We therefore adopt

$$T_1 = T_n = T. \quad (4.3)$$

**Definition 4.1.** We define  $\tilde{E}_i^k$  to be the execution time of the  $k^{\text{th}}$  instance of task  $\tau_i$ .

Hence,  $\tilde{E}_i^k$  is the execution time needed by task  $\tau_i$  to process the  $k^{\text{th}}$  frame.

Component execution is determined by task priorities, data availability, buffer sizes and time triggering at the boundaries of the system. The execution times of tasks may vary depending on the data they process, however, we assume that after producing the  $k^{\text{th}}$  frame the total execution time needed to produce the following  $M$  frames is (strictly) bounded by  $MT$ .

**Notation** We use  $\hat{E}_i = \max_k \tilde{E}_i^k$  to represent the worst-case execution time of task  $\tau_i$  on any frame. We use  $\tau_{a..b}$  to denote a sub-chain of tasks  $\tau_i$  with  $a \leq i \leq b$ , and  $\vec{E}_{a..b}^k = \sum_{i=a}^b \tilde{E}_i^k$  to denote the execution time needed by chain  $\tau_{a..b}$  to process the  $k^{\text{th}}$  frame.

The pseudo code for a data-driven task is shown in Figure 4.3. In each iteration, task  $\tau_i$  reads a frame from  $q_{i-1}$  using *ReadAcquire*( $q_{i-1}$ , *inFrame*) and retrieves a reference to a buffer slot inside  $q_i$  using *WriteAcquire*( $q_i$ , *outFrame*). While processing the input frame from  $q_{i-1}$  it writes the output frame to the slot it obtained from  $q_i$ . After it has finished processing the frame, it releases the input frame by calling *ReadRelease*( $q_{i-1}$ , *inFrame*) and marks the output frame as ready for reading by calling *WriteRelease*( $q_i$ , *outFrame*).

```

var inFrame, outFrame: Frame reference;
while true do
  outFrame := WriteAcquire( $q_i$ );
  inFrame := ReadAcquire( $q_{i-1}$ );
  Processi(inFrame, outFrame);
  ReadRelease( $q_{i-1}$ , inFrame);
  WriteRelease( $q_i$ , outFrame);
end while

```

Figure 4.3: Pseudo-code for a data-driven task  $\tau_i$ .

The head task has no input buffer and the tail task has no output buffer. Moreover, the head and tail tasks are time-driven, as outlined in Figure 4.4 and Figure 4.5. Note that a time-driven task can block on both communication and time.

```

var outFrame: Frame reference;
    k: integer;
k := 0;
while true do
  DelayUntil( $O_1 + kT_i$ );
  outFrame := WriteAcquire( $q_i$ );
  Process1(inputFrameBuffer, outFrame);
  WriteRelease( $q_i$ , outFrame);
  k := k + 1;
end while

```

Figure 4.4: Pseudo-code for a time-driven head task  $\tau_1$ .

Note that *inputFrameBuffer* and *displayBuffer* represent references to memory which is external to the task chain. *Process*<sub>1</sub>() and *Process*<sub>*n*</sub>() will simply copy frames to and from the local buffers  $q_1$  and  $q_{n-1}$ , respectively.

```

var inFrame: Frame reference;
      k: integer;
k := 0;
while true do
  DelayUntil( $O_n + kT_n$ );
  inFrame := ReadAcquire( $q_{n-1}$ );
  Processn(inFrame, displayBuffer);
  ReadRelease( $q_{n-1}$ , inFrame);
  k := k + 1;
end while

```

Figure 4.5: Pseudo-code for a time-driven tail task  $\tau_n$ .

### Scalable tasks and components

Scalable tasks and components are those which can adapt their workload to the available resources. In this thesis we consider scalable tasks and components which can operate in one of several predefined modes. A *task mode* (or *component mode*) represents a tradeoff between the Quality of Service (QoS) provided by the task (or component) and its resource requirements. We assume a resource constrained system, meaning that not all tasks and components can operate in their highest modes (i.e. modes requiring the most resources) at the same time, forcing some tasks or components to operate in lower modes. A *system mode* identifies the task and component modes of all tasks and components in the system. In order to provide system wide QoS during runtime, the system may decide to switch between system modes, resulting in a *mode change*. We define scalable tasks and components similarly to Real and Crespo (2004).

**Definition 4.2** (System mode). *A system mode identifies the mode of operation of the system. Let  $\mathcal{M} \subset \mathbb{N}$  be the set of system modes. Without loss of generality we assume that  $\mathcal{M}$  is a set of consecutive numbers  $\{1, \dots, |\mathcal{M}|\}$ .*

**Definition 4.3** (Scalable task). *We define  $\Gamma^S \subset \Gamma^{|\mathcal{M}|}$  to be the set of all scalable tasks in the system. A scalable task  $\tau_i \in \Gamma^S$  is represented as a tuple of tasks*

$$\tau_i = (\tau_i^1, \tau_i^2, \dots, \tau_i^{|\mathcal{M}|}) \quad (4.4)$$

where  $|\mathcal{M}|$  is the number of system modes, and each task  $\tau_i^k$  for  $1 \leq k \leq |\mathcal{M}|$  represents the resource requirements and provisions of task  $\tau_i$  during system mode  $k$ .

**Definition 4.4** (Scalable component). *We define  $\mathcal{C}^S \subset \mathcal{C}^{|\mathcal{M}|}$  to be the set of all scalable components in the system. A scalable component  $c \in \mathcal{C}^S$  is represented as a tuple of components*

$$c = (c^1, c^2, \dots, c^{|\mathcal{M}|}) \quad (4.5)$$

where  $|\mathcal{M}|$  is the number of system modes, and each component  $c^k$  for  $1 \leq k \leq |\mathcal{M}|$  represents the resource requirements and provisions of component  $c$  during system mode  $k$ .

**Example 4.1.** Consider a scalable multimedia processing application comprised of two tasks communicating via a FIFO buffer. In a system comprised of two such applications  $a_1, a_2 \in \mathcal{A}$  we will have two buffers. If we assume a memory constrained platform such that the available memory cannot accommodate both applications at their highest quality at the same time, we may need to reallocate the memory between the buffers during runtime to provide a system wide Quality of Service. Assuming the system can operate in two modes (representing more memory assigned to application  $a_1$  or  $a_2$ ), we can model the memory requirements of the two buffers residing in memory  $mem$  by means of scalable components  $c_1, c_2 \in \mathcal{C} \times \mathcal{C}$ , where

$$R_{c_1^1} = R_{c_2^2} = \{(mem, n_{more})\} \quad (4.6)$$

$$R_{c_2^1} = R_{c_1^2} = \{(mem, n_{less})\} \quad (4.7)$$

where  $n_{more}$  and  $n_{less}$  represent the number of units of the  $mem$  resource which are required by the two buffers in the two system modes, with  $n_{more} > n_{less}$  and  $n_{more} + n_{less} \leq N_{mem}$ .  $\square$

A non-scalable task (or component) can be regarded as a degenerate case of a scalable task (or component) which can only operate in a single mode, i.e. all of its task (or component) modes are the same.

At any time a task (or component) is operating in one of its modes.

**Definition 4.5** (Current task mode). We define  $\mu^\Gamma : \mathcal{T} \times \Gamma^S \rightarrow \mathcal{M}$ , where  $\mu^\Gamma(t, \tau_i)$  represents the system mode in which task  $\tau_i$  is operating at time  $t$ .

**Definition 4.6** (Current component mode). We define  $\mu^C : \mathcal{T} \times \mathcal{C}^S \rightarrow \mathcal{M}$ , where  $\mu^C(t, c)$  represents the system mode in which component  $c$  is operating at time  $t$ .

**Definition 4.7** (Mode change request). A mode change request is an event  $e$  which is specified by its

- time  $Time_e \in \mathcal{T}$ , representing the time when the mode change request occurred,
- target mode  $Mode_e \in \mathcal{M}$ .

We use  $\Delta$  to denote the set of all mode change request events.

**Definition 4.8** (Mode change completion time). We define  $mcc : \Delta \rightarrow \mathcal{T}$ , where  $mcc(e)$  is the mode change completion time of the mode change request  $e$ , i.e. the earliest time after a mode change request time  $Time_e$  when all tasks and components are operating in the target mode  $Mode_e$ , i.e.

$$mcc(e) = t \in \mathcal{T} \mid Time_e \leq t \wedge (\forall \tau_i \in \Gamma^S : \mu^\Gamma(t, \tau_i) = Mode_e) \wedge \quad (4.8)$$

$$(\forall c \in \mathcal{C}^S : \mu^C(t, c) = Mode_e) \wedge \quad (4.9)$$

$$(\forall x \in \mathcal{T} : Time_e \leq x < t \Rightarrow \quad (4.10)$$

$$(\exists \tau_i \in \Gamma^S : \mu^\Gamma(x, \tau_i) \neq Mode_e) \wedge \quad (4.11)$$

$$(\exists c \in \mathcal{C}^S : \mu^C(x, c) \neq Mode_e)). \quad (4.12)$$

Note that at some points in time, for example during a mode change, different tasks or components may be operating in different system modes. At all times, however, the cumulative resource requirements of all the tasks and components may never exceed the available resources.

**Definition 4.9** (Mode change latency). *We define  $\mathcal{L} : \Delta \rightarrow \mathcal{T}$ , where  $\mathcal{L}(e)$  is the mode change latency of mode change request  $e$ , representing the length of the time interval between  $Time_e$  and the time when all tasks and components have changed to their target mode  $Mode_e$ , i.e.*

$$\mathcal{L}(e) = mcc(e) - Time_e. \quad (4.13)$$

### Real-time constraints

In this chapter we focus on applications which pose a QoS requirement in terms of a minimum and maximum bound on the time interval between consecutive different outputs generated by the last task in the chain. The application expresses its real-time constraints in terms of relative deadlines  $D_1$  on task  $\tau_1$  and  $D_n$  on task  $\tau_n$ , with  $\hat{E}_1 \leq D_1 \leq T_1$  and  $\hat{E}_n \leq D_n \leq T_n$ . More precisely, we require that the  $k$ 'th instance of tasks  $\tau_1$  and  $\tau_n$  (processing the  $k$ 'th frame) execute within time intervals  $[O_1 + kT_1, O_1 + kT_1 + D_1)$  and  $[O_n + kT_n, O_n + kT_n + D_n)$ , respectively.

In Section 4.5 there is an additional requirement in terms of a maximum bound on the mode change latency, i.e. if  $\mathcal{L}_\Delta$  is the mode change latency bound, then we must guarantee that

$$\forall e \in \Delta : \mathcal{L}(e) \leq \mathcal{L}_\Delta. \quad (4.14)$$

## 4.3 Reducing memory requirements

In this section we analyze the execution of a surveillance system, consisting of a video digitizer at the head, a video renderer at the tail, and a number of data-driven tasks. The data-driven tasks have the role of improving the video frames received from the video digitizer through additional processing. We address the problem of how large the buffers must be and how we must choose the task priorities, in order to meet the real-time constraints.

In Section 4.3.1 we show that, in case of varying execution times, meeting the real-time constraints of the last task in the chain requires a particular priority assignment, and the first and last buffers in the chain to have capacity for  $M$  and  $M + 1$  frames, respectively, with all other buffers having capacity for 1 frame<sup>1</sup>. We also observe that in the above scenario the number of frames in transit never exceeds  $M + 1$ .

In Section 4.3.2 we introduce the concept of a *shared memory pool*, which encapsulates the memory shared between all buffers in an application. We show how a shared memory pool in an application consisting of a chain of  $n$  tasks can save

---

<sup>1</sup>In case of an MPEG-like video encoding, a video-processing task may need to store past frames for encoding or decoding a new frame. In this chapter we ignore the internal memory requirements of each task, and focus on the memory required for communicating frames between tasks.



memory for storing  $M + n - 3$  frames, for  $n \geq 3$ . To be more precise, since at different stages of the task chain frames may have different sizes, we can save memory for storing  $M + n - 3$  *smallest* frames. Memory is managed in terms of fixed-sized blocks, which simplifies the reallocation of memory between buffers, allowing for an efficient implementation of a shared memory pool. We evaluate the memory savings in a real application and show experimental results for a H.264 video encoder.

In Section 4.3.3 we combine shared memory pools with scalable applications and discuss how they affect the memory requirements of applications in different modes, and how they can reduce the mode change latency.

### 4.3.1 Meeting real-time constraints

Because of the equal periods of the head and tail tasks, and the assumption that each instance of a task consumes and produces exactly one frame, there is a constant number of frames in transit within the system. Clearly, if the worst-case execution time for processing one frame is within  $T$  the system will satisfy the real-time constraints with just 1 frame in transit.

However, this strict restriction on the execution time of the complete chain is not realistic. We would like to allow the processing time for individual frames to take longer than  $T$ , as long as shorter processing of other frames will compensate for it.

In the following subsections we will derive the task priorities and buffer capacities which will prevent the tasks from ever blocking on data (the head and tail task may still block on time). The absence of blocking will allow us then to satisfy the real-time constraints introduced in Section 4.2.2.

### Component priorities

When the execution time of a frame is larger than  $T$ , more than one frame will be in transit. Then, the priority assignment becomes important, as observed by Weffers-Albu (2008); Holenderski et al. (2011c).

**Example 4.2.** Consider the system of Figure 4.6 with three tasks where the head task has the lowest priority. If the execution time of a frame exceeds  $D_1 = T$  then the head task will miss the deadline of the next frame as the head task will not be scheduled before a computation has been completed. A similar reasoning holds for the tail task.  $\square$

Example 4.2 suggests that the head and tail tasks should be assigned priorities higher than any of the data-driven tasks in between. With these choices we can regard the system as consisting of three parts: high priority  $\tau_1$ , low priority data-driven chain  $\tau_{2..n-1}$  and high priority  $\tau_n$ . The following example demonstrates that the priority assignment of the tasks in the chain  $\tau_{2..n-1}$  is not arbitrary either.

Let  $\pi_1$  be maximal,  $\pi_2$  be minimal,  $\pi_n = \pi_1 + 1$ , and  $\pi_{n-1} = \pi_2 - 1$ . Since tasks  $\tau_1$  and  $\tau_n$  are time-driven and have priorities higher than any data-driven task in the chain, they can interrupt the execution of the chain at arbitrary places. However, since priorities  $\pi_2$  and  $\pi_{n-1}$  are lower than the priorities of all other tasks in the

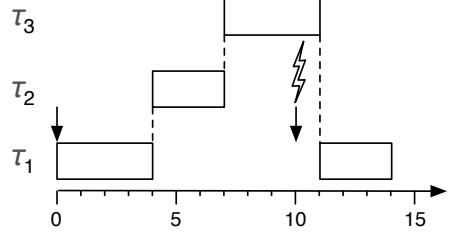


Figure 4.6: Example of the  $\tau_1$  missing its deadline if it is assigned the lowest priority, for  $D_1 = T = 10$ . After the head task has written the frame to the first buffer, the higher priority data-driven tasks preempt  $\tau_1$  which then has to wait until they complete.

chain  $\tau_{2..n-1}$ , tasks  $\tau_1$  and  $\tau_n$  will not affect the execution order of actions in  $\tau_{2..n-1}$ . In the remainder of this section we assume this particular priority assignment.

### Variable execution time

We now define and bound the variable execution time more precisely, which will be relevant later for deriving the necessary buffer capacities.

**Definition 4.10.** We define  $F : \mathbb{N} \rightarrow \mathcal{T}$ , where  $F(k)$  is an upper bound on the execution time needed by the chain  $\tau_{1..n}$  to produce the  $k^{\text{th}}$  frame, after producing the  $k - 1^{\text{th}}$  frame.

We allow  $F(k)$  to vary, but the duration of each size  $M$  window has to be smaller than  $MT$ , i.e.

$$\sum_{i=k-M+1}^k F(i) < MT, k \geq M. \quad (4.15)$$

Here  $M$  is a natural number, which can be regarded as a system parameter. The strict “smaller than” relation indicates that there will always be some idle time in the processing of any sequence of  $M$  frames.

Later in this section we will derive the buffer capacities which are necessary to prevent the tasks from blocking on output. For now, however, let us assume that the buffers are unbounded.

During  $MT$  time units  $\tau_1$  will be activated  $M$  times, inserting  $M$  frames into the chain. Now, consider the system at time 0 when  $\tau_1$  starts and suppose  $\tau_n$  does not execute. Then, according to (4.15), after  $MT$  time units, the last buffer will contain  $M$  frames. We start  $\tau_n$  shortly after  $MT$  and choose  $O_n = MT + D_1$  (with  $\hat{E}_1 \leq D_1 \leq T - \hat{E}_n$ ), making sure that, if we assign  $\tau_n$  a lower priority than  $\tau_1$ , then  $\tau_1$  will not be preempted by  $\tau_n$ . From this point on the number of frames in transit within the chain is either  $M$  or  $M + 1$ , as long as  $\tau_1$  does not block on output and  $\tau_n$  does not block on input.

By the time  $O_n = MT + D_1$ , the chain  $\tau_{1..n-1}$  will have done work equal to

$$\sum_{i=1}^M \vec{E}_{1..n-1}^i. \quad (4.16)$$

During the time interval  $[O_n, O_n + MT)$  the chain will produce  $M$  frames and will execute for

$$\sum_{i=M+1}^{2M} \vec{E}_{1..n-1}^i + \sum_{i=1}^M \tilde{E}_n^i. \quad (4.17)$$

According to (4.15), the chain is guaranteed to reach idle state within at most  $MT$  time units. At that time all of the  $M$  or  $M + 1$  frames in transit will be residing in  $q_{n-1}$ . In general, starting from an idle state, processing a window of size  $M$  preceding the production of frame  $k$  will require

$$\sum_{i=k-2M+1}^{k-M} \vec{E}_{1..n-1}^i + \sum_{i=k-M+1}^k \tilde{E}_n^i. \quad (4.18)$$

Since the head task will simply write the raw frame data to  $q_1$  and the tail task will simply read the data from  $q_2$ , their execution times will not vary and we can use their worst-case execution times and rewrite (4.18) as

$$M\hat{E}_1 + \sum_{i=k-2M+1}^{k-M} \vec{E}_{2..n-1}^i + M\hat{E}_n. \quad (4.19)$$

We can use (4.19) as a lower bound for (4.15), and consequently have

$$M\hat{E}_1 + \sum_{i=k-2M+1}^{k-M} \vec{E}_{2..n-1}^i + M\hat{E}_n \leq \sum_{i=k-M+1}^k F(i) < MT. \quad (4.20)$$

### Capacity of the first and last buffer

The following examples demonstrate how this particular priority assignment affects the required buffer capacities.

**Example 4.3.** Let us assume the above priority assignment and the following scenario. Task  $\tau_1$  writes a frame to  $q_1$ . Subsequently  $\tau_2$  reads it from  $q_1$  and processes it. Let the frame be computationally intensive. Since  $\pi_2$  is minimal, the data-driven chain  $\tau_{2..n-1}$  will process each new frame completely to  $q_{n-1}$  before  $\tau_2$  executes again. According to (4.20) the data-driven chain may be busy with processing the frame for at most  $M(T - \hat{E}_1 - \hat{E}_n)$  time units. During that time the time-driven  $\tau_1$ , which has the highest priority, will have written  $M$  frames to  $q_1$ .  $\square$

**Example 4.4.** Continuing with the last example, let us consider the capacity of the last buffer in the chain. Equation (4.15) implies that after  $MT$  time units the

data-driven chain will process the frames in no time, since the execution time for processing any window of size  $M$  is bounded by  $MT$ . Therefore, the data-driven chain  $\tau_{2..n-1}$  will process the next  $M$  frames before  $\tau_1$  writes another frame into  $q_1$ , essentially purging the first buffer. Since the head and tail task share the same period, these  $M$  frames will accumulate in  $q_{n-1}$ .

Assume that the next frame which enters the chain is very easy to process, i.e. that  $\tau_{2..n-1}$  will push this frame through to  $q_{n-1}$  before  $\tau_n$  gets a chance to remove a frame from  $q_{n-1}$  (i.e. before it completes the call to *ReadRelease()* in Figure 4.5). Since  $q_{n-1}$  had already accumulated  $M$  frames, it will now contain  $M + 1$  frames. However, since tasks  $\tau_1$  and  $\tau_n$  share the same period  $T$ , the following frame will not arrive before  $\tau_n$  had the chance to remove a frame from  $q_{n-1}$ . The last buffer will therefore never exceed  $M + 1$  frames.  $\square$

The previous two examples imply that the buffer capacities should be  $Cap_{q_1} = M$  and  $Cap_{q_{n-1}} = M + 1$ .

### Capacity of the middle buffers

In the previous examples we made no assumption on the capacities of the middle buffers, i.e. buffers  $q_i$  for  $2 \leq i \leq n - 2$ . Now we show that all middle buffers can have capacity 1.

**Lemma 4.1.** *If  $Cap_{q_i} = 1$  for  $2 \leq i \leq n - 2$ , then none of the tasks in  $\tau_{2..n-1}$  will block on output.*

*Proof.* Let us consider the situation at time  $O_n = MT + D_1$ . During the initial  $MT$  time units, task  $\tau_1$  has inserted  $M$  frames into the chain. According to (4.15), by the time  $O_n$  the subchain  $\tau_{1..n-1}$  had enough time to push the  $M$  frames to the last buffer  $q_{n-1}$ . Assume that at time  $MT$ , i.e. before  $\tau_n$  had a chance to remove a frame from  $q_{n-1}$ , a new frame arrives which is very easy to process. The chain  $\tau_{2..n-1}$  will immediately push this frame through to  $q_{n-1}$ , which at this point will reach  $M + 1$  frames. However, since tasks  $\tau_1$  and  $\tau_n$  share the same period  $T$ , the next frame will not arrive before  $\tau_n$  had the chance to remove a frame from  $q_{n-1}$ . Since  $Cap_{q_{n-1}} = M + 1$ , task  $\tau_{n-1}$  cannot become blocked on  $q_{n-1}$ . Moreover, since  $\pi_2$  is minimal,  $\tau_{2..n-1}$  will process each new frame completely to  $q_{n-1}$  before  $\tau_2$  gets a chance to insert the next frame into  $q_2$ . Hence none of the buffers within the chain  $\tau_{2..n-1}$  will ever need to store more than 1 frame, and consequently  $Cap_{q_i} = 1$  for  $2 \leq i \leq n - 2$  is sufficient for none of the tasks in  $\tau_{2..n-1}$  to ever block on output.  $\square$

### Meeting real-time constraints

Let  $\pi_1$  be maximal,  $\pi_2$  be minimal,  $\pi_n = \pi_1 + 1$ , and  $\pi_{n-1} = \pi_2 - 1$ . Since the head and tail tasks are activated periodically and have higher priority than the data-driven tasks, if we show that  $\tau_1$  will never block on output and  $\tau_n$  will never block on input, then we will have shown that the tasks  $\tau_1$  and  $\tau_n$  will meet their real-time constraints.

**Lemma 4.2.**  *$\tau_1$  will never block on output and  $\tau_n$  will never block on input.*

*Proof.* We demonstrate that this blocking of  $\tau_1$  or  $\tau_n$  cannot occur, by showing that  $q_1$  can never be full and  $q_{n-1}$  can never be empty. We show it by contraposition.

Assume that blocking of  $\tau_1$  or  $\tau_n$  on communication does in fact occur and consider the first blocking of either of the two tasks at time  $t$ . Since we have assumed a single processor allowing only a single task executing at a time, the two tasks cannot become blocked at the same time.

This blocking task cannot be  $\tau_1$  since, as long as  $\tau_n$  does not block,  $\tau_{2..n}$  can either process or buffer the frames (given the buffer capacities derived earlier in this section); hence, the contents of  $q_1$  can vary up to  $M$  but will reach 0 within each  $MT$  time interval.

Then the blocking task must be  $\tau_n$ , waiting for a new frame. However, as long as  $q_{n-1}$  contains less than  $M$  frames, the system is not idle as there are frames in transit; because of (4.15) the state with  $q_{n-1}$  containing  $M$  frames recurs within at most  $MT$  time units again and thus can never reach 0. It follows that no such first moment of blocking exists.  $\square$

Now that we have shown that  $\tau_1$  does not block on output and  $\tau_n$  does not block on input, we can state the following theorem.

**Theorem 4.1.** *Given a single application comprised of a task chain in Figure 4.2 with a time-driven head and tail task, executing on a single processor, satisfying (4.15) based on (4.20), and the following settings:*

1.  $\pi_1$  is maximal,  $\pi_2$  is minimal,
2.  $\pi_n = \pi_1 + 1$ ,  $\pi_{n-1} = \pi_2 - 1$ ,
3.  $O_n = MT + D_1$ ,
4.  $Cap_{q_1} = M$ ,  $Cap_{q_{n-1}} = M + 1$ ,

*the real-time constraints of tasks  $\tau_1$  and  $\tau_n$  will be satisfied, for*

1.  $3 \leq n$ ,
2.  $\hat{E}_1 \leq D_1 \leq T - \hat{E}_n$ ,  $\hat{E}_n \leq D_n \leq T$ ,
3.  $\sum_{i=k-M+1}^k F(i) < MT$  for  $k \geq M$ .

Note that since we made no assumptions on the sizes of buffers  $q_i$ ,  $1 < i < n - 1$ , they can all have capacity 1.

In this section we have implicitly assumed a single application in the system. In case there are several applications running side by side, we have to increase the lower bound on deadlines  $D_1$  and  $D_n$ , taking the interference of other applications into account.

### 4.3.2 Reducing memory requirements

From Theorem 4.1 we know that the total capacity of all buffers in an application consisting of a chain of  $3 \leq n$  tasks, as shown in Figure 4.2, is equal to  $M + (n - 3) + (M + 1)$  frames, where  $M$  represents the first buffer,  $(n - 3)$  represents the buffers with capacity 1 between the first and last buffer, and  $(M + 1)$  represents the last buffer. However, as mentioned in Section 4.3.1, the total number of frames in transit never exceeds  $M + 1$  frames.

Rather than allocating each buffer its required capacity, we can have them share a common memory pool, since all buffers together will never require more memory than for storing  $M + 1$  frames. In this way can save the memory for storing  $M + n - 3$  frames, for  $3 \leq n$ .

At different stages of the task chain frames may have different sizes. Let  $s_i$  be the frame requirement for a single frame at stage  $i$ , i.e the size (in terms of blocks) of the largest frame ever stored in the  $i$ 'th buffer in the chain. A chain of  $n$  tasks defines a collection of frame requirements:

$$\underbrace{\{s_1, \dots, s_1\}}_M, \underbrace{\{s_2, s_3, \dots, s_{n-2}\}}_{n-3}, \underbrace{\{s_{n-1}, \dots, s_{n-1}\}}_{M+1}.$$

If we order the frame requirements in the application in ascending order of  $s_i$ , then using a shared memory pool can save the memory space required by the  $M + n - 3$  smallest frame requirements. More precisely, if we arrange the  $M + (n - 3) + (M + 1) = 2M + n - 2$  frame requirements in ascending order in a sequence

$$\langle r_1, r_2, \dots, r_{2M+n-2} \rangle, \quad (4.21)$$

such that

$$\forall i : 1 \leq i \leq 2M + n - 2 : r_i \leq r_{i+1}, \quad (4.22)$$

then the *absolute* memory savings are given by

$$\sum_{i=1}^{M+n-3} r_i, \quad (4.23)$$

and the *relative* memory savings are given by

$$\frac{\sum_{i=1}^{M+n-3} r_i}{\sum_{i=1}^{2M+n-2} r_i}. \quad (4.24)$$

The memory reservations based on fixed-sized blocks simplify the reallocation of memory between buffers, allowing for an efficient implementation of a shared memory pool.

### Experimental results

Figure 4.7 shows an application example of a H.264 video encoder (Wiegand et al., 2003), which is commonly used in the consumer electronics domain. We used it to evaluate the memory savings in a real application.

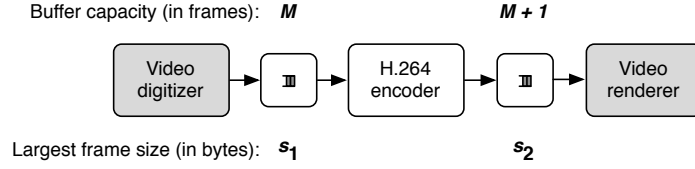
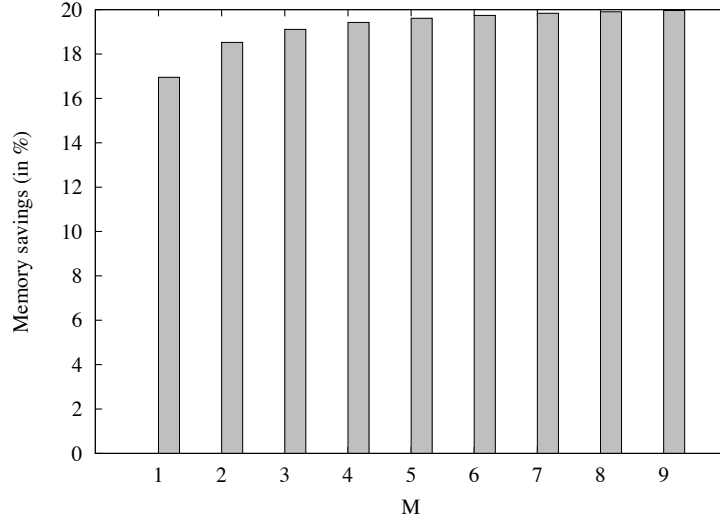


Figure 4.7: A video encoding application.

The application consists of three tasks: the video digitizer provides raw frames in CIF format (with resolution 352x288 pixels) for the H.264 video encoder, which produces a 300kbs video stream with the same resolution for the video renderer. The  $s_1$  parameter is equal to the size of a raw input frame, i.e  $s_1 = 352 \times 288 = 101376$  bytes. We have measured the largest frame ever produced by the H.264 encoder for a series of video sequences<sup>2</sup> to be  $s_2 = 26002$  bytes. By using a shared memory pool, in our chain of  $n = 3$  tasks we can save the memory for storing  $M + n - 3 = M$  of the smaller frames, which in this case are the encoded frames of size  $s_2$ . The relative memory savings are therefore given by

$$\frac{M * s_2}{M * s_1 + (M + 1)s_2}. \quad (4.25)$$

Figure 4.8 shows the relative memory savings of our approach as a function of  $M$ , by filling in the above values for  $s_1$  and  $s_2$  in (4.25).

Figure 4.8: Memory savings in our example application as a function of  $M$ .

<sup>2</sup>Available at <http://media.xiph.org/video/derf/>

Note that video scaling algorithms in Section 4.5 and (Wüst et al., 2005; Jarnikov et al., 2004) can be applied to guarantee that the  $M$  parameter holds, i.e. that the processing of any sequence of  $M$  frames does not exceed  $MT$ .

In our video encoder application the raw frames were 4 times larger than the encoded frames. In general, the smaller the difference between the frame requirements at different stages, the larger the memory savings, as shown in Figure 4.9.

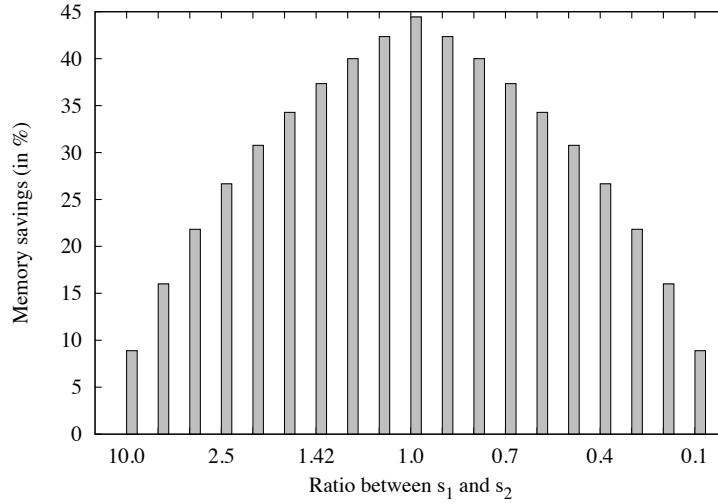


Figure 4.9: Memory savings for different ratios between  $s_1$  and  $s_2$ , assuming  $n = 3$  and  $M = 4$ .

### 4.3.3 Scalable streaming applications

In this section we investigate reallocating memory between applications. Let us continue with the multimedia processing application example shown in Figure 4.2 and consider a system comprised of two such applications. In the new setting both applications are scalable, meaning that when they are provided more resources they can generate higher quality output. More specifically, given more processing time and more memory, each application will generate a higher quality video encoding. Higher quality video will require more memory for storing the buffered frames, thus increasing the  $s_i$  values. Also, varying processing time will result in greater fluctuations of the latency of the processing chain and consequently larger  $M$ , requiring larger buffers along the processing chain.

In Section 4.5 we show how to use in-buffer scaling to reduce the memory requirements of each buffer. In this section we concentrate on how a shared memory pool will affect the memory requirements of a scalable application in different modes.

Equation (4.24) indicates that the relative memory savings due to a shared memory pool increase with increasing  $M$ , as visualized in Figure 4.8.



A higher quality mode (i.e. a mode requiring more resources, in particular more processor time) is likely to exhibit larger variation in execution time of the individual tasks, and hence also of the complete chain, thus increasing  $M$  (provided it is not assigned a larger processor share after the mode change).

Also, (4.23) and (4.24) indicate that the smaller the variation in  $s_i$  the larger the memory savings due to a shared memory pool. Many multimedia streaming applications can be classified as encoders or decoders. An encoder receives a fixed-sized input (e.g. raw video frames from a camera) and encodes it into an output, of which the size depends on the quality settings. Conversely, a decoder will receive a variable sized input and generate a fixed-sized output (e.g. decoded frames rendered to a screen). Given the fixed-sized output or input frames, an application operating in a lower quality mode is likely to exhibit large variation in the frame sizes at different stages of the chain.

In summary, scalable applications are likely to exhibit larger relative memory savings due to a shared memory pool when they execute in higher modes.

Note that if the memory reservations underlying the memory pools of different applications are managed in terms of fixed-sized blocks of the same size, then the reallocation of memory between memory pools belonging to different applications will be simpler and more efficient. Memory blocks can be simply added and removed from a list of available blocks, compared to a solution based on finding the best fit between memory requirements and available blocks. Moreover, rather than scaling the memory reservations for all buffers one by one, a shared memory pool allows to scale the memory requirements of the complete chain in a single step<sup>3</sup>. Thus a shared memory pool, next to reducing the memory requirements of a scalable application, can also reduce the mode change latency.

## 4.4 Handling overload conditions

In this section we generalize the application model to a directed acyclic graph, with time-driven boundary components and all other components being data-driven. Hentschel et al. (2003) present a study of a multimedia processing application, shown in Figure 4.1. They observe that fluctuation in the processing time for decoding different video frames may fill up the buffers along the video processing subchain. The buffers may continue to fill up until the demultiplexer component becomes blocked, when it is not able to write both the audio and the video frames into its two output buffers. This is an example of how the decoding of the audio stream may become blocked due to insufficient buffer space along the video processing path. In the end this leads to the dropping of audio frames, manifested by sound artifacts.

In this section we will focus on the audio and video subchains following the demultiplexer. The demultiplexer reads multiplexed frames from its input buffer and writes the audio part to its audio output buffer  $b_a$  and the video part to its video output buffer  $b_v$ . If either of these queues is full, then the demultiplexer blocks.

---

<sup>3</sup>The data residing in the buffers still needs to be managed for each buffer individually.

Let  $t$  be the time when  $b_v$  becomes full. If we do not address the congestion in the video subchain, then the demultiplexer will block and consequently stop feeding  $b_a$ , eventually leading to audio artifacts.

One option is to extend  $b_v$  with additional memory, to prevent the demultiplexer from blocking. However, since we assumed a memory constrained platform, we cannot simply allocate more memory to one application without other applications suffering. We therefore propose to free up some space for the demultiplexer by dropping some of the frames in  $b_v$ . Note that dropping frames from any other buffer along the video subchain will not alleviate the problem, since the demultiplexer will still be blocked on its output buffer.

We cannot simply pop the head element from the video output buffer. Since a single video frame is allowed to span across several buffer elements, the head element could belong to a video frame which is currently being processed by the video decoder.

Waiting until the decoder has finished processing the current frame, and then discarding the head frame by simply popping it, is not an option either, since this is the very delay we want to avoid.

Instead, we propose to selectively drop those frames from  $b_v$  which can no longer be processed in time. Since we have assumed a bounded end-to-end latency, and adjusted the buffer capacities accordingly, an overload condition means that a deadline was already missed. We want to prevent the deadline miss to propagate to other video *and* audio frames, leading to artifacts in the audio stream. We therefore propose to scan the video output buffer for the beginning of the next frame and drop all those elements which belong to that frame. A mechanism for dropping arbitrary frames also makes it possible to scan the buffer for the least significant video frame (e.g. a B or P frame in case of MPEG encoding Isovich and Fohler (2004)) and to drop that one instead, thus preserving the perceived quality of the output video as much as possible.

For this to work we extend the buffer component interface in Section 4.2.2 with additional methods providing limited access to arbitrary buffer elements:

*Element* ***GetNextElement***(*Buffer*  $q$ ) allows to iterate through the buffer elements in buffer  $q$ . Every following invocation returns the following buffer element, or a *NULL* pointer if the end of the queue was reached.

*void* ***ResetNextElement***(*Buffer*  $q$ ) resets the pointer returned by the next call to *GetNextElement*( $q$ ) to the head of the queue  $q$ .

*void* ***Drop***(*Buffer*  $q$ , *Element*  $e$ ) deletes the buffer element pointed to by  $e$  from the queue  $q$ .

We need to iterate through the buffer elements (using the methods *GetNextElement*() and *ResetNextElement*()) and drop those belonging to the least significant frames (using the method *Drop*()). This congestion control should be triggered by the demultiplexer whenever it notices a full output buffer. It should be executed by a

component which is aware of the semantics of the frames in the congested buffer.

Only the access to the shared data structures (such as the administration data structures for buffers and budgets) is synchronized, according to the Stack Resource Policy Baker (1991). Having the highest priority component do all the scaling and dropping of buffer elements guarantees that no other component will interfere.

## 4.5 Bounding the mode change latency

In Section 4.4 we have dealt with overload conditions. In this section we investigate reallocating memory between applications. We describe a scalable multimedia application, which can operate in one of several modes (see Section 4.2.2), and show how to reduce the mode change latency. We outline our implementation of the proposed method in the  $\mu\text{C}/\text{OS-II}$  real-time operating system, and present simulation results which validate our approach.

A mode change latency in a synchronous mode change protocol is composed of two parts: the waiting time between a mode change request and the start of a mode change (Section 4.5.1), and duration of the mode change itself (Section 4.5.2).

### 4.5.1 Bounding the waiting time before a mode change

Upon a mode change request the system needs to know *when* the old mode tasks (or components) are ready to release their resources (i.e. when they have completed) before reallocating these resources to the new mode tasks (or components), and it must deal with task and component interdependencies, in case they access shared resources. Our solution based on FPDS offers a simple implementation of mode changes, from the perspective of component design and system integration, while improving on the existing mode change latency bound.

The system may decide to change the mode of a task or component at an arbitrary moment during the execution of a segment. We assume that the computations performed by the tasks are scalable, such that the execution of a task can be terminated at segment boundaries.

Upon a mode change request, tasks need to execute mode change routines responsible for adapting the resource requirements of the components they use. *We let the QM task  $\tau_{qm}$  (released upon a mode change request and responsible for managing the mode change) execute the component mode change routines on behalf of the components and assign  $\tau_{qm}$  the highest priority<sup>4</sup>.* After the currently running segment completes,  $\tau_{qm}$  performs the mode changes for all components. Since  $\tau_{qm}$  has the highest priority other tasks will not be able to interfere with the mode change, thus reducing the mode change latency.

Our approach is especially suited for applications composed of scalable tasks or components which produce incremental results, e.g. a scalable MPEG decoder, which incrementally processes enhancement layers to produce higher quality results. After

---

<sup>4</sup>In case of FPPS with non-preemptive resource access protocol, the priority of  $\tau_{qm}$  must be lower than the priority assigned by the resource access protocol to a task executing a critical section.

each layer the decoding can be aborted, or continued to improve the result. We assume a scalable MPEG decoder (Haskell et al., 1996; Jarnikov, 2007), with a layered input stream. Every frame in the stream has a *base layer*, and a number of additional *enhancement layers*, which can be decoded to increase the quality of the decoded video stream.

### Quality Manager

We add an additional task to the system, the Quality Manager (QM) task  $\tau_{qm}$ , which is responsible for coordinating the mode changes. There is one QM per application, which is responsible for distributing the resources between the tasks and components comprising the application. In a system comprised of several applications, there may be a global QM, which distributes the resources between the applications.  $\tau_{qm}$  is released upon a mode change request and has the highest priority among all tasks belonging to the same application.

The QM is aware of the possible modes of all *scalable tasks and components* in  $\Gamma^S$  and  $\mathcal{C}^S$  (e.g. buffer components). We are not concerned with how the QM selects system modes, and for the sake of simplicity we assume that the set of components is fixed and that the mode selection is table driven, based on a *system mode table* pre-computed offline<sup>5</sup>.

Upon a mode change request, the QM determines the necessary task and component mode changes by computing the “difference” between the target system mode and the current system mode in the system mode table.

Each scalable component implements the following methods, which are called by the QM upon a mode change:

***BeforeModeChange(targetMode)*** The argument *targetMode* refers to the component’s mode after the mode change. This method allows the component to do any pre-processing before a mode change. For example, if the memory requirements of a buffer in *targetMode* are smaller than in the current mode, then the buffer gets a chance to gracefully scale down its requirements (e.g. by discarding some of its elements or shrinking the data stored inside the elements) and to decrease its memory reservation by calling *DiscardMemoryReservation()*.

***AfterModeChange(oldMode)*** The argument *oldMode* refers to the component’s mode before the mode change. This method allows the component to do any post-processing after a mode change. For example, if the memory requirements of a buffer in the current mode are larger than in the *oldMode*, then the buffer gets a chance to increase its memory reservation by calling *RequestMemoryReservation()*.

---

<sup>5</sup>Our mode change approach is applicable also to open systems, where tasks and components may enter and leave the system during runtime.

### The Swift Mode Change protocol

A mode change request is represented by the arrival of the QM task  $\tau_{qm}$ , which performs the following steps, illustrated in Figure 4.10:

1. Identify the tasks and components involved in the mode change, by comparing the current application mode and the target application mode.
2. Wait for the currently active segments which are involved in the mode change or are using components involved in the mode change to complete. The protocol for this step depends on the scheduling algorithm:

FPPS: In order to prevent tasks *not* involved in the mode change from interfering, QM raises the priority of the tasks involved in the mode change higher than those which are not (for the duration of their current segment). This guarantees that these segments will complete without being preempted by other tasks. As soon as they complete,  $\tau_{qm}$  is resumed and proceeds with the mode change.

FPDS: When  $\tau_{qm}$  is started, the fact that we are using FPDS guarantees that all other active tasks are remaining at a segment boundary<sup>6</sup>. Hence the FPDS scheduler takes care of this step automatically, by having the currently running segment complete before starting  $\tau_{qm}$ .

3. Call *BeforeModeChange()* in all components involved in the mode change.
4. Adapt the global data structure keeping track of the current system mode.

Note that the task mode basically controls the execution path through a task. Each task is responsible to check its mode before any mode dependent code is executed. Therefore, no additional work is required to change the task modes, besides adapting the data structure keeping track of the current system mode.

5. Call *AfterModeChange()* in all components involved in the mode change.

Notice that we presented here a synchronous mode change protocol without periodicity, where the current segments of tasks involved in the mode change are forced to complete first (by raising their priority), possibly interfering with the unchanged components *not* involved in the mode change. However, this interference is bounded by the mode change latency and thus it can be taken into account in the schedulability analysis as a blocking term for all tasks, including the unchanged ones.

### Analysis

We now present the analysis for system mode change latency under FPPS and FPDS.

**Definition 4.11.** We define  $\acute{E} : \mathcal{C} \rightarrow \mathcal{T}$ , where  $\acute{E}(c)$  is the sum of the worst case pre- and post-processing times of component  $c$ .

---

<sup>6</sup>A segment boundary corresponds to a preemption point in the literature on FPDS.

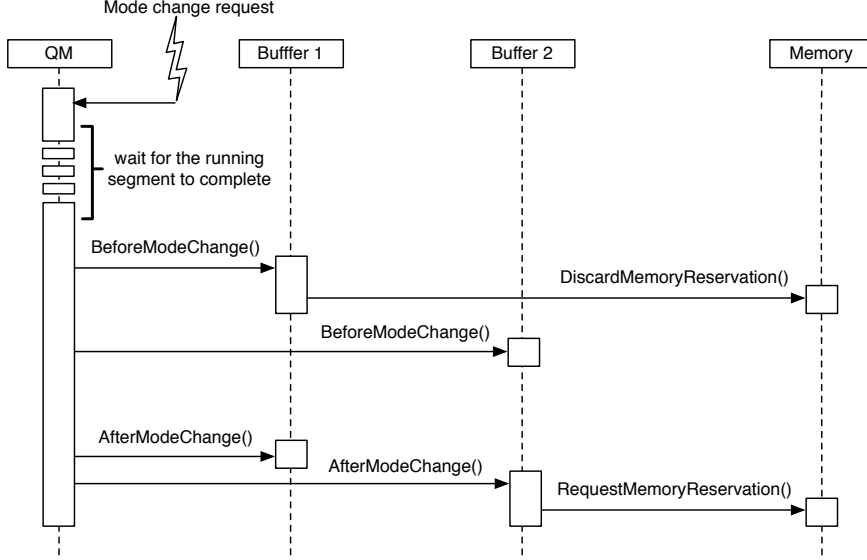


Figure 4.10: A message sequence diagram illustrating the Swift Mode Change protocol for a mode change request requiring the Buffer 1 component to reduce its memory requirements and Buffer 2 to increase its memory requirements.

**Definition 4.12.** We define  $\phi^\Gamma : \mathcal{M} \times \mathcal{M} \rightarrow 2^{\Gamma^S}$ , where  $\phi^\Gamma(x, y)$  is the set of tasks directly involved in the mode change from system mode  $x$  to  $y$ , i.e.

$$\phi^\Gamma(x, y) = \{\tau_i \in \Gamma^S \mid \tau_i^x \neq \tau_i^y\} \quad (4.26)$$

**Definition 4.13.** We define  $\phi^C : \mathcal{M} \times \mathcal{M} \rightarrow 2^{C^S}$ , where  $\phi^C(x, y)$  is the set of components directly involved in the mode change from system mode  $x$  to  $y$ , i.e.

$$\phi^C(x, y) = \{c \in C^S \mid c^x \neq c^y\} \quad (4.27)$$

The system mode change latency is given by

$$\mathcal{L}(x, y) = \mathcal{L}_W(x, y) + \sum_{c \in \phi^C(x, y)} \dot{E}(c) + E_{sys} \quad (4.28)$$

where

- $\mathcal{L}(x, y)$  is the latency of a system mode change from system mode  $x$  to  $y$ .
- $\mathcal{L}_W(x, y)$  is the worst-case time interval between a mode change request and the time when all tasks accessing components in  $\phi^C(x, y)$  have reached a segment boundary. It corresponds to step 2 in the mode change protocol.  $\mathcal{L}_W(x, y)$  depends on the scheduling algorithm and is discussed in the following two sections.

- $\sum_{c \in \phi^c(x,y)} \dot{E}(c)$  is the time required by all components directly involved in the mode change to do any pre- and post-processing, corresponding to steps 3 and 5 in the mode change protocol.
- $E_{sys}$  is the time needed to identify the tasks and components involved in the mode change and to adapt the mode tables, corresponding to steps 1 and 4 in the mode change protocol. Since this overhead is relatively small compared to the other terms, we represent it with a constant worst-case overhead.

**Fixed-priority preemptive scheduling** In a preemptive system, a segment may be preempted by a higher priority segment. This higher priority segment may again be preempted by a segment with an even higher priority. Therefore, due to arbitrary preemption, several segments may be preempted and active at the same time. In the worst case we can have a chain of segments which are preempted just after they started executing.

Since tasks may share components, the tasks involved in a mode change may block on the unchanged tasks which are not involved in the mode change. For example, in Figure 4.11, if a mode change involves only the output buffer of the  $\tau_d$  component, then the decoder task (involved in the mode change) may block on the network task (not involved in the mode change), since they communicate over a shared buffer component.

**Definition 4.14.** We define  $\Phi(x, y) : \mathcal{M} \times \mathcal{M} \rightarrow 2^\Gamma$ , where  $\Phi(x, y)$  is the set of tasks directly and indirectly involved in the mode change from system mode  $x$  to  $y$ . It includes a transitive closure of tasks which require components involved in the mode change, i.e.

$$\Phi(x, y) = \phi^\Gamma(x, y) \cup f(\phi^c(x, y))$$

where  $f(\phi^c(x, y))$  is the set of tasks indirectly involved in the mode change, i.e.

$$\begin{aligned} f : 2^c &\rightarrow 2^\Gamma \text{ and } f(C) = g(C) \cup f(g(h(C))) \\ g : 2^\Gamma &\rightarrow 2^c \text{ and } g(T) = \bigcup_{\tau_i \in T} \lambda(\tau_i) \\ h : 2^c &\rightarrow 2^\Gamma \text{ and } h(C) = \bigcup_{c \in C} \gamma(c) \end{aligned}$$

The set union gets rid of duplicates due to tasks accessing several components, and components serving several tasks.

**Lemma 4.3.** Given a set of scalable tasks  $\Gamma^S$  and a mode change from system mode  $x$  to  $y$ , we can divide  $\Gamma^S$  into two disjoint subsets:  $\Gamma_c^S = \Phi(x, y)$  and  $\Gamma_u^S = \Gamma^S \setminus \Phi(x, y)$ , representing the set of tasks involved in the mode change (directly and indirectly), and the set of unchanged tasks, respectively. For FPPS with a non-preemptive resource access protocol<sup>7</sup>, the maximum blocking time  $B_{\Phi(x,y)}$  experienced by components in

<sup>7</sup> A “non-preemptive resource access protocol” refers to a synchronization protocol where segments requiring non-preemptive resources are executed non-preemptively, e.g. by temporarily raising their priority higher than any other priority in the task set.

$\Gamma_c^S$  due to components in  $\Gamma_u^S$  is bounded by the duration of the longest critical section<sup>8</sup> among tasks in  $\Gamma_u^S$ .

*Proof.* We prove Lemma 4.3 by contradiction. Let  $b_{max} \in \mathcal{S}$  be the longest critical section among tasks in  $\Gamma_u^S$  and let  $\tau \in \Gamma_c^S$  arrive just after a critical section  $b_i \in \mathcal{S}$  in one of the tasks in  $\Gamma_u^S$  has started. The non-preemptive resource access protocol guarantees that at most one critical section can be entered at a time (assuming no nested critical sections). If the blocking time  $B_{\Phi(x,y)}$  is larger than  $b_{max}$  then there must be a sequence of at least two critical sections  $b_i$  and  $b_j$  among the tasks in  $\Gamma_u^S$  which execute without preemption by  $\tau$ . Assuming the mode change protocol raises the priorities of the tasks in  $\Gamma_c^S$  higher than any task in  $\Gamma_u^S$  upon a mode change request,  $\tau$  is guaranteed to have a priority higher than the tasks containing  $b_i$  and  $b_j$ . Therefore, after  $b_i$  is finished  $b_j$  will not be able to start before  $\tau$  completes.  $\square$

Since a segment  $s$  has to complete before allowing the components providing resources in  $R_s$  to change their mode, for FPPS the  $\mathcal{L}_W(x, y)$  term in (4.28) is bounded by the *sum* of durations of *longest* segments of tasks accessing the components involved in the mode change *plus* the blocking time due to the unchanged components, i.e.

$$\mathcal{L}_W^{FPPS}(x, y) = \sum_{\tau_i \in \Phi(x, y)} \left( \max_{\tau_{i,j} \in S_i} E_{i,j} \right) + B_{\Phi(x, y)} \quad (4.29)$$

Combining (4.28) and (4.29) we get

$$\mathcal{L}^{FPPS}(x, y) = \sum_{\tau_i \in \Phi(x, y)} \left( \max_{\tau_{i,j} \in S_i} E_{i,j} \right) + B_{\Phi(x, y)} + \sum_{c \in \phi^C(x, y)} \dot{E}(c) + E_{sys} \quad (4.30)$$

**Fixed-priority with deferred preemption scheduling** When using FPDS with non-preemptive segments, a component may be preempted only at segment boundaries. This implies that at most one segment may be active at a time (the currently running one); all other components will be waiting at one of their segment boundaries. We therefore can avoid waiting until all the components currently accessing those components involved in the mode change have completed, and wait only until the currently running component reaches its next segment boundary.

Therefore, for FPDS the  $\mathcal{L}_W(x, y)$  term in (4.28) is bounded by the *duration of longest segment* among *all* components in the system (since all segments are non-preemptive, including those not involved in the mode change):

$$\mathcal{L}_W^{FPDS}(x, y) = \max_{\tau_i \in \Gamma^S} \left( \max_{\tau_{i,j} \in S_i} E_{i,j} \right) \quad (4.31)$$

Note that by considering the complete set of scalable tasks  $\Gamma^S$  we take into account the blocking due to unchanged components which share components with the components involved in the mode change.

---

<sup>8</sup>A critical section is a segment requiring at least one non-preemptive resource.



Combining (4.28) and (4.31) we get

$$\mathcal{L}^{FPDS}(x, y) = \max_{\tau_i \in \Gamma^S} \left( \max_{\tau_{i,j} \in S_i} E_{i,j} \right) + \sum_{c \in \phi^C(x, y)} \dot{E}(c) + E_{sys} \quad (4.32)$$

**Intermezzo** Until now we have assumed that under FPDS segments are non-preemptive. We can relax this assumption and distinguish between two classes of components: *application components* and *framework components*. An application component is non-preemptive with respect to other application components, but it can be interrupted by a framework component *as long as* they do not share common non-preemptive resources. From the perspective of the interrupted component the framework components behave similarly to interrupt service routines, resuming the preempted component upon completion. If we implement the QM component  $\tau_{qm}$  as a framework component we can reduce the mode change latency by considering only those components which are involved in the mode change, rather than considering all components in the system, since a segment not sharing any resources with components involved in the mode change can be preempted by a framework component. The  $\mathcal{L}_W(x, y)$  term in (4.28) therefore becomes

$$\mathcal{L}_W^{FPDS}(x, y) = \max_{\tau_i \in \Phi(x, y)} \left( \max_{\tau_{i,j} \in S_i} E_{i,j} \right) \quad (4.33)$$

Combining (4.28) and (4.33) we get

$$\mathcal{L}^{FPDS}(x, y) = \max_{\tau_i \in \Phi(x, y)} \left( \max_{\tau_{i,j} \in S_i} E_{i,j} \right) + \sum_{c \in \phi^C(x, y)} \dot{E}(c) + E_{sys} \quad (4.34)$$

**Improving the bound for synchronous mode change protocols, without periodicity** As indicated in Section 4.1.4, the currently best known bound on the mode change latency in a synchronous mode change protocol without periodicity, due to (Real, 2000), is equal to

$$\mathcal{L} = \sum_{\tau_i \in \Gamma_{old} \cup \Gamma_{unch}} E_i \quad (4.35)$$

where  $\Gamma_{old}$  is the set of tasks in the old mode and  $\Gamma_{unch}$  is the set of unchanged tasks. Their algorithm behaves similarly to the one we described for the FPPS case, in Section 4.5.1. We can apply our results for FPDS to the processor mode change domain and reduce this mode change latency bound.

If we make the components  $\tau_i \in \Gamma_{old} \cup \Gamma_{unch}$  non-preemptive, then at most one component will be running at a time, and the mode change latency will be reduced to

$$\mathcal{L} = \max_{\tau_i \in \Gamma_{old} \cup \Gamma_{unch}} E_i \quad (4.36)$$

Of course, the lower bound comes at a price of a tradeoff between the latency bound and the schedulability of the component set, where the non-preemptive components may increase the blocking time of higher priority components.

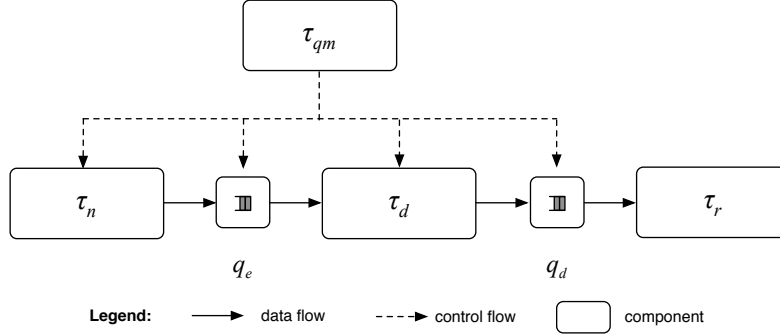


Figure 4.11: An example of a scalable multimedia application.

On the one hand, it is likely that a component will be required to complete after a mode change request, in order to avoid the corruption of shared resources. Therefore the increase in blocking time may not be significant. On the other hand, there are more efficient resource access protocols than Fixed-Priority Non-preemptive Scheduling (FPNS), with lower bounds on the maximum blocking time. For example FPNS can be considered as the most pessimistic configuration of FPDS: if components can be subdivided into shorter segments, then the mode change latency can be reduced, while *at the same time* improving the schedulability of the component set.

### Simulation results

We simulated the complete system from the perspective of a single application, and represented the resource contention between applications by varying the target modes in mode changes.

The simulation setup is shown in Figure 4.11. It is an instance of an application comprised of a chain of components described in Section 4.2. It consists of three communicating tasks. The network task  $\tau_n$  receives incoming frames via the network and writes them to the buffer  $q_e$  containing encoded frames. The decoder task  $\tau_d$  reads the encoded frames from  $q_e$ , decodes them, and writes the result to  $q_d$  containing decoded frames. The renderer task  $\tau_r$  reads the decoded frames from  $q_d$  and renders them. The rendering hardware buffers the last rendered frame, and in case a new frame has not arrived in time, the buffered frame is rendered again. For simplicity we assume that each incoming video frame fits within a single buffer element. The buffer components reside in a shared memory  $mem$  (which is large enough to accommodate both buffers), and all tasks require a shared processor  $p$  to execute. The execution of tasks  $\tau_n$ ,  $\tau_d$ , and  $\tau_r$  is comprised of three phases: initialize the task, execute the body of the task, and finalize the task. The component and task parameters are summarized in Table 4.1, with the parameters in bold depending on the system mode.

A mode change request was generated periodically with period  $T_{qm} = 1.05T$ . Running the simulation for  $20T$  ensured that mode changes were requested at differ-

$c$	$\pi_c$	$O_c$	$T_c$	$D_c$	$R_c$	$P_c$
$q_e$	5	0	$\infty$	$\infty$	$\{(mem, \mathbf{N}_e)\}$	$\{(b_e, \mathbf{N}_e)\}$
$q_d$	6	0	$\infty$	$\infty$	$\{(mem, \mathbf{N}_d)\}$	$\{(b_d, \mathbf{N}_d)\}$

$\tau_i$	$\pi_i$	$O_i$	$T_i$	$D_i$	$S_i$
$\tau_{qm}$	1	0	52.5 ms	52.5 ms	$\langle(1, \{p, b_e, b_d\})\rangle$
$\tau_n$	2	0	50 ms	50 ms	$\langle(\varepsilon, \{p\}), (E_n, \{p, b_e\}), (\varepsilon, \{p\})\rangle$
$\tau_r$	3	1	50 ms	50 ms	$\langle(\varepsilon, \{p\}), (E_r, \{p, b_d\}), (\varepsilon, \{p\})\rangle$
$\tau_d$	4	2	50 ms	50 ms	$\langle(\varepsilon, \{p\}), (E_d, \{p, b_e, (b_d, \mathbf{n}_d)\}), (\varepsilon, \{p\})\rangle$

Table 4.1: Summary of the component and task parameters.

ent points during the execution of components. Every mode change involved  $\tau_d$ ,  $q_e$  and  $q_d$ . The mode changes would alternate between the high and low modes defined as follows:

**low mode:**  $q_e$  and  $q_d$  have 2 buffer elements ( $N_e = N_d = 2$ ), and each video frame produced by the  $\tau_d$  requires a single buffer element ( $n_d = 1$ ).

**high mode:**  $q_e$  has 2 buffer elements ( $N_e = 2$ ),  $q_d$  has 4 buffer elements ( $N_d = 4$ ), and each video frame produced by the  $\tau_d$  requires 2 buffer elements ( $n_d = 2$ ).

The setup was embedded in the real-time operating system  $\mu\text{C}/\text{OS-II}$  (Labrosse, 1998), which was extended with support for periodic components, FPDS, and the architecture described in this chapter (the RM, the QM and the corresponding component interfaces). The RM used the memory partitions provided by  $\mu\text{C}/\text{OS-II}$  to provide guaranteed memory allocations. We simulated the computation of task  $\tau_i$  by means of busy-waiting. To get consistent results we ran the simulation within the cycle accurate OpenRISC simulator inside Linux (Ubuntu 8.10).

Following the observation that FPNS is the most pessimistic configuration for FPDS, we had each task consist of a single segment, and compared the mode change latency under FPPS vs. FPDS by scheduling the segments preemptively vs. non-preemptively. In case of FPPS, since the synchronization of access to buffers was limited to updating their administration data, the blocking time due to components sharing the two buffers was insignificant compared to the component execution times.

In Figures 4.12 and 4.13 the period  $T$  was set to 50ms and the total utilization of all three components was fixed at 0.6. We varied  $E_d$  and distributed the remaining execution time equally between  $E_n$  and  $E_r$ . The “V” shape of the FPDS results is due to the mode change latency being dominated by  $\tau_n$  and  $\tau_r$  for lower values of  $E_d$ .

Since there is at most a single frame in each buffer, the contribution of the  $\sum_{c \in \phi^c(x,y)} \bar{E}(c)$  term in (4.28), representing the scaling down of frames inside the *BeforeModeChange()* and *AfterModeChange()* methods, is relatively small. Hence the measured mode change latency shown in the figure is dominated by the  $\mathcal{L}_W(x, y)$  term.

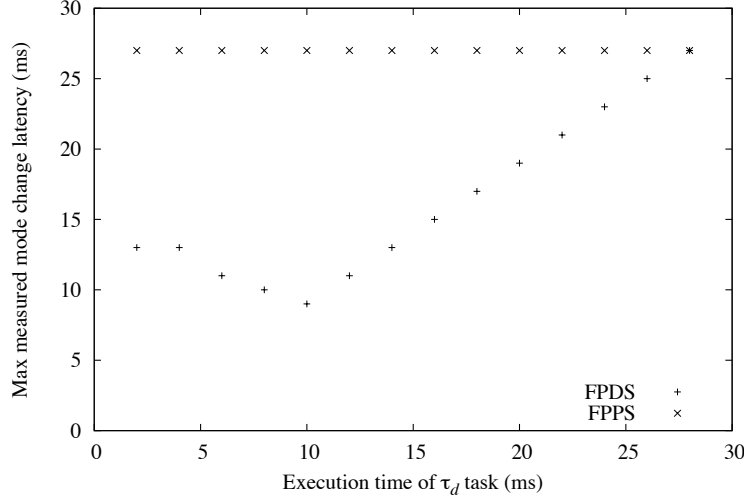


Figure 4.12: Execution time  $E_d$  vs. maximum measured mode change latency.  $E_n = E_r = 30ms - E_d/2$

The results in Figures 4.12 and 4.13 validate the analysis in Section 4.5.1 and illustrate the improvement of FPDS over FPPS in terms of the worst-case and average-case system mode change latency.

#### 4.5.2 Bounding the mode change duration

In the previous section we have focused on reducing the waiting time between a mode change request and the start time of the mode change. In this section we focus on reducing the duration of the mode change itself.

A mode change will affect each component in one of three ways with respect to resource provisions: reduce, increase or keep them the same. The interesting case is when the target mode allocates fewer resources than the current mode and the component has to adapt its resource requirements by giving up some of its resources. Reducing the mode of a buffer component (implemented by *BeforeModeChange()*) can be divided into two steps: scaling down its frames followed by reducing its budget size. There are several approaches to scaling down the frames.

According to Section 4.3.1, the end-to-end latency of the complete task chain is bounded by  $MT$ . If we reduce the application mode, the  $M$  parameter is likely to change as well, since processing frames across the chain is likely to take less time. Let  $M_c$  and  $M_t$  be the  $M$  parameters in the current and target modes, respectively.

We continue by describing two different approaches to scaling down the frames throughout the chain. The first approach drops all buffered frames upon a mode change request, while the second approach applies the method presented above and

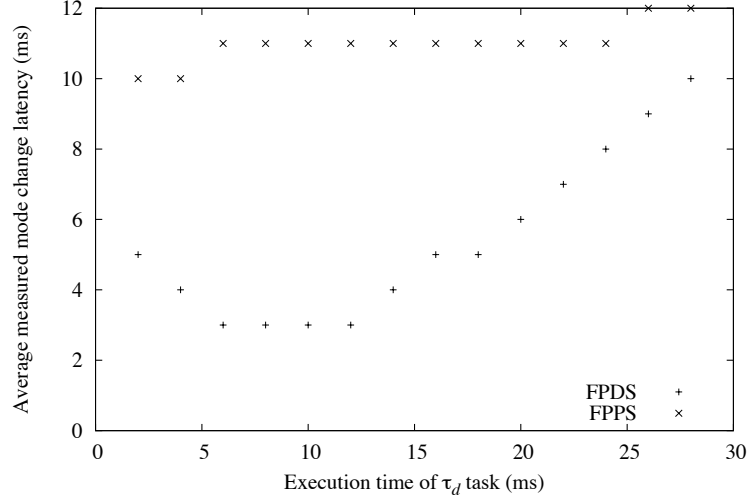


Figure 4.13: Execution time  $E_d$  vs. average measured mode change latency.  $E_n = E_r = 30ms - E_d/2$

drops only selected frames. For each approach we will analyze its mode change latency bound from the application and the system perspective (referred to as the *application latency* and the *system latency*).

#### Approach 1: drop all buffered frames

Instead of waiting for the renderer to read all the frames from the last buffer, we can:

1. Immediately drop all frames from all buffers.
2. Suspend  $\tau_r$  and wait until the pipe is filled again before resuming it.

The first step avoids the delay in the previous approach, however, dropping frames means losing the work invested in processing the dropped frames. Note that  $M_c + 1$  frames will be dropped.

The second step again suffers a delay of  $(M_t + 1)T$  time units. Moreover, due to interdependencies between video frames (e.g. in MPEG a B frame depends on a P or an I frame), several of the first frames which arrive in step 2 may be useless, thus increasing the delay even further.

#### Approach 2: drop only selected frames

In our approach we scale the two buffers  $q_e$  and  $q_d$  differently. We reduce the mode of  $q_e$  by dropping only selected frames, and reduce the mode of  $q_d$  by reducing the resolution of the frames already in the buffer.

1. Drop selected frames from  $q_e$  (and optionally  $q_d$ ), with preference for the least significant ones.
2. Reduce the quality of the remaining frames in both buffers.

We can accomplish the first step using the methods *GetNextElement()*, *ResetNextElement()* and *Drop()*, as described in Section 4.4. Note that we need to drop exactly  $M_c - M_t$  frames. We start with dropping selected frames from  $q_e$  (in case of MPEG the B and P frames)<sup>9</sup>. However, if there are not enough frames in  $q_e$  to drop, then we proceed with dropping selected frames from  $q_d$ , until we have dropped  $M_c - M_t$  frames.

The algorithm for selecting the desired frames to drop strongly depends on the application and the video encoding. In the worst case, selectively dropping the desired frames will require linearly scanning through  $M_c + 1$  frames in all the buffers. If iterating through a single frame and optionally dropping it costs  $E_{drop}$ , then the worst-case total cost for selectively dropping the desired frames will be  $(M_c + 1)E_{drop}$  time units<sup>10</sup>.

	System latency	Application latency
Approach 1	0	$(M_t + 1)T$
Approach 2	$(M_c + 1)(E_{drop} + E_{scale})$	$(M_c + 1)(E_{drop} + E_{scale})$

Table 4.2: Impact of different scaling approaches on the mode change latency with respect to the complete system and the application itself.

In the second step, the mechanism for reducing the quality of frames is likely to be different for different buffers. For example, in  $q_e$  we can drop the enhancement layers (Jarnikov, 2007), while in  $q_d$  we can reduce the resolution. Note that dropping enhancement layers assumes a particular video encoding and in general may not always be possible.

We can accomplish the second step using the standard read and write interface of a buffer described in Section 4.2.2. Let  $n_d$  be the number of frames in  $q_d$  upon the mode change request. We cycle through all the  $n_d$  frames by popping the head frame from the  $q_d$ , reducing its resolution and pushing it back on  $q_d$ , until we have processed all the  $n_d$  frames. We can do the same for the frames in  $q_e$ <sup>11</sup>. In the worst case we will need to scale  $M_c + 1$  frames in all the buffers. If  $E_{scale}$  is the worst-case time required to scale a frame in any of the buffers<sup>12</sup>, then the second step will take  $(M_c + 1)E_{scale}$  time units.

Both buffers will be able to reduce their budgets, allowing the system to reallocate the reclaimed memory, after  $(M_c + 1)(E_{drop} + E_{scale})$  time units.

<sup>9</sup>We assume here temporal video scaling. See (Jarnikov, 2007) for other scaling methods, such as spatial or SNR scaling.

<sup>10</sup>In the best case, an optimal algorithm would require  $nE_{drop}$  time for dropping  $n$  frames.

<sup>11</sup>If the enhancement layers are stored in separate buffer elements, then we can optimize this step by using the buffer interface for selective dropping.

<sup>12</sup>Note that dropping enhancement layers is likely to take less time than reducing the resolution of a frame, hence  $E_{scale}$  is pessimistic.

From the application perspective, since the scaling is executed at the highest priority, the renderer may suffer a delay of  $(M_c + 1)(E_{drop} + E_{scale})$  time units.

Note that since we drop exactly  $M_c - M_t$  frames we end up with  $M_t$  frames in the chain, hence we do not need to suspend  $\tau_r$  to fill the pipeline (as it is done in the first approach). Also note, that if  $M_c = M_t$ , then we can skip step 1 altogether, reducing the system and application latency to  $(M_c + 1)E_{scale}$ .

Table 4.2 summarizes the impact of the different approaches on the system and the application latency. Note that both latencies define a time interval starting with the arrival of the mode change request.

In practice, iterating through a frame and scaling down the resolution of a frame are simple operations (relative to decoding) and so  $E_{drop} + E_{scale}$  is very likely to be smaller than the frame period  $T$ .

### Simulation results

The simulation setup was similar to the one described in Figure 4.11 in the previous section. We ran the simulation with the following parameters:

- period of the network component  $\tau_n$  and the renderer component  $\tau_r$ ,  $T = 50ms$ ,
- execution time of the network component  $E_{\tau_n} = 2ms$ ,
- execution time of the decoder component  $E_{\tau_d} = 3ms$ ,
- execution time of the renderer  $E_{\tau_r} = 3ms$ ,
- worst-case time needed to scale down a single frame  $E_{scale} = 2ms$ ,
- period of the QM component  $T_{qm} = 410ms$ ,
- $M_c = M_t = M$ ,
- two application modes: high and low,
- each encoded frame in  $q_e$  occupies one buffer element,
- each decoded frame in  $q_d$  occupies one buffer element in the low mode and two buffer elements in the high mode.

The QM component would periodically request a mode change alternating between the high and low modes. We varied the execution time of the complete component chain by executing the decoder component for  $E_{\tau_d}$  most of the time, and once in  $MT$  having it fill the remaining capacity by executing it for an additional  $MT - M(E_{\tau_n} + E_{\tau_d} + E_{\tau_r})$ .

The particular component parameters were selected based on data collected in a study of video frame sizes in MPEG encoding by Fitzek and Reisslein (2001)<sup>13</sup>. The processing time of the decoder was assumed to be proportional to the frame size. We

<sup>13</sup>Their collected data is available at <http://www.tkn.tu-berlin.de/research/trace/trace.html>

determined the ratio between the minimum and maximum execution times for the decoder by averaging over several data sets.

The simulations were run for 3300ms. The first invocation of the renderer component was offset with  $T(M + 0.5)$  ms to fill the pipeline (according to Weffers-Albu (2008)), resulting in around 60 frames for approach 2.

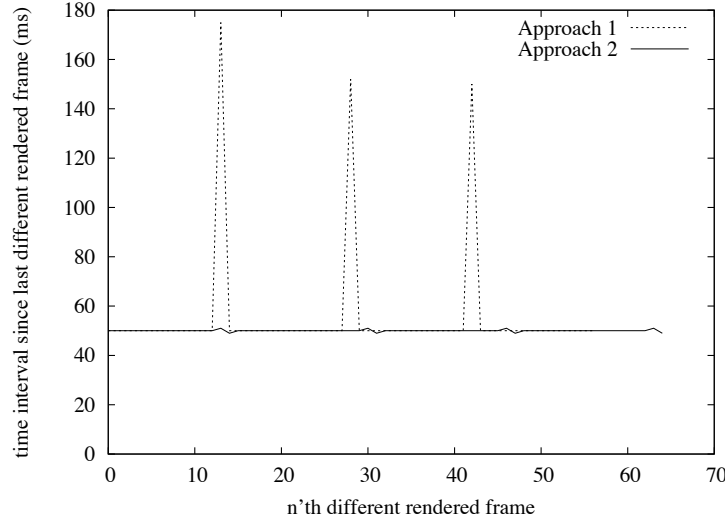


Figure 4.14: Simulation results, for  $T = 50ms$ ,  $C_n = 2ms$ ,  $C_d = 3ms$ ,  $C_r = 3ms$ ,  $E_{scale} = 2ms$ ,  $T_q = 410ms$ , and  $M = 2$

Figure 4.14 compares approaches 1 and 2, illustrating their application latency. The graph visualizes the interruptions in the rendered frames. A peak means that during one or more consecutive iterations of the renderer the  $q_e$  buffer was empty, forcing the renderer to repeat the last rendered frame. Therefore a peak means a larger interval between consecutive *different* rendered frames, possibly leading to video artifacts. Note that during the time interval represented by a peak no new frames are rendered, meaning that fewer different frames are rendered, explaining why the peaks for approach 2 are slightly shifted.

Figure 4.14 confirms the behavior claimed in Section 4.5 for  $M = 2$ : the large peaks of about 150 ms for approach 1 correspond to the waiting time of  $(M_t + 1)T$ . The small peaks for approach 2 correspond to the overhead for scaling the frames residing in  $q_d$ .

Figure 4.15 shows the simulation results for  $M = 8$ . As expected, the behavior is very similar, with larger application latencies for approach 1 due to the larger  $M_t$  resulting in a longer waiting time for filling the pipeline.

The lack of a bump in Figure 4.15 may be explained by most frames residing in the encoded buffer when a mode change occurred, due to  $MT = T_{qm}$ . Since the overhead for dropping frames is negligible it does not show in the figure.



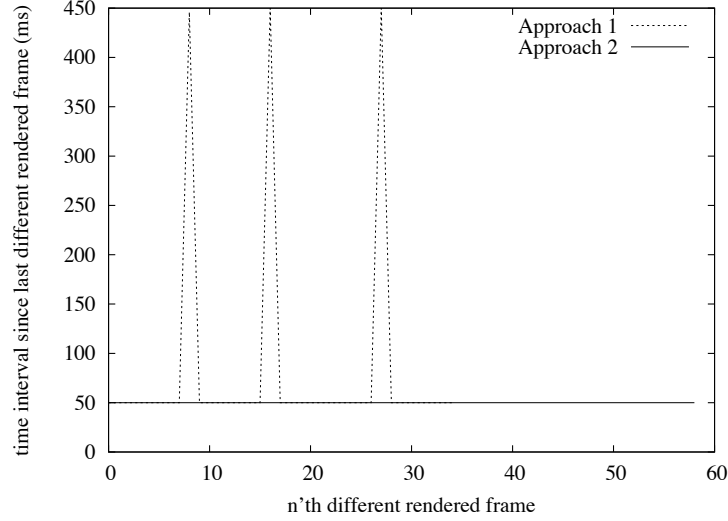


Figure 4.15: Simulation results, for  $T = 50ms$ ,  $C_n = 2ms$ ,  $C_d = 3ms$ ,  $C_r = 3ms$ ,  $E_{scale} = 2ms$ ,  $T_q = 410ms$ , and  $M = 8$

## 4.6 Discussion

In Section 4.3 we have shown a general mechanism for reducing memory requirements in a streaming application comprised of a chain of components with periodic head and tail components communicating via shared buffers. The proposed method is based on having the buffers share a common memory pool. We have shown that the total capacity of all buffers in an application consisting of a chain of  $3 \leq N$  components is equal to  $M + (N - 3) + (M + 1)$  frames. We exploited the fact that in the above scenario the total number of frames in transit never exceeds  $M + 1$ , and proposed to share a memory pool with capacity for  $M + 1$  frames between all the buffers. As a result, in an application consisting of a chain of  $N$  components, we can save memory for storing  $M + N - 3$  frames. To be more precise, since at different stages of the component chain frames may have different sizes, we can save memory for storing  $M + N - 3$  *smallest* frames.

Managing the memory in terms of fixed-sized blocks simplifies the reallocation of memory between buffers, allowing for an efficient implementation of a shared memory pool. The results for an H.264 encoder show memory savings of around 19%. If applied to scalable applications, a shared memory pool will result in greater relative memory savings for applications operating in higher modes.

In Section 4.4 we have shown how additional access to the buffer, in particular the support for dropping arbitrary frames, can be used to guarantee the Quality of Service of the application during overload conditions.

In Section 4.5 we have investigated mode changes in scalable applications. We

compared two approaches, based on FPPS and FPDS, in the presence of resource sharing. We showed that combining scalable components with FPDS (exploiting its non-preemptive property and coinciding early termination points with preemption points), guarantees a shorter worst case latency than FPPS, and applied these results to improve the existing latency bound for mode changes in the processor domain. By adopting the presented architecture, a mode change in a pipelined application can be performed almost instantaneously, without the need to clear the whole pipeline, further reducing the mode change latency. The presented mode change protocol is simple and avoids the complication and overheads associated with task signaling during a mode change, necessary in existing mode change protocols.

We have also shown how a combination of two in-buffer scaling approaches: dropping selected frames in the buffer, and reducing the resolution of frames in the buffer by iterating through all its elements, can guarantee a low mode change latency from the system and application perspective. We have validated our analysis with simulation results.

## Chapter 5

# Multi-resource management

Modern computers consist of several processing units, connected via one or more interconnects to a memory hierarchy, auxiliary processors and other devices. A simple approach to sharing such platforms between several applications treats the machine as a single resource: the task having access to the processor has also implicitly access to all other resources, such as a bus, memory, or network. Consequently, only a single task is allowed to execute at a time. On the one hand, this approach avoids the complexity of fine-grained scheduling of multiple resources. On the other hand, it prevents tasks with independent resource requirements to execute concurrently and thus use the available resources more efficiently. For example, a video processing task requiring the processor and operating on the processor's local memory can execute concurrently with a DMA transfer task moving data between the global memory and the network interface. In this chapter we assume that a task represents workload which does not necessarily require a processor.

With the advent of multiprocessor platforms new scheduling algorithms have been devised aiming at exploiting some of the available concurrency. However, they are again limited to tasks which execute on one processor at a time. In this chapter we address the problem of scheduling tasks on a multiprocessor platform, where a task can execute on several processors at the same time. Moreover, a task may also specify requirements for other heterogeneous resources, such as a bus, digital signal processor, shared memory variable, etc. In this respect our problem is related to parallel-task scheduling on multiple resources, where a task may execute on several processors at the same time.

Parallel-task scheduling was originally investigated in the context of large main-frame computers without real-time constraints (Ousterhout, 1982). When threads belonging to the same task execute on multiple processors and communicate via shared memory, then it is often desirable to schedule these threads at the same time (called *gang scheduling* (Ousterhout, 1982)), in order to avoid invalidating the shared memory (e.g. L2 cache) by switching in threads of other tasks. Also, simultaneous scheduling of threads which interact frequently will prevent the otherwise sequential execution due to synchronization points and the large context switching overheads

(Ousterhout, 1982). Parallel-task scheduling is especially desired in data intensive applications (e.g. multimedia processing), where multithreaded tasks operate on the same data (e.g. a video frame), performing different functions at the same rate (Anderson and Calandrino, 2006). To the best of our knowledge, existing literature on preemptive parallel-task scheduling with real-time constraints has only considered independent tasks. In this chapter we present a fixed-priority preemptive multi-resource scheduling algorithm for parallel tasks with real-time constraints.

Table 5.1 lists the concurrent scheduling problems found in real-time literature and expresses them in terms of our model in Chapter 2, in particular in terms of constraints on the resource and the application models. It uses the following notation:

- $\{(p, k) \mid k = 1\}$  represents a set containing single-unit resources,
- $\{(p, 1)\}$  represents a singleton set containing one single-unit resource.
- $\{(p, k)\} \mid k \geq 1$  represents a singleton set containing one multi-unit resource,

Name	$\mathcal{P}$	$R_{i,j}$
Partitioned multi-processor scheduling	$\{(p, k) \mid k = 1\}$	$\{(p, 1)\}$
Global multi-processor scheduling	$\{(p, k)\} \mid k \geq 1$	$\{(p, 1)\}$
Partitioned parallel-task scheduling	$\{(p, k) \mid k = 1\}$	$\{(p, k) \mid k = 1\}$
Global parallel-task scheduling	$\{(p, k)\} \mid k \geq 1$	$\{(p, k)\} \mid k \geq 1$

Table 5.1: Classification of different concurrent scheduling problems

Note that scheduling is part of the mapping, which is a relation between the application model and the resource model. By restricting the resource and application models in Table 5.1, we are introducing constraints on the mapping.

In *partitioned* scheduling each segment is allocated to a particular *subset* of processors, unlike *global* scheduling where a task specifies only a *number* of (homogeneous) processors it requires, which the system allocates during runtime. While in *multi-processor* scheduling a segment represents an execution requirement for *one* of several processors, in *parallel-task* scheduling a segment represents a requirement for simultaneous execution on *several* processors.

## Problem description

Current multiprocessor synchronization protocols only consider tasks which execute on a single processor at a time. They are not suitable for synchronizing parallel tasks, which may execute on several processors at a time and share resources. A simple approach to scheduling such tasks on a platform comprised of multiple heterogeneous resources is to “collapse” all the processors into one virtual processor and use uniprocessor scheduling. However, this will result in at most one task executing at a time. Our goal in this chapter is to provide a scheduling algorithm for partitioned parallel tasks with real-time constraints which can exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources.

## Contributions

In this chapter we present a new partitioned parallel-task scheduling algorithm called Parallel-SRP (PSRP), which generalizes MSRP (Gai et al., 2001) for multiprocessors, and the corresponding schedulability analysis for the problem of multi-resource scheduling of parallel tasks with real-time constraints. We show that the algorithm is deadlock-free, derive a maximum bound on blocking, and use this bound as a basis for a schedulability test. We present an example which demonstrates the benefits of PSRP.

## Publications

The PSRP algorithm and its schedulability analysis was presented in (Holenderski et al., 2012c).

## 5.1 Related work

To the best of our knowledge our work is the first to consider parallel-task scheduling on multiple resources with real-time constraints. In this section we discuss the related literature from the domains of multiprocessor scheduling with shared resources and multiprocessor scheduling of parallel tasks. See (Davis and Burns, 2011) for a thorough survey of hard real-time scheduling for multiprocessor systems.

### 5.1.1 Multiprocessor scheduling of independent tasks

Most of the existing literature on multiprocessor scheduling can be classified in two categories: *partitioned* scheduling, where tasks are assigned to a particular processor, and *global* scheduling, where tasks or jobs can migrate between processors. Calandrinio et al. (2007); Easwaran et al. (2009); Lipari and Bini (2010) present a clustering approach, which generalizes partitioned and global scheduling. Given a platform with  $m$  processors, they define a cluster as a set of  $m'$  processors, where  $1 \leq m' \leq m$ . Tasks are partitioned into clusters and globally scheduled within the clusters. A *physical* cluster of size  $m'$  is statically allocated to a set of  $m'$  processors, while a *virtual* cluster is allocated to processors dynamically such that at any time at most  $m'$  tasks within a cluster are executing.

### 5.1.2 Multiprocessor scheduling of dependent tasks

When tasks share non-preemptive resources, they may block when the resource they are trying to access is already locked by another task. Such conflicts may be resolved offline by means of a table-driven schedule, or during runtime by means of synchronization protocols. In this thesis we focus on runtime mechanisms, in particular fixed-priority scheduling.

In priority-based scheduling blocking may lead to *priority inversion* when a higher priority task is blocked on a resource locked by a lower priority task. In real-time

systems it is important to bound the duration of priority inversion. Sha et al. (1990) investigate uniprocessor systems where tasks share several single-unit non-preemptive resources and propose the Priority Inheritance Protocol (PIP) for bounding priority inversion, and the Priority Ceiling Protocol (PCP), which also avoids deadlock. The Stack Resource Policy (SRP) by Baker (1991) avoids deadlock, bounds the priority inversion to a single critical section and allows all tasks to share a single stack in systems scheduled according to fixed or dynamic priority and where tasks share multi-unit non-preemptive resources.

In multiprocessor scheduling of dependent tasks each task requires one of the available processors and may share additional logical resources with other tasks. Scheduling with such dependencies requires multiprocessor synchronization protocols.

Dijkstra (1964, 1982) presents one of the earlier synchronization protocols for multiprocessors, called the Banker's algorithm. The algorithm focuses on avoiding deadlock when several concurrently executing tasks share a common multi-unit non-preemptive resource, but it does not provide any real-time guarantees. Tasks do not have priorities nor timing constraints (besides terminating in a finite amount of time), and each task is assumed to execute on its own processor. They may acquire and release the units of the shared resource in any order, as long as the total number of claimed units does not exceed a specified maximum, and as long as they release all claimed units upon completion. Habermann (1969) presents a generalization of the Banker's algorithm to *several* nonpreemptive multi-unit resources.

More recently, existing real-time synchronization protocols for uniprocessors have been extended to multiprocessors. They focus on synchronizing access to global resources, which are resources accessed from tasks executing on different processors. There are two main approaches for handling global resources: when a task wants to access a global resource which is already locked by another task executing on a different processor, the task may be (i) suspended, or (ii) it may execute a spin-lock. When a spin-lock is used, a task blocking on a locked global resource continues to spin on the resource, preventing lower priority tasks from executing. In the suspension approach, when a task blocks on a global resource it is suspended, enabling lower priority tasks to execute. The key benefit of the suspension approach is that the idle time during suspension is available for execution by other tasks. The disadvantages, however, are the penalties of back-to-back executions and multiple priority inversions per task.

Rajkumar et al. (1988) takes the suspension based approach and presents the Multiprocessor Priority Ceiling Protocol (MPCP) and Distributed Priority Ceiling Protocol (DPCP) (for distributed memory systems). Gai et al. (2001) takes the spin-lock approach and presents the Multiprocessor Stack Resource Policy (MSRP). All three protocols assume partitioned EDF scheduling. The Flexible Multiprocessor Locking Protocol (FMLP) by Block et al. (2007) can be regarded as a combination of MPCP and MSRP and can be applied to both partitioned and global EDF multiprocessor scheduling. Brandenburg and Anderson (2008b) extend FMLP to partitioned static-priority scheduling (FMLP P-SP).

MPCP, DPCP, MSRP and FMLP assign ceilings to locks in order to defer requests which otherwise could be granted. Deferring the requests allows to reduce and bound

the blocking due to priority inversion.

The authors in Gai et al. (2003); Brandenburg et al. (2008); Brandenburg and Anderson (2008a) investigate the performance penalties between various spin-lock and suspension based protocols (MPCP, DPCP, MSRP and FMLP) and conclude that spin-lock based approaches incur smaller scheduling penalty than suspension based, especially for short critical sections and when access to local resources dominates access to global resource. The authors in Lakshmanan et al. (2009) extend the original suspension-based MPCP description with spin locking, compare the two implementations and show the opposite, i.e. that for low preemption overheads and long critical sections the suspension-based approaches perform better, while in other settings they perform similarly.

Block et al. (2007) claim that FMLP outperforms MSRP, by showing that FMLP can schedule more task sets than MSRP. They assume freedom in partitioning the task set, i.e. that tasks may be assigned to arbitrary processors, and exploit this assumption to schedule task sets which are not schedulable under MSRP. Arbitrary partitioning, however, may not necessarily hold for heterogeneous systems, where different processors may provide different functionality. Our PSRP algorithm is spin-lock based. The advantage of choosing this approach is simpler design and analysis, compared to a suspension based approach. We based our algorithm on MSRP, as it suits our model better, in the sense that given the particular resource requirements of tasks we cannot exploit the advantages of FMLP.

The description of MSRP in Gai et al. (2001, 2003) does not address multi-unit resources, which were supported by the original SRP description for a uniprocessor Baker (1991). Our PSRP algorithm supports multi-unit nonpreemptive resources.

Nested critical sections can lead to deadlock. MSRP and MPCP explicitly forbid nested global critical sections. FMLP supports nested critical sections, by means of *resource groups*. Two resources belong to the same resource group iff there exists a task which requests both resources at the same time. Before a task can lock a resource  $r$  it must first acquire the lock to the corresponding resource group  $G(r)$ . This ensures that only a single task can access the resources in a group at any given time. On the other hand, the resource groups in effect introduce large super-resources, which can be accessed by at most one task at a time, thus limiting concurrency in the system. In this thesis we model tasks as sequences of parallel segments which require concurrent access to a set of resources. Nested global critical sections are addressed by a task segment requiring simultaneous access to all of its required resources (similar to the approach proposed in Havender (1968) for the general problem of deadlock avoidance in multitasking), without the need for locking entire resource groups. It can be visualized as stretching the inner critical sections outward, until they overlap with the outermost one. This stretching is safe, in the sense that it preserves the integrity of the resources guarded by inner critical sections.

Notice that the schedulability analysis for PSRP resembles the holistic scheduling analysis presented by Tindell and Clark (1994). They describe the end-to-end delay for a pipeline of tasks in a distributed system, where each task is bound to a processor and can trigger a task on another processor by sending a message via a shared network. Their tasks correspond to our segments, and their pipelines of

tasks correspond to our tasks. However, in their model each task executes on a single processor and may require only local nonpreemptive resources. Their model was extended in García et al. (2000) to include tasks which can synchronize on and generate multiple events. They allow tasks to execute concurrently on different nodes, but do not enforce parallel execution, while in this thesis we assume parallel provision of all resources required by a task segment.

### 5.1.3 Multiprocessor scheduling of parallel tasks

While under multiprocessor scheduling of sequential tasks each task executes on exactly one processor at a time, under parallel task scheduling a task needs to execute on several preemptive resources (e.g. processors) or nonpreemptive resources (e.g. graphical processing units) simultaneously.

A well-known method for addressing the parallel task scheduling problem is called *gang scheduling*. It was first introduced in Ousterhout (1982), and later discussed among others in Feitelson (1990); Feitelson and Rudolph (1992). In its original formulation it was intended for scheduling concurrent jobs on large multiprocessor systems without real-time constraints.

The work on parallel task scheduling with real-time constraints dates back to Li and Malek (1988), where the authors extend Amdahl's law Amdahl (1967) to include communication delay. They estimate the lower and upper bounds on speedup in a multiprocessor system and propose a method for estimating the response time of parallel tasks which incorporates the communication delays. They assume a uniform distribution of workload between the processors. In contrast, in this thesis we target systems with arbitrary workload distribution.

More recently, Anderson and Calandrino (2006) present a “spread-cognizant” scheduler (SCS), which is a homogenous multi-processor scheduling algorithm for independent tasks which encourages individual threads of a multi-threaded task to be scheduled together. They observe that when such threads are cooperative and share a common working set, this method enables more effective use of on-chip shared caches resulting from fewer cache misses. They consider a multi-core architecture with symmetric single-threaded cores and a shared L2 cache. They employ the global PD<sup>2</sup> and EDF schedulers. Notice that their algorithm only “encourages” individual threads of a multithreaded task to be scheduled together, unlike gang scheduling, which guarantees that these threads will be scheduled together. Also, the threads belonging to the same task may have different execution times, but a common period.

Kato and Ishikawa (2009) present a preemptive EDF gang scheduling algorithm on homogenous multiprocessors (Gang EDF), together with the corresponding schedulability analysis. They assume that a task  $\tau_i$  requires a subset of  $m_i$  homogeneous processors, and that tasks are fully preemptive and independent.

Goossens and Bertin (2010) present an exact schedulability test for several variants of preemptive fixed-priority gang scheduling on homogenous multiprocessors (Gang FPS), including Parallelism Monotonic, Idling, Limited Gang, and Limited Slack Reclaiming. They assume that a task  $\tau_i$  requires a subset of  $m_i$  homogeneous processors, and that tasks are fully preemptive and independent.



The authors of (Lakshmanan et al., 2010) adopt the *basic fork-join* model. A task starts executing in a single master thread until it encounters a *fork* construct. At that moment it spawns multiple threads which execute in parallel. A *join* construct synchronizes the parallel threads. Only the master thread can fork and join. A task can therefore be modeled as an alternating sequence of single- and multi-threaded subtasks. They assume that tasks are fully preemptive and independent.

Saifullah et al. (2011) address the problem of scheduling independent periodic parallel tasks with implicit deadlines on multi-core processors. They propose a new task decomposition method that decomposes each parallel task into a set of sequential tasks and prove that their task decomposition achieves a resource augmentation bound when the decomposed tasks are scheduled using global EDF and partitioned deadline monotonic scheduling, respectively. They do not consider shared resources.

#### 5.1.4 Summary of the related work

Figure 5.2 summarizes the related work discussed earlier in this section using the terminology of our system model.

## 5.2 Recap of related synchronization protocols

In this section we first recapitulate the definitions of various kinds of blocking which can result from sharing non-preemptive resources. We then recapitulate several synchronization protocols which aim at reducing the blocking. In particular, we describe the Banker’s algorithm, the Stack Resource Policy (SRP), and the MSRP protocol, which inspired our PSRP algorithm described later in this chapter.

### 5.2.1 Blocking due to shared non-preemptive resources

We can summarize the different terms related to blocking in uniprocessor systems (Buttazzo, 2004):

**Blocking** or **priority inversion** occurs when a higher priority task waits for the execution of a lower priority task.

**Direct blocking** occurs when a higher priority task tries to acquire a resource already held by a lower priority task. Direct blocking is necessary to ensure mutually-exclusive access to a shared resource, in order to maintain its consistency.

**Transitive blocking** occurs when a higher priority task is directly blocked by a middle priority task, which itself is directly blocked by the lower priority task.

**Push-through blocking** or **avoidance blocking** occurs when a medium priority task is blocked by a lower priority task that has inherited a higher priority from a task it directly or transitively blocks. Push-through blocking is necessary to avoid “unbounded” priority inversion.

	Global vs. Partitioned	Priorities	$\mathcal{P}$	$\mathcal{N}$	$R_{i,j}$	$ S_i $	Nested global critical sections
Banker's algorithm	P, G	F, D	$\{(p, k) \mid k = 1\}^3$	$\{(n, k) \mid k \geq 1\}$	$\{(p, 1)\}$ or $\{(p, 1), (n, k)\} \mid k \geq 1$	$N^5$	yes
Habermann (1969)	P, G	F, D	$\{(p, k) \mid k = 1\}^3$	$\{(n, k) \mid k \geq 1\}$	$\{(p, 1)\}$ or $\{(p, 1), (n, k)\} \mid k \geq 1$	$N^5$	yes
SRP	$_{-1}$	F, D	$\{(p, 1)\}$	$\{(n, k) \mid k \geq 1\}$	$\{(p, 1)\}$ or $\{(p, 1), (n, k)\} \mid k \geq 1$	$N$	-
MPCP	P	F	$\{(p, k) \mid k = 1\}$	$\{(n, k) \mid k = 1\}$	$\{(p, 1)\}$ or $\{(p, 1), (n, 1)\}$	$N$	no
MSRP	P	F, D	$\{(p, k) \mid k = 1\}$	$\{(n, k) \mid k = 1\}$	$\{(p, 1)\}$ or $\{(p, 1), (n, 1)\}$	$N$	no
FMLP	P, G	D	$\{(p, k) \mid k = 1\}$	$\{(n, k) \mid k = 1\}$	$\{(p, 1)\} \cup \{(n, k) \mid k = 1\}$	$N$	yes <sup>4</sup>
FMLP P-SP	P, G	F, D	$\{(p, k) \mid k = 1\}$	$\{(n, k) \mid k = 1\}$	$\{(p, 1)\} \cup \{(n, k) \mid k = 1\}$	$N$	yes <sup>4</sup>
SCS	G	D	$\{(p, k) \mid k \geq 1\}$	$\emptyset$	$\{(p, k) \mid k \geq 1\}^2$	$N^6$	-
Gang EDF	G	D	$\{(p, k) \mid k \geq 1\}$	$\emptyset$	$\{(p, k) \mid k \geq 1\}$	1	-
Gang FPS	G	F	$\{(p, k) \mid k \geq 1\}$	$\emptyset$	$\{(p, k) \mid k \geq 1\}$	1	-
PSRP	P	F	$\{(p, k) \mid k = 1\}$	$\{(n, k) \mid k \geq 1\}$	$\{(p, k) \mid k = 1\} \cup \{(n, k) \mid k \geq 1\}$	$N$	yes <sup>4</sup>

- P Partitioned scheduling  
G Global scheduling  
F Fixed priority  
D Dynamic priority (EDF)  
1 It does not make sense to talk about the global or partitioned scheduling, since there is a single resource.  
2 Segment items do not have to be provided simultaneously (they are “encouraged” to be scheduled together).  
3 Each component is executing on its own processor.  
4 Group lock for nested global critical sections.  
5 Segments may be nested (rather than only sequential).  
6 All segments have the same length (the scheduling quantum), and all segments belonging to the same components have a requirement for the same number of units of the multi-unit processor.  
- Not applicable.

Table 5.2: Summary of the related work

**Chained blocking** occurs when a task experiences a sequence of two or more direct or transitive blockings.

**Deadlock** occurs when two or more tasks are waiting indefinitely for an event that will never happen. A deadlock can occur when resources are acquired in nested fashion and there is a cyclic dependency (Hansen, 1973, p. 123).

Any priority-driven system will suffer priority inversion if tasks are allowed to share non-preemptive resources. The synchronization protocols presented in the remainder of this section all avoid deadlock and chained blocking, at the cost of introducing push-through blocking.

### 5.2.2 Banker's algorithm

Dijkstra (1964, 1982) presents the Banker's algorithm, which avoids deadlock in a system where several (concurrent) tasks share a common multi-unit resource. In its original formulation the shared resource is money which is lent out by a banker to several clients, who need the money to complete their project. Since all clients can operate in parallel, we can consider them as tasks executing on a multiprocessor platform, consisting of as many processors as there are tasks. Each client can borrow one or more florins<sup>1</sup>, which are then added to his loan, and may return one or more florins, which are then subtracted from his loan. Each client specifies the maximum loan he will ever require at a time and is obliged to return any outstanding loan upon completion of his project, which he is assumed to complete in finite amount of time. The money provided by the banker can be regarded as a non-preemptive multi-unit resource, since the money lent out to a client cannot be claimed back by the banker until the client returns it, and each florin can be regarded as a unit of the multi-unit resource. The algorithm is invoked whenever a client requests a loan. It checks if granting the client the loan will never lead to a deadlock, given the outstanding claims and loans of other clients, i.e. if it can be guaranteed that all (present and future) requests can be granted within a finite amount of time if clients current request is granted. In case a potential deadlock is detected, the request is denied, otherwise it is granted.

The algorithm maintains two variables for each client  $i$ : the current  $loan(i)$  and the outstanding  $claim(i)$ . Let  $need(i)$  be the maximum loan which client  $i$  will ever require. At all times for each client it holds:

$$loan(i) + claim(i) = need(i).$$

Let  $capital$  be the banker's initial capital and  $cash$  be the remaining money, given the outstanding loans. At any time it holds:

$$cash = capital - \sum_i loan(i).$$

---

<sup>1</sup>Old Dutch currency.

The banker may not lend out more money than the initial capital, i.e.

$$cash \geq 0.$$

The banker maintains a list of clients in ascending order according to their outstanding claims, hence

$$claim(i) \leq claim(i + 1).$$

To guarantee absence of deadlock, the banker simulates actually granting the request and checks if the resulting state is safe. The algorithm for determining if the resulting state is safe is outlined in Algorithm 1. It returns *true* if the state is safe and *false* otherwise.

---

**Algorithm 1** The Banker's Algorithm

---

```

available := cash;
i := 1;
while claim(i) ≤ available do
    available := available + loan(i);
    if available < capital then
        i := i + 1;
    else
        return true;
    end if
end while
return false;

```

---

While the Banker's algorithm prevents deadlock, it does not bound the response time of tasks. It only assumes that each task has a finite execution time, and consequently it can only guarantee that each task will complete within a finite amount of time.

### 5.2.3 Stack Resource Policy (SRP)

The Stack Resource Policy (SRP) was introduced by Baker (1991). It was intended as a priority inversion protocol for accessing shared logical multi-unit resources on a uniprocessor system. The protocol prevents deadlock, chained blocking, allows to share a single stack and has a *very* straightforward implementation.

The seminal paper on SRP considered a uniprocessor system. In order to support fixed-priority scheduling as well as Earliest Deadline First (EDF), Baker (1991) equipped each task  $\tau_i$  with a *preemption level*. Unlike the priority of a task in EDF, the preemption level is fixed during runtime. SRP also assigns each resource  $r$  a *resource ceiling*  $\varphi(r)$  which is equal to the highest preemption level among all tasks accessing the resource. During runtime, the *system ceiling*  $\Pi$  is equal to the highest resource ceiling among the currently accessed resources, or  $\pi_{\perp}$  if no resources are accessed. In this chapter we consider fixed-priority scheduling, so we ignore the preemption levels in SRP, which are needed for EDF scheduling, and use tasks' priorities instead.

The original SRP scheduling rule says:

**Definition 5.1** (SRP scheduling rule). *A task is not allowed to start executing until its priority is the highest among the ready tasks and its preemption level is higher than the system ceiling.*

Algorithm 2 outlines a uniprocessor fixed-priority scheduler supporting SRP based synchronization.

---

**Algorithm 2** Single processor FPPS scheduler based on SRP

---

```

 $\tau_i := \text{highest priority task in the ready\_queue}$ 
if  $\pi_i < \pi_{\text{running}} \wedge \pi_i < \Pi$  then
     $\tau_{\text{running}} := \tau_i$ 
end if

```

---

#### 5.2.4 Multiprocessor Stack Resource Policy (MSRP)

In this section we summarize the Multiprocessor Stack Resource Policy (MSRP) by Gai et al. (2001), which forms the basis for the PSRP algorithm proposed in this chapter.

MSRP is an extension of SRP to multiprocessors. Gai et al. (2001) assume partitioned multiprocessor scheduling, meaning that each task is statically allocated to a processor. Depending on this allocation, they distinguish between local and global resources: *local resources* are accessed by tasks assigned to the same processor, while *global resources* are accessed by tasks assigned to different processors. Similarly to SRP, the original MSRP relies on preemption levels to allow sharing of resources under EDF scheduling. However, by substituting task priorities for preemption levels, MSRP can be easily applied to fixed-priority systems.

The MSRP protocol is defined by the following five rules:

1. For local resources, the algorithm is the same as the SRP algorithm. In particular, for every local resource  $r$  we define a resource ceiling  $\varphi(r)$  equal to the highest priority among the tasks using the resource, and for every processor  $p$  we define a system ceiling  $\Pi(t, p)$  which at any moment  $t$  is equal to the highest resource ceiling among all resources locked by the tasks on  $p$ , or  $\pi_{\perp}$  if no resources are accessed. At time  $t$ , a task  $\tau_i$  is allowed to preempt a task already executing on  $p$  only if its priority  $\pi_i$  is higher than  $\Pi(t, p)$ .
2. Tasks are allowed to access local resources through nested critical sections. It is possible to nest local and global resources. However, it is not possible to nest global critical sections.
3. For each global resource  $r$ , every processor  $p$  defines a resource ceiling  $\varphi_p(r)$  greater than or equal to the highest priority of the tasks on  $p$ .

4. When at time  $t$  a task  $\tau_i$  allocated to processor  $p$  accesses a global resource  $r$ , the system ceiling  $\Pi(t, p)$  is raised to  $\varphi_p(r)$  making the task non-preemptable on  $p$ . Then, the task checks if the resource is free: in this case, it locks the resource and executes the critical section. Otherwise, the task is inserted in  $r$ 's global FIFO queue, and then performs a spin-lock.
5. When at time  $t$  a task  $\tau_i$  allocated to processor  $p$  releases a global resource  $r$ , the algorithm checks the corresponding FIFO queue, and, in case some other task  $\tau_j$  is waiting, it grants access to  $r$ , otherwise  $r$  is unlocked. Then, the system ceiling  $\Pi(t, p)$  is restored to the previous value.

Our PSRP algorithm presented in the following section differs from MSRP in the following ways:

- MSRP disallows nested global critical sections, allowing a task to acquire only a single global resource at a time. PSRP supports global nested critical section by allowing each segment to acquire several global resources, effectively shifting the inner critical sections outward Havender (1968).
- PSRP supports multi-unit non-preemptive resources, while MSRP supports only single-unit non-preemptive resources.
- Under MSRP each task requires exactly one preemptive resource, while PSRP allows a task segment to require several preemptive resources (e.g. several processors in parallel).
- MSRP assumes that all segments of one task are assigned to the same processor. Under PSRP, each segment can start executing on a new set of processors. In multi-processor terminology we can say that our tasks can migrate across processors at segment boundaries.
- Under MSRP, segments requiring global resources execute non-preemptively. PSRP extends the notion of a global resource, allowing to schedule parallel segments requiring several preemptive resources non-preemptively.

### 5.3 Towards multi-resource sharing

At a first glance the gang scheduling problem looks similar to synchronization on multiprocessors: several tasks can execute concurrently and can share global logical resources. There are variants where tasks are assigned to a particular processor (partitioned scheduling) and where tasks or jobs can migrate between processors (global scheduling). While global scheduling deals with both allocating tasks to processors and scheduling them during runtime, partitioned scheduling assumes task allocation is done offline limiting the runtime effort to local scheduling on each processor. In both cases, however, a task executes on exactly one processor at a time. The key part of gang scheduling is that a task needs to execute on several preemptive resources

(e.g. processors) or non-preemptive resources (e.g. graphical processing units) concurrently. A gang scheduler resembles a global multiprocessor scheduler in the sense that it maintains a global ready queue, while in partitioned multiprocessor scheduling each processor maintains its local ready queue and schedules tasks locally.

Nested critical sections are problematic, as they may lead to deadlock. There are several approaches to deal with nested *global* critical sections:

- Disallow nested global critical sections (Rajkumar et al., 1988; Gai et al., 2003)
- Allow locking of resources only in a particular order (Hansen, 1973)
- Claim nested resources simultaneously (Block et al., 2007; Brandenburg and Anderson, 2008b)
- Allow arbitrary nesting (Dijkstra, 1964; Habermann, 1969)

Similar to the Banker's algorithm, the original SRP protocol handles multi-unit resources, however, SRP applies only to uniprocessor systems. It follows a similar idea to the one behind the Banker's algorithm: if a request for a resource could ever lead to deadlock, the request is denied. While the Banker's algorithm checks for deadlock at the time the task tries to access a non-preemptive resource, SRP checks for deadlock at the time the task is activated. The resource ceilings in SRP, which are computed offline, are a very efficient encoding of the deadlock conditions. During runtime, whenever a task requests a resource, the scheduler avoids deadlock by simply checking if the task's priority is higher than the system ceiling. Priority based scheduling together with a known priority assignment policy (e.g Rate Monotonic or Deadline Monotonic) and timing properties of tasks, allow to also provide real-time guarantees. Therefore, SRP can be regarded as an efficient implementation of the Banker's algorithm in the domain of priority-based uniprocessor scheduling with shared resources and real-time constraints.

Following a similar reasoning one may consider MSRP as an efficient implementation of the Banker's algorithm in the domain of priority-based multiprocessor scheduling with shared resources and real-time constraints. In the remainder of this chapter we present PSRP, a scheduling algorithm for fixed-priority parallel-task scheduling with shared resources and real-time constraints, which was inspired by the Banker's algorithm, SRP and MSRP.

## 5.4 System model

In the remainder of this chapter we assume that all preemptive resources are single-unit, i.e.,

$$\forall r \in \mathcal{P} : N_r = 1. \quad (5.1)$$

This restriction does not apply to non-preemptive resources, which can have arbitrarily many units, i.e.,

$$\forall r \in \mathcal{N} : N_r \geq 1. \quad (5.2)$$

### 5.4.1 Local vs. global resources

MSRP distinguishes between local and global resources. A resource is called local if it is accessed only by tasks assigned to the same processor, otherwise it is called global. Let us first explain the rationale behind local and global resources before adapting their definition to parallel processor systems.

In uniprocessor systems, when a task blocks on a shared logical resource, then the resource can only be released by another task running on the same processor. Therefore, the only option is to suspend the blocked task and allow the other task to continue, so that eventually the resource will be released.

In multiprocessor systems, a task can access local and global resources. Similar to uniprocessors, a local resource can only be acquired by another task running on the same processor. A global resource, however, can be acquired by a task running on a different processor. We therefore have two options for a task blocked on a global resource: we can either suspend it and allow another task assigned to the same processor to do useful work while the blocked task is waiting, or we can have the blocked task perform a spin-lock (also called a “busy wait”).

In either case, the blocking time has to be taken into account in the schedulability analysis. When a task suspends on a global resource, it allows lower priority tasks to execute and lock other resources, potentially leading to priority inversion. When a task spins on a global resource, it wastes the processor time which could have been used by other tasks. Hence, for global resources it is very important to keep the resource holding times short, more important than for local resources. In the multiprocessor case it therefore makes sense to distinguish between local and global resources.

Similar to MSRP, our PSRP algorithm relies on the notion of local and global resources. Unfortunately, the definition of local and global resources for MSRP in Section 5.2.4 assumes that a task requires exactly one processor, and hence it is not sufficient for our parallel task model. Also, it considers only non-preemptive resources, which makes it impossible to model tasks which require several (preemptive) processors at the same time. We therefore generalize the notion of local and global resources. The essential property of a global resource is that it is required by segments which can attempt to access their resources independently of each other (e.g. segments which are not sequentialized on one shared processor).

**Definition 5.2.** We define  $\kappa : \mathcal{R} \rightarrow \mathcal{S}$ , where  $\kappa(r)$  is the set of segments requiring resource  $r$ , i.e.

$$\kappa(r) = \{s \in \mathcal{S} \mid r \in R_s\} \quad (5.3)$$

**Definition 5.3** (Local and global resources). We define a resource  $r$  as local if (i) it is preemptive and accessed only by segments which require only nonpreemptive resources besides  $r$ , or (ii) it is nonpreemptive and accessed only by segments which also share one and the same preemptive resource  $p$ . Otherwise the resource is global. More formally, we use  $\mathcal{R}^L$  and  $\mathcal{R}^G$  to denote the sets of local and global resources,



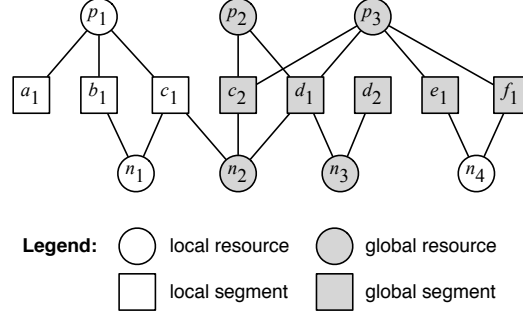


Figure 5.1: Example illustrating local and global resources and segments in a segment requirements graph, for a system comprised of  $\mathcal{P} = \{p_1, p_2, p_3\}$ ,  $\mathcal{N} = \{n_1, n_2, n_3, n_4\}$ ,  $\mathcal{S} = \{a_1, b_1, c_1, c_2, d_1, d_2, e_1, f_1\}$ ,  $\Gamma = \{a, b, c, d, e, f\}$  with  $S_a = \langle a_1 \rangle$ ,  $S_b = \langle b_1 \rangle$ ,  $S_c = \langle c_1, c_2 \rangle$ ,  $S_d = \langle d_1, d_2 \rangle$ ,  $S_e = \langle e_1 \rangle$ ,  $S_f = \langle f_1 \rangle$ , and  $R_{a_1} = \{p_1\}$ ,  $R_{b_1} = \{p_1, n_1\}$ ,  $R_{c_1} = \{p_1, n_1, n_2\}$ ,  $R_{c_2} = \{p_2, p_3, n_2\}$ ,  $R_{d_1} = \{p_2, p_3, n_2, n_3\}$ ,  $R_{d_2} = \{n_3\}$ ,  $R_{e_1} = R_{f_1} = \{p_3, n_4\}$ .

respectively, such that

$$\begin{aligned} \mathcal{R}^L &= \{r \in \mathcal{P} \mid \forall b \in \kappa(r) : (\forall s \in R_b \setminus \{r\} : s \in \mathcal{N})\} \cup \\ &\quad \{r \in \mathcal{N} \mid (\forall b \in \kappa(r) : R_b \cap \mathcal{P} \neq \emptyset) \wedge (|\mathcal{P} \cap \bigcup_{b \in \kappa(r)} R_b| = 1)\}, \\ \mathcal{R}^G &= \mathcal{R} \setminus \mathcal{R}^L. \end{aligned} \quad (5.4)$$

Notice that the local/global classification in MSRP is limited only to nonpreemptive resources, while in our definition it also includes preemptive resources. Figure 5.1 illustrates the difference between local and global resources.

#### 5.4.2 Local vs. global segments

Similarly to MSRP distinguishing between local and global critical sections (guarding access to local and global resources, respectively), in PSRP we distinguish between local and global segments.

**Definition 5.4** (Local and global segments). *We define a local segment as one requiring at least one local preemptive resource, otherwise the segment is global. More formally, we use  $\mathcal{S}^L$  and  $\mathcal{S}^G$  to denote the sets of local and global segments, respectively, such that*

$$\begin{aligned} \mathcal{S}^L &= \{s \in \mathcal{S} \mid \exists r \in R_s : r \in \mathcal{R}^L \cap \mathcal{P}\}, \\ \mathcal{S}^G &= \mathcal{S} \setminus \mathcal{S}^L. \end{aligned} \quad (5.5)$$

Figure 5.1 illustrates the difference between local and global segments. The intention of PSRP is to schedule segments  $a_1, b_1, c_1$  preemptively and in fixed-priority order, and  $c_2, d_1, d_2, e_1, f_1$  nonpreemptively and in FIFO order.

Resource holding time is the duration of a continuous time interval during which a segment owns a resource, preventing other segments to access it. Minimizing the holding time is important in the schedulability analysis, as it adds to the blocking time. Nonpreemptive scheduling of global segments will keep the holding times of global resources short. Our choice for executing global segments in FIFO order is in line with MSRP.

When identifying global and local resources and segments, we consider only the identity of the required resources, i.e. a resource or a segment will be marked local or global irrespective of the number of required resource units.

A segment can experience *interference* from other segments: it can be *blocked* by lower priority segments and *preempted* by higher priority segments (on a local resource). The term *waiting* is used to describe competition for a global resource. A local segment can wait only for global segments using the same global resource. A global segment can wait for both global and local segments sharing a global resource. During its execution a local segment may be preempted by higher priority segments (global segments execute non-preemptively).

Figure 5.2 shows the state diagrams for a local and global segment. A segment may be *inactive*, *ready*, *executing*, *waiting* (on a global resource), or *blocked* (on a local resource). A local segment may be both waiting and blocked at the same time.

A task  $\tau_i$  inherits the state from its currently active segment. If no segment in  $S_i$  is active, then task  $\tau_i$  is considered inactive.

## 5.5 Parallel-SRP (PSRP)

In this section we present the Parallel-SRP (PSRP) algorithm, which is inspired by MSRP and can be regarded as its generalization to the parallel task model. The PSRP algorithm follows the following set of rules:

1. For local resources the algorithm is the same as SRP. In particular, for every local nonpreemptive resource  $r \in \mathcal{R}^L \cap \mathcal{N}$ , we define a *resource ceiling*  $\varphi(r)$  to be equal to the highest priority of any task requiring  $r$ . For every local preemptive resource  $p \in \mathcal{R}^L \cap \mathcal{P}$  we define a system ceiling<sup>2</sup>  $\Pi(t, p)$  which at any moment  $t$  is equal to the maximum resource ceiling among all local nonpreemptive resources locked by any segment that also locks  $p$ . We also equip  $p$  with a *ready queue*  $queue(p)$ , which stores tasks waiting for or executing on  $p$  sorted by priority. At time  $t$ , a task at the head of  $queue(p)$  is allowed to preempt a task already executing on  $p$  only if its priority is higher than  $\Pi(t, p)$ .
2. Each global resource  $r \in \mathcal{R}^G$  is equipped with a FIFO *resource queue*  $queue(r)$ <sup>3</sup>, which stores tasks waiting for or executing on  $r$ .

<sup>2</sup>The term “system ceiling” suggests a system wide property, rather than a property of each preemptive resource. We chose this terminology, however, to comply with the existing literature on multiprocessor synchronization (Gai et al., 2001).

<sup>3</sup>MSRP also equips each global resource with a FIFO queue.

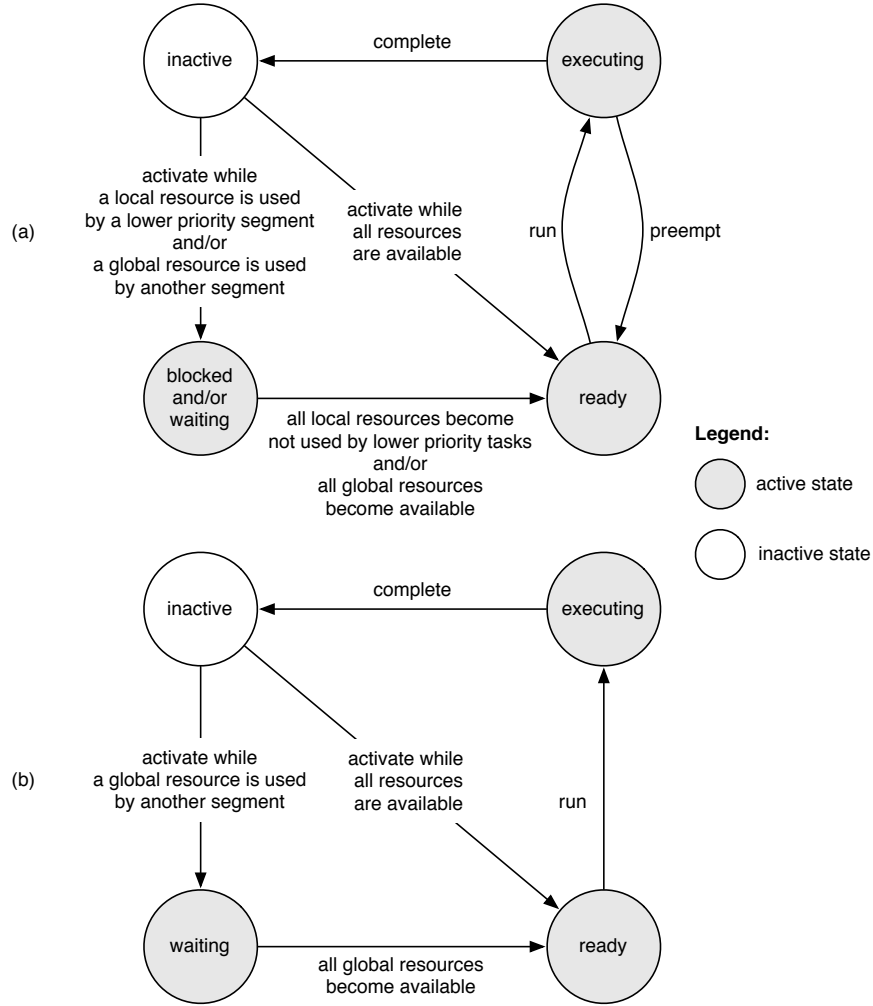


Figure 5.2: State diagram for (a) a local segment, and (b) a global segment.

3. When a task  $\tau_i$  attempts to lock a set of resources  $R$  using  $lock(R)$ , it is inserted into the resource queues of all global resources in  $R$ . Moreover, this insertion is atomic, meaning that no other task can be inserted into any of the resource queues in  $R$  before  $\tau_i$  has been inserted into all queues in  $R$ .

When a task  $\tau_i$  releases a set of resources  $R$  using  $unlock(R)$ , it is removed from the resource queues of all global resources in  $R$ . Each  $unlock(R)$  must be preceded by a  $lock(R)$  call, with the same  $R$ .

4. A task  $\tau_i$  is said to be *ready* at time  $t$  if for all resources  $r$  required by its currently active segment  $\tau_{i,j}$  the following conditions hold:

- $r$  has enough units available
- if  $r \in \mathcal{R}^L \cap \mathcal{P}$  then  $\tau_i$ 's priority is higher than the system ceiling of  $r$  at time  $t$
- if  $r \in \mathcal{R}^G \cup (\mathcal{R}^L \cap \mathcal{P})$  then all tasks in front of it at the head of  $queue(r)$  are also ready.

More formally,

$$\begin{aligned}
 ready(t, \tau_i) \equiv & ( \forall r \in R_{\alpha(t, \tau_i)} : N_r \leq \eta(t, r) ) \wedge \\
 & ( \forall r \in R_{\alpha(t, \tau_i)} \cap (\mathcal{R}^L \cap \mathcal{P}) : \pi_i < \Pi(t, r) ) \wedge \\
 & ( \forall r \in R_{\alpha(t, \tau_i)} \cap (\mathcal{R}^G \cup (\mathcal{R}^L \cap \mathcal{P})) : head(t, queue(r), k) = \tau_i \\
 & \wedge \forall j \in \mathbb{N} : 0 \leq j < k \Rightarrow ready(t, head(t, queue(r), j)) )
 \end{aligned}$$

where  $head(t, q, i)$  is the  $i^{\text{th}}$  element from the head of queue  $q$  at time  $t$ , with  $head(t, q, 0)$  being the first element at the head of  $q$  at time  $t$ .

5. The scheduler is called after adding a task  $\tau_i$  to a ready queue or a resource queue, or if after removing a task from a queue the queue is not empty. The scheduler then traverses the ready and resource queues and schedules all the ready tasks which become executing. This will occupy a certain number of units of the resources in the system. If so, then the task is scheduled and becomes executing. Otherwise,  $\tau_i$  performs a spin-lock (at the highest priority) on each resource containing  $\tau_i$  at the head of its queue and becomes waiting<sup>4</sup>.
6. The following invariant is maintained: the system ceiling of each local preemptive resource is equal to the top priority whenever a segment requiring a global resource is spinning or executing on it, i.e.

$$\forall t \in \mathcal{T}, p \in \mathcal{P} : (\exists s \in \mathcal{S} : R_s \cap \mathcal{R}^G \neq \emptyset \wedge \sigma(t, s, p) > 0) \Rightarrow \Pi(t, p) = \pi_{\top}$$

According to rule 5, if a task  $\tau_i$  requires several units of a resource  $r$ , then it will wait by performing a spin-lock until enough of the segments currently using  $r$  have completed and released a sufficient number of units of  $r$  for  $\tau_i$  to start executing.

Notice that a segment does not have to require a preemptive resource. It is very well possible for a segment to require only non-preemptive resources. On some resources “spinning” may not make sense (e.g. spinning on a bus). The spinning operation essentially “reserves” a resource, preventing other tasks to execute on it. It can therefore be implemented differently on different resources. Figure 5.3 shows the state diagram for any resource (preemptive or non-preemptive). It distinguishes between actively “using” a resource and “reserving” it without doing actual work (e.g. spinning).

---

<sup>4</sup>On some resources “spinning” may not make sense (e.g. spinning on a bus). Spinning essentially “reserves” a resource, preventing other tasks from executing on it, and can be implemented differently on different resources.

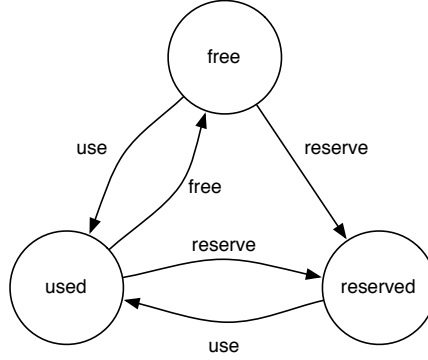


Figure 5.3: State diagram for a resource.

## 5.6 Schedulability analysis for PSRP

We first show that PSRP does not suffer from deadlock nor from livelock, and then we proceed to bound the worst-case response time (WCRT) of tasks.

**Lemma 5.1.** *PSRP does not suffer from deadlock.*

*Proof.* Access to local resources is synchronized using SRP, which was proved to be deadlock free by Baker (1991). We therefore need to show the absence of dependency cycles when accessing global resources, in particular when (i) segments are waiting for resources and (ii) after they have started executing.

(i) Without loss of generality, let us assume that there is a deadlock between two segments  $\tau_{i,j}$  and  $\tau_{x,y}$ , which require two resources  $r_1, r_2 \in R_{i,j} \cap R_{x,y}$ . For the deadlock to occur each segment would have to acquire one of the resources, while the other resource was held by the other segment. Consequently, since the resource queues handle the tasks in FIFO order, there would have to be a moment when the resource queues were in the following state:  $queue(r_1) = \langle \dots, \tau_{i,j}, \dots, \tau_{x,y}, \dots \rangle$  and  $queue(r_2) = \langle \dots, \tau_{x,y}, \dots, \tau_{i,j}, \dots \rangle$ . However, since the addition to all queues in  $R_{i,j}$  and  $R_{x,y}$  is atomic (see rule 3), such a queue state is not possible. Hence a deadlock cannot occur during the waiting phase.

(ii) Since we have assumed that critical sections do not span across task segments and that all resources required by a segment are provided simultaneously, once a segment starts executing it will be able to complete and release the acquired resources. Hence a deadlock cannot occur during the execution phase.  $\square$

**Lemma 5.2.** *PSRP does not suffer from livelock.*

*Proof.* Similarly to Lemma 5.1, we need to show the absence of livelock when segments are accessing global resources. In particular, we need to show that every segment  $\tau_{i,j}$  requiring global resources will eventually start executing. According to Lemma 5.1, a segment will not deadlock during its waiting phase, so, as long as other segments waiting in a resource queue in front of it eventually complete, it will

eventually start executing. Since each segment needs to execute for a finite amount of time, and since, according to Lemma 5.1, it will not deadlock during the execution phase either, every segment inserted into a resource queue will have to wait for at most a finite amount of time. Hence a livelock cannot occur.  $\square$

To show that task are schedulable, we derive the bound on the WCRT of all segments, and check if the WCRT of the last segment of every task, measured from the arrival time of the task, is within the task's deadline.

**Lemma 5.3.** *A local segment requires exactly one preemptive local resource.*

*Proof.* According to Definition 5.3, segments which share a local resource share exactly one preemptive resource. According to Definition 5.4, every local segment requires at least one local preemptive resource. Lemma 5.3 follows.  $\square$

**Lemma 5.4.** *A global segment requires at least one global resource.*

*Proof.* Let  $s$  be a global segment, i.e.  $s \in \mathcal{S}^G$ . Then,

$$\begin{aligned}
 s &\in \mathcal{S}^G \\
 &\equiv \{ \text{Equation (5.5)} \} \\
 &\quad \forall r \in R_s : \neg(r \in \mathcal{R}^L \wedge r \in \mathcal{P}) \\
 &\equiv \{ \text{Equations (5.5), (5.4), (2.1)} \} \\
 &\quad \forall r \in R_s : r \in \mathcal{R}^G \vee r \in \mathcal{N} \\
 &\equiv \{ \text{Equation (5.4)} \} \\
 &\quad \forall r \in R_s : r \in \mathcal{R}^G \vee (r \in \mathcal{N} \wedge r \in \mathcal{R}^G) \vee (r \in \mathcal{N} \wedge r \in \mathcal{R}^L) \\
 &\equiv \forall r \in R_s : r \in \mathcal{R}^G \vee (r \in \mathcal{N} \wedge r \in \mathcal{R}^L) \tag{5.6}
 \end{aligned}$$

We introduce the following shorthand notation to simplify the representation of (5.4) in the remainder of this proof:

$$\begin{aligned}
 f(r) &\stackrel{\text{def}}{=} \forall b \in \kappa(r) : (\forall s \in R_b \setminus \{r\} : s \in \mathcal{N}), \\
 g(r) &\stackrel{\text{def}}{=} \forall b \in \kappa(r) : R_b \cap \mathcal{P} \neq \emptyset, \\
 h(r) &\stackrel{\text{def}}{=} |\mathcal{P} \cap \bigcup_{b \in \kappa(r)} R_b| = 1.
 \end{aligned}$$

We can safely assume that each segment requires at least one resource, i.e.  $\forall s \in \mathcal{S} : R_s \neq \emptyset$ . We now show by contradiction that no segment can require only local

non-preemptive resources, i.e. we show that  $\neg \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge r \in \mathcal{R}^L)$ :

$$\begin{aligned}
& \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge r \in \mathcal{R}^L) \\
\equiv & \quad \{ \text{Equation (5.4), definition of } f(r), g(r) \text{ and } h(r) \} \\
& \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge ((r \in \mathcal{P} \wedge f(r)) \vee (r \in \mathcal{N} \wedge g(r) \wedge h(r)))) \\
\equiv & \quad \{ \text{Equation (2.1)} \} \\
& \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge g(r) \wedge h(r)) \\
\Rightarrow & \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge g(r)) \\
\equiv & \quad \{ \text{Definition of } g(r) \} \\
& \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge (\forall b \in \kappa(r) : R_b \cap \mathcal{P} \neq \emptyset)) \\
\Rightarrow & \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N} \wedge ((\exists q \in R_s : q \in \mathcal{P}) \vee R_s = \emptyset)) \\
\equiv & \quad \{ \forall s \in \mathcal{S} : R_s \neq \emptyset \} \\
& \exists s \in \mathcal{S} : (\forall r \in R_s : r \in \mathcal{N}) \wedge (\exists q \in R_s : q \in \mathcal{P}) \\
\equiv & \exists s \in \mathcal{S} : \text{false} \\
\equiv & \text{false}
\end{aligned}$$

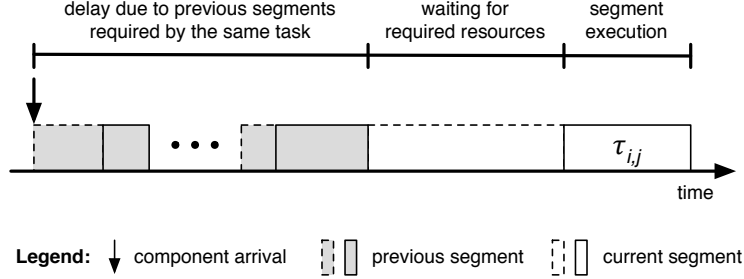
Since no segment can require only local non-preemptive resources, and since according to (5.6) each resource required by a global segment is either local non-preemptive or global, each global segment must require at least one global resource.  $\square$

According to Lemma 5.3, a local segment requires exactly one preemptive resource. This preemptive resource will dictate the behavior of the local segments sharing it. PSRP will use the priority-ordered ready queue to schedule local segments based on their priority. According to Lemma 5.4, a global segment requires at least one global resource, and according to Definition 5.4 it does not require any local preemptive resources. PSRP will use the resource queues attached to the global resources to schedule the global segments nonpreemptively in FIFO order. In the remainder of this section we derive an equation for the WCRT for global and local segments.

### 5.6.1 Response time of global segments

A global segment spin-locks and executes on all its required resources at the highest priority. Consequently, as a global segment cannot be preempted, its response time is comprised of three time intervals, as illustrated in Figure 5.4.

The delay due to segments preceding  $\tau_{i,j}$  in  $R_i$  is equal to the response time of the previous segment  $\tau_{i,j-1}$  for  $j > 1$ , and 0 for  $j = 1$  (i.e. the first segment in sequence  $R_i$ ). If  $\tau_{i,j-1}$  is global, then its response time is again comprised of the three intervals illustrated in Figure 5.4. The response time of a local segment is discussed later in Section 5.6.2. The execution time of segment  $\tau_{i,j}$  is simply  $E_{i,j}$ . The interesting part is the time that segment  $\tau_{i,j}$  spends waiting for resources in  $R_{i,j}$  (which includes the spin-lock time).

Figure 5.4: Response time of a global segment  $\tau_{i,j}$ .

The MSRP algorithm assumes that at any time each global segment  $\tau_{i,j}$  requires one preemptive and at most one nonpreemptive resource. Also, access to a global resource is granted to segments in FIFO order. Consequently, they observe that the worst-case spinning time of segment  $\tau_{i,j}$  on a preemptive resource is equal to the sum of the segment execution times of all segments sharing the nonpreemptive resource with  $\tau_{i,j}$ . In our model, a segment can require an arbitrary number of preemptive and nonpreemptive resources, which may result in a longer spinning time.

**Definition 5.5.** *The requirements of all segments can be represented by a segment requirements graph  $G = (V, E)$  where the set of vertices  $V = \mathcal{R} \cup \mathcal{S}$ , and the set of edges  $E \subseteq 2^{\mathcal{S} \times \mathcal{R}}$  represents the resource requirements of segments, i.e.*

$$(\tau_{i,j}, r) \in E \Leftrightarrow (\tau_{i,j} \in \mathcal{S} \wedge r \in R_{i,j}). \quad (5.7)$$

The graph is tripartite, as we can divide  $E$  into two disjoint sets  $E^{\mathcal{P}}$  and  $E^{\mathcal{N}}$ , such that

$$\forall (\tau_{i,j}, r) \in E^{\mathcal{P}} : (\tau_{i,j} \in \mathcal{S} \wedge r \in \mathcal{P}), \quad (5.8)$$

$$\forall (\tau_{i,j}, r) \in E^{\mathcal{N}} : (\tau_{i,j} \in \mathcal{S} \wedge r \in \mathcal{N}). \quad (5.9)$$

A segment requirements graph may seem similar to a resource allocation graph introduced by Holt (1972) and used for deadlock analysis. However, unlike a resource allocation graph, a segment requirements graph is an undirected graph. Its purpose is finding dependencies between budgets to compute the waiting times in Chapter 5, rather than identifying deadlocks due to cyclic dependencies.

**Notation** In the segment requirements graphs we draw budget nodes as rectangles and resource nodes as circles.

**Example 5.1.** Consider a platform comprised of four processors  $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$ , executing an application consisting of four tasks, each containing one segment. We name these segments  $\mathcal{S} = \{a, b, c, d\}$ , and define their resource requirements as follows:  $R_a = \{p_1\}$ ,  $R_b = \{p_1, p_2\}$ ,  $R_c = \{p_2, p_3\}$ , and  $R_d = \{p_3, p_4\}$ , as shown in Figure 5.5. Notice that all resources and segments are global.



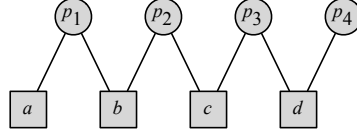


Figure 5.5: A segment requirements graph for a system comprised of  $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$ ,  $\mathcal{N} = \emptyset$ ,  $\mathcal{S} = \{a, b, c, d\}$  with  $R_a = \{p_1\}$ ,  $R_b = \{p_1, p_2\}$ ,  $R_c = \{p_2, p_3\}$ , and  $R_d = \{p_3, p_4\}$ .

Let us assume a scenario, where the processors are idle and segments  $a, b, c, d$  arrive soon after each other, as shown in Figure 5.6. When segment  $a$  arrives and processor  $p_1$  is idling, it is immediately scheduled and starts executing. When segment  $b$  arrives, requiring processors  $p_1$  and  $p_2$ , and encounters a busy processor  $p_1$ , it is added to the resource queues of  $queue(p_1)$  and  $queue(p_2)$ . Since it is at the head of  $queue(p_2)$  it starts spinning on  $p_2$  (at the highest priority). Soon after segment  $c$  arrives and similarly is inserted into the resource queues of  $queue(p_2)$  and  $queue(p_3)$  and starts spinning on  $p_3$ . When segment  $d$  arrives soon after segment  $c$ , it is inserted into  $queue(p_3)$  and  $queue(p_4)$  and starts spinning on  $p_4$ . When segment  $a$  completes and releases  $p_1$ , it is removed from  $queue(p_1)$ , enabling segment  $b$ , which starts executing. This process continues, subsequently releasing segments  $c$  and  $d$ . Notice that segment  $d$  cannot start executing before  $c$  has completed, which cannot start before  $b$  has completed, which cannot start before  $a$  has completed.  $\square$

A segment may be required to wait inside of a resource queue, either passively waiting in the queue's tail or actively spinning at its head. Example 5.1 suggests that, under PSRP, a segment may need to wait on its required resources until all segments which it “depends on” in the segment requirements graph have completed. We can observe, however, that some of the dependencies in a segment requirements graph are not feasible. In particular,

- Segments belonging to the same task are executed sequentially (by definition), and therefore cannot interfere with each other. We can therefore ignore segments belonging to task  $\tau_i$  when computing the time that segment  $\tau_{i,j}$  may need to wait.
- A segment depends only on other segments which share global resources, because only segments which require at least one global resource may wait inside of a resource queue (as resource queues are defined only for global resources). Segments which require only local resources will never be inserted into a resource queue. Therefore, such segments will not interfere with other global segments and can be safely ignored when computing the time that a segment may need to wait.

We now define the notion of a *partial segment requirements graph*, which includes only those dependencies in a segment requirements graph which are indeed feasible. We use these graphs later to formalize the notion of dependency.

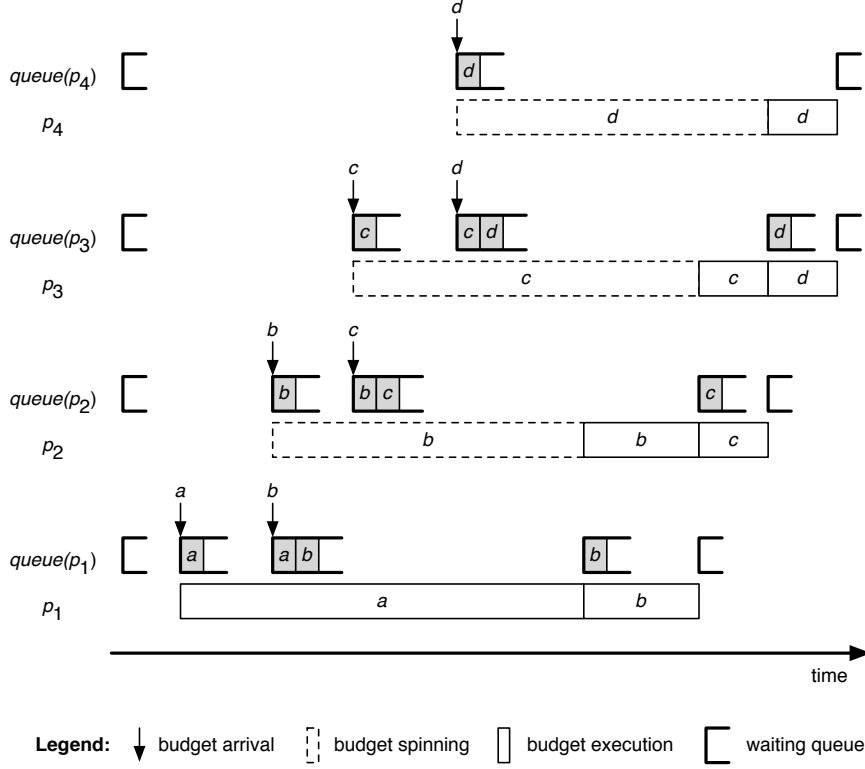


Figure 5.6: Example of transitive blocking of global segments. The figure shows the arrival and execution of segments  $\mathcal{S} = \{a, b, c, d\}$  on preemptive resources  $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$  and the contents of their resource queues. Since we assumed that each task contains only one segment, for ease of presentation we refer to the tasks inside the resource queues by the corresponding segment names.

**Definition 5.6.** A partial segment requirements graph  $G' = (V', E')$  derived from segment requirements graph  $G = (V, E)$  is a subgraph of  $G$ , with  $V' \subseteq V$  and  $E' \subseteq E$ , such that

- 1)  $V'$  contains all global resources, but no local resources, i.e.

$$\mathcal{R}^G \subseteq V' \wedge \mathcal{R}^L \cap V' = \emptyset,$$

- 2) segments requiring only local resources are ignored

$$\forall \tau_{i,j} \in \mathcal{S} : R_{i,j} \subseteq \mathcal{R}^L \Rightarrow \tau_{i,j} \notin V',$$

- 3) if  $\tau_i$  requires at least one global resource, then there is exactly one segment

from  $R_i$  in  $V'$ , i.e.

$$\begin{aligned} \forall \tau_i \in \Gamma : ((\exists \tau_{i,j} \in R_i : R_{i,j} \cap \mathcal{R}^G \neq \emptyset) \\ \Rightarrow |\{\tau_{i,j} \mid \tau_{i,j} \in V'\}| = 1), \end{aligned}$$

4)  $E'$  contains all the edges (and only those edges) from  $E$  which have both end-points in  $V'$ , i.e.

$$\forall \{a, b\} \in E : (a \in V' \wedge b \in V') \Leftrightarrow \{a, b\} \in E'.$$

Condition 1 in Definition 5.6 makes sure that segments which require only local resources will be unreachable from global segments. Condition 2 removes those segments from a partial segment requirements graph to keep it concise. Condition 3 discards segments belonging to the same task.

**Definition 5.7.** We define  $\text{partial}(G)$  as the set of all possible partial segment requirements graphs which can be derived from the segment requirements graph  $G$ .

**Lemma 5.5.** The  $\text{partial}(G)$  set contains

$$\prod_{\tau_i \in \Gamma} |\{\tau_{i,j} \mid \tau_{i,j} \in R_i \wedge R_{i,j} \cap \mathcal{R}^G \neq \emptyset\}| \leq \prod_{\tau_i \in \Gamma} |R_i|$$

graphs.

*Proof.* It follows directly from conditions 2 and 3 in Definition 5.6. The inequality is due to the fact that a partial segment requirements graph does not contain segments which require only local resources.  $\square$

Figure 5.7 illustrates the partial graphs derived from the segment requirements graph in Figure 5.1.

**Definition 5.8.** Let  $G = (V, E)$  be a segment requirements graph. We define  $\delta(\tau_{i,j}, g)$  to be the set of segments which  $\tau_{i,j}$  can reach in the partial segment requirements graph  $g \in \text{partial}(G)$ . We say that “ $\tau_{i,j}$  can reach  $\tau_{x,y}$  in  $g$ ” iff both segments belong to the same connected subgraph of  $g$ , and  $\tau_{i,j} \neq \tau_{x,y}$ . We say that “ $\tau_{i,j}$  depends on  $\tau_{x,y}$ ” iff

$$\exists g \in \text{partial}(G) : \tau_{x,y} \in \delta(\tau_{i,j}, g). \quad (5.10)$$

Notice that the dependency relation is symmetric, i.e.

$$\tau_{x,y} \in \delta(\tau_{i,j}, g) \Leftrightarrow \tau_{i,j} \in \delta(\tau_{x,y}, g), \quad (5.11)$$

and transitive, i.e.

$$\tau_{x,y} \in \delta(\tau_{i,j}, g) \wedge \tau_{i,j} \in \delta(\tau_{a,b}, g) \Rightarrow \tau_{x,y} \in \delta(\tau_{a,b}, g). \quad (5.12)$$

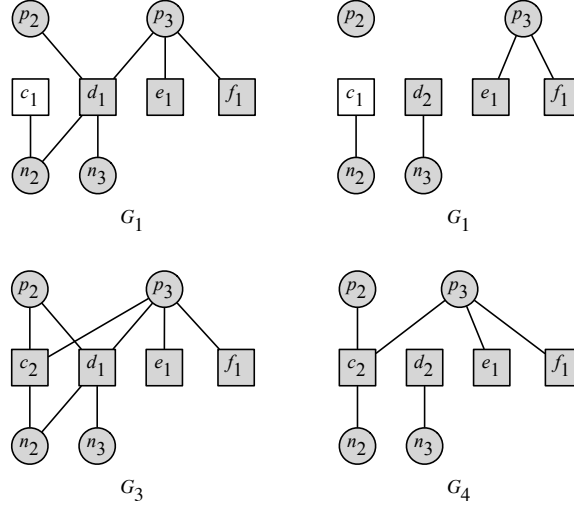


Figure 5.7: Partial segment requirements graphs derived from the segment requirements graph in Figure 5.1, assuming tasks  $\Gamma = \{a, b, c, d, e, f\}$ , with  $S_a = \langle a_1 \rangle$ ,  $S_b = \langle b_1 \rangle$ ,  $S_c = \langle c_1, c_2 \rangle$ ,  $S_d = \langle d_1, d_2 \rangle$ ,  $S_e = \langle e_1 \rangle$ ,  $S_f = \langle f_1 \rangle$ .

**Example 5.2.** Figure 5.8 shows an example of the dependencies in the partial segment requirements graphs in Figure 5.7.

$\tau_{i,j}$	$\delta(\tau_{i,j}, G_1)$	$\delta(\tau_{i,j}, G_2)$	$\delta(\tau_{i,j}, G_3)$	$\delta(\tau_{i,j}, G_4)$
$c_1$	$\{d_1, e_1, f_1\}$	$\emptyset$	$\emptyset$	$\emptyset$
$c_2$	$\emptyset$	$\emptyset$	$\{d_1, e_1, f_1\}$	$\{e_1, f_1\}$
$d_1$	$\{c_1, e_1, f_1\}$	$\emptyset$	$\{c_2, e_1, f_1\}$	$\emptyset$
$d_2$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$e_1$	$\{c_1, d_1, f_1\}$	$\{f_1\}$	$\{c_2, d_1, f_1\}$	$\{c_2, f_1\}$
$f_1$	$\{c_1, d_1, e_1\}$	$\{e_1\}$	$\{c_2, d_1, e_1\}$	$\{c_2, e_1\}$

Figure 5.8: Dependencies for segments in Figure 5.7, where  $G_1, G_2, G_3, G_4$  represents the partial segment requirements graphs in Figure 5.7.

□

**Lemma 5.6.** Under PSRP, each segment  $\tau_{i,j} \in \mathcal{S}$  will have to wait on global resources before it can start executing for at most

$$\text{wait}(\tau_{i,j}) = \max \left( 0, \max_{g \in \text{partial}(G)} \sum_{\tau_{x,y} \in \delta(\tau_{i,j}, g)} E_{x,y} \right), \quad (5.13)$$

where  $G$  is the segment requirements graph.

*Proof.* Consider the situation when a segment  $\tau_{i,j}$  tries to start executing and acquire resources in  $R_{i,j}$ . If any of the resources is not available,  $\tau_{i,j}$  will have to wait. Let  $wait\_resource(\tau_{i,j}, r)$  be the worst-case time that segment  $\tau_{i,j}$  may spend waiting due to resource  $r$ .

When  $\tau_{i,j}$  tries to access a global resource  $r$  which is not available, then  $\tau_i$  will be inserted at the end of  $queue(r)$ . Since  $queue(r)$  is a FIFO queue, a segment  $\tau_{x,y}$  residing inside of  $queue(r)$  in front of  $\tau_{i,j}$  will have to complete first, before  $\tau_x$  can be added at the end of  $queue(r)$  again. Hence a task may be represented only once inside of a resource queue, and therefore the length of the resource queue is at most equal to the number of tasks requiring  $r$ . In other words, a task  $\tau_x$ , which is sharing resource  $r$  with segment  $\tau_{i,j}$ , will interfere with  $\tau_{i,j}$  (during the time  $\tau_{i,j}$  is waiting on  $r$ ) for the duration of at most one of its segments in  $R_{\tau_x}$ .

Let  $B(\tau_{i,j}, r)$  be the worst-case set of segments which are waiting in  $queue(r)$  in front of  $\tau_{i,j}$ . Each segment  $\tau_{x,y} \in B(\tau_{i,j}, r)$  can itself be waiting on other resources: for each resource  $s \in R_{x,y}$ , segment  $\tau_{x,y}$  may need to wait for all segments in  $B(\tau_{x,y}, s)$ . For each of those segments in  $B(\tau_{x,y}, s)$  we can apply the same reasoning. In effect, segment  $\tau_{i,j}$  may need to wait for many segments which it indirectly depends on. A straightforward approach would be to designate all segments which are reachable from  $\tau_{i,j}$  in  $G$  as the set that segment  $\tau_{i,j}$  depends on. We now show how to bound this set by removing the unnecessary vertices from  $G$ .

(i) The fact that  $\tau_{x,y}$  is inside of a resource queue implies that its priority is higher or equal to the system ceiling of any preemptive resource it may require, meaning that it cannot be waiting any more for local resources. We therefore need to consider only global resources.

(ii) At any moment in time only one segment of a task can be active. Therefore, segment  $\tau_{i,j}$  will not depend on segments belonging to the same task, i.e segments in  $R_i \setminus \{\tau_{i,j}\}$ .

(iii) Segment  $\tau_{i,j}$  will not depend on any segment which a segment  $\tau_{i,k}$  from the same task depends on, unless  $\tau_{i,j}$  also depends on it after removing  $\tau_{i,k}$  from  $G$ . The same holds for any other segment in  $\mathcal{S}$ .

According to (i), (ii) and (iii) we need to consider only segments which are reachable from  $\tau_{i,j}$  in  $G$ , after we remove the vertices corresponding to the local resources, and segments belonging to the same task from  $G$ . In other words, segment  $\tau_{i,j}$  depends only on segments  $\tau_{x,y}$ , such that (according to Definition 5.7 and 5.8)  $\tau_{x,y} \in \delta(\tau_{i,j}, g)$ , where  $g \in partial(G)$ . Moreover, since (according to Lemma 5.1) there are no dependency cycles, we need to consider only a single job of each  $\tau_{x,y}$ .

Segment  $\tau_{i,j}$  will have to wait for  $wait\_resource(\tau_{i,j}, r)$  time on all resources  $r \in R_{i,j}$ . Since a segment is inserted into the resource queues of all resources  $r \in R_{i,j}$  simultaneously, and any spin-locks are performed concurrently, its total waiting time is given by (5.13).  $\square$

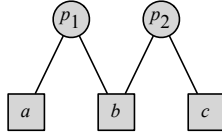
**Example 5.3.** Figure 5.9 shows an example of the waiting times for segments in the partial segment requirements graphs in Figure 5.7 for example values of  $E_{i,j}$ .

$\tau_{i,j}$	$E_{i,j}$	$wait(\tau_{i,j})$
$c_1$	2	3
$c_2$	2	3
$d_1$	2	3
$d_2$	16	0
$e_1$	0.5	4.5
$f_1$	0.5	4.5

Figure 5.9: Waiting times for segments in Figure 5.7.

□

Note that (5.13) is pessimistic. Figure 5.10 illustrates the source of the pessimism for  $wait(b)$ . According to PSRP, segment  $b$  may be delayed by  $a$  or  $c$ , but not both. Lemma 5.6, however, assumes that in the worst-case  $b$  will have to wait for both  $a$  and  $c$ , which is pessimistic in case  $a$  and  $c$  do not share a common resource.

Figure 5.10: A segment requirements graph for a system comprised of  $\mathcal{P} = \{p_1, p_2\}$ ,  $\mathcal{N} = \emptyset$ ,  $\mathcal{S} = \{a, b, c\}$  with  $R_a = \{p_1\}$ ,  $R_b = \{p_1, p_2\}$ , and  $R_c = \{p_2\}$ .

**Corollary 5.1.** *A segment  $\tau_{i,j}$  which requires only local resources will never have to wait inside of a resource queue, i.e.*

$$\forall \tau_{i,j} \in \mathcal{S} : R_{i,j} \subseteq \mathcal{R}^L \Rightarrow wait(\tau_{i,j}) = 0.$$

**Definition 5.9.** *For segment  $\tau_{i,j}$  we use  $E'(\tau_{i,j}) = wait(\tau_{i,j}) + E_{i,j}$  to denote the execution time of  $\tau_{i,j}$  extended with its waiting time.*

**Definition 5.10.** *We define  $A(\tau_{i,j})$  to be the worst-case activation time of segment  $\tau_{i,j}$  relative to the arrival time of its parent task  $\tau_i$ .  $A(\tau_{i,j})$  is equal to the WCRT of the previous segment in  $R_i$ , or 0 in case  $\tau_{i,j}$  is the first segment in  $R_i$ , i.e.*

$$A(\tau_{i,j}) = \begin{cases} WCRT(\tau_{i,j-1}) & \text{if } j > 1, \\ 0 & \text{otherwise.} \end{cases} \quad (5.14)$$

**Theorem 5.1.** *Under PSRP, the WCRT of a global segment  $\tau_{i,j} \in \mathcal{S}^G$ , measured since the arrival of the parent task, is bounded by*

$$WCRT(\tau_{i,j}) = A(\tau_{i,j}) + E'(\tau_{i,j}). \quad (5.15)$$

*Proof.* Since each segment  $\tau_{i,j}$  belonging to task  $\tau_i$  is dispatched only after the previous segment  $\tau_{i,j-1}$  has completed (or when  $\tau_i$  has arrived, in case of the first segment), and since we assumed  $D_i \leq T_i$  for all tasks  $\tau_i$ , segments belonging to the same task do not interfere with each other. Since each segment is dispatched immediately after the previous one has completed (or at the moment  $\tau_i$  has arrived, in case of the first segment), there is no idle time between the segments. Therefore, segment  $\tau_{i,j}$  will attempt to lock its required resources at time  $A(\tau_{i,j})$ .

At this moment it will start waiting on all the resources which it requires but which are unavailable. It will wait for at most  $wait(\tau_{i,j})$  time units.

Since segments spin at the highest priority, immediately after it stops spinning it will start executing. Also, since we assumed that all nested critical sections have been shifted outwards and since the system ceiling of all resources in  $R_{i,j}$  is raised to the top priority at the moment  $\tau_{i,j}$  starts executing, segment  $\tau_{i,j}$  cannot be preempted nor blocked once it starts executing. In the worst-case it will therefore execute for  $E_{i,j}$  time before completing. In order to compute the WCRT of a global segment  $\tau_{i,j}$ , we therefore simply have to sum up its release jitter, total waiting time and execution time.  $\square$

### 5.6.2 Response time of local segments

In this section we derive the WCRT of a local segment.

**Lemma 5.7.** *Under PSRP, the maximum blocking that a local segment  $\tau_{i,j}$  can experience is given by*

$$B(\tau_{i,j}) = \begin{cases} \max\{0, B^L(\tau_{i,j}), B^G(\tau_{i,j})\} & \text{if } R_{i,j} \subseteq \mathcal{R}^L, \\ B^L(\tau_{i,j}) & \text{otherwise.} \end{cases} \quad (5.16)$$

where

$$B^L(\tau_{i,j}) = \max\{E_{x,y} \mid R_{x,y} \cap \mathcal{R}^G = \emptyset \wedge \pi_x > \pi_i \wedge (\exists r \in R_{x,y} \cap R_{i,j} : \varphi(r) \leq \pi_i)\}, \quad (5.17)$$

$$B^G(\tau_{i,j}) = \max\{E'(\tau_{x,y}) \mid R_{x,y} \cap \mathcal{R}^G \neq \emptyset \wedge \pi_x > \pi_i \wedge R_{x,y} \cap R_{i,j} \neq \emptyset\}. \quad (5.18)$$

*Proof.* A local segment  $\tau_{i,j}$  can be blocked by local and global segments. Let  $B^L(\tau_{i,j})$  and  $B^G(\tau_{i,j})$  be the blocking time experienced by  $\tau_{i,j}$  due to local and global resources, respectively.

According to Definition 5.4, global segments use *only* global resources or local non-preemptive resources. Local segments therefore only compete with local segments on local preemptive resources. Access to local preemptive resources is managed using SRP. According to SRP, segment  $\tau_{i,j}$  may be blocked by a lower priority segment only once, before  $\tau_{i,j}$  starts executing. Moreover, this blocking time is equal to the

length of the longest segment among those which have a lower priority than  $\tau_{i,j}$  and share resources with  $\tau_{i,j}$ . Equation (5.17) follows.

A local segment  $\tau_{i,j}$  which requires only local resources may also be blocked by a lower priority local segment  $\tau_{x,y}$  which requires also global resources, when it spins or executes on those global resources. According to Lemma 5.3, every local segment uses exactly one local preemptive resource. The PSRP algorithm allows a segment to execute the *lock()* operation only if its priority is higher than the system ceiling of the local preemptive resource shared with  $\tau_{i,j}$ . Since the ready segments are scheduled on the preemptive resource according to their priority,  $\tau_{i,j}$  can be blocked by only one segment  $\tau_{x,y}$  and at most once. Moreover,  $\tau_{x,y}$  must have started executing before  $\tau_{i,j}$  has arrived, otherwise  $\tau_{i,j}$  would have been scheduled instead. According to Lemma 5.1, the resource holding time of segment  $\tau_{x,y}$  on each of its required resources is bounded by  $E'(\tau_{x,y})$ . Equation (5.18) follows.

Since (according to Definitions 5.3 and 5.4) exactly one preemptive resource  $p$  will be shared between all local segments sharing resources with a local segment  $\tau_{i,j}$ , access to all other (nonpreemptive) resources required by  $\tau_{i,j}$  will be synchronized by SRP on  $p$ . Segment  $\tau_{i,j}$  which requires only local resources can therefore block on either a local segment or a global segment, but not both. The first condition in (5.16) follows.

A local segment  $\tau_{i,j}$ , which requires at least one global resource, will start spinning at the highest priority as soon as it reaches the highest priority on the preemptive resource. Since the spinning time is already taken into account in  $E'(\tau_{i,j})$ , we only need to consider blocking on local segments, and can ignore blocking on global segments. The second condition in (5.16) follows.  $\square$

**Example 5.4.** Applying Lemma 5.7 to our leading example in Figure 5.1 (with segment priorities decreasing alphabetically) will result in the following blocking times for local segments:

$$B(a_1) = \max\{E'(b_1), E'(c_1)\}, B(b_1) = E'(c_1), B(c_1) = 0.$$

Notice that Lemma 5.7 ignores the fact that  $c_1$  may block on  $d_1$ , since it is taken into account in the  $E'(c_1)$  term in Theorem 5.2.  $\square$

**Theorem 5.2.** *Under PSRP, the WCRT of a local segment  $\tau_{i,j} \in \mathcal{S}^L$ , measured since the arrival of the parent task, is bounded by*

$$WCRT(\tau_{i,j}) = A(\tau_{i,j}) + w(\tau_{i,j}), \quad (5.19)$$

where  $w(\tau_{i,j})$  is the smallest value which satisfies

$$w(\tau_{i,j}) = B(\tau_{i,j}) + E'(\tau_{i,j}) + \sum_{\tau_{x,y} \in X} \left\lceil \frac{w(\tau_{i,j}) + J(\tau_{x,y})}{T_x} \right\rceil E'(\tau_{x,y}), \quad (5.20)$$

where  $J(\tau_{x,y}) = A(\tau_{x,y}) - \sum_{z < y} E_{x,z}$  is the activation jitter of segment  $\tau_{x,y}$ , and  $X = \{\tau_{x,y} \mid \pi_x < \pi_i \wedge (R_{x,y} \cap R_{i,j} \cap \mathcal{R}^L \neq \emptyset)\}$  is the set of higher priority segments which share a local resource with  $\tau_{i,j}$ .



*Proof.* As soon as a local segment  $\tau_{i,j}$  is released, it will try to lock all its required resources in  $R_{i,j}$ . If any of the resources it requires are not available, it will block for  $B(\tau_{i,j})$  given by (5.16). When  $\tau_{i,j}$  is ready to resume after the initial blocking, we distinguish between two cases, depending on whether (i)  $\tau_{i,j}$  requires only local resources, or (ii)  $\tau_{i,j}$  requires at least one global resource.

In case (i), according to Corollary 5.1, segment  $\tau_{i,j}$  will not wait inside of a resource queue, i.e.  $wait(\tau_{i,j}) = 0$ . During the time that the segment is blocked or executing, higher priority segments sharing local resources with  $\tau_{i,j}$  can arrive and interfere with it. The inter-arrival time between two consecutive invocations of a higher priority segment  $\tau_{x,y}$  is equal to its tasks period, with the first arrival suffering an activation jitter  $J(\tau_{x,y})$ , which can be bounded by the activation time of  $\tau_{x,y}$  minus the execution time of all the segments preceeding it in  $S_x$ . Equation (5.20) follows.

In case (ii), during the time  $\tau_{i,j}$  is blocked, higher priority segments may arrive. However, since  $\tau_{i,j}$  requires a global resource, as soon as it becomes ready to execute it will be inserted into the resource queue of all resources in  $R_{i,j}$  and start spinning at the highest priority on the single local preemptive resource which it requires (according to Lemma 5.3). The spinning time is included in the  $E'(\tau_{i,j})$  term in (5.20). As soon as all the resources in  $R_{i,j}$  are available, it will continue executing at the highest priority on the preemptive resource. Therefore, higher priority segments arriving during the time  $\tau_{i,j}$  is waiting or executing (i.e. “during” the  $E'(\tau_{i,j})$  term) will not interfere with  $\tau_{i,j}$ . Since in this theorem we are providing an upper bound, equation (5.20) follows.

A local segment  $\tau_{i,j}$  will be delayed (relative to the arrival of its parent task) by the WCRT of the previous segment (if any), represented by the  $A(\tau_{i,j})$  term in (5.19).  $\square$

### 5.6.3 Response time of tasks

Now that we know how to compute the WCRT of local and global segments, we can easily determine the WCRT of tasks.

**Corollary 5.2.** *Under PSRP, the WCRT of a task  $\tau_i \in \Gamma$  is given by the WCRT of the last segment in  $R_i$ .*

Note that the WCRT of a segment depends on the activation time of another segment. In turn, the activation time of a segment depends on the WCRT of another segment. However, since the priority of all segments of a given task is the same, this mutual dependency problem can be solved by simply computing response and activation times in order from the highest priority task to the lowest priority task.

**Example 5.5.** Earlier in this chapter we have observed that a common approach for scheduling tasks on a platform comprised of multiple heterogeneous resources is to treat the whole platform as a whole, allowing at most one task to use the platform at a time. In this example we compare this approach to PSRP and show that PSRP can indeed exploit the concurrency available on a multi-resource platform by verifying for an example task set with a utilization greater than 1 all deadlines are met.

Consider our leading example platform from Figure 5.1, comprised of three processors  $\mathcal{P} = \{p_1, p_2, p_3\}$ , three logical resources  $\mathcal{N} = \{n_1, n_2, n_3, n_4\}$ , executing an application consisting of tasks  $\Gamma = \{a, b, c, d, e, f\}$  and segments  $\mathcal{S} = \{a_1, b_1, c_1, c_2, d_1, d_2, e_1, f_1\}$ , specified in Figure 5.11. The corresponding partial segment requirements graphs are shown in Figure 5.7 with the derived dependencies for global segments in Figure 5.8.

$\tau_i$	$\pi_i$	$O_i$	$T_i$	$D_i$	$S_i$	$\tau_{i,j}$	$E_{i,j}$	$R_{i,j}$
$a$	1	0	14	14	$\langle a_1 \rangle$	$a_1$	2	$\{p_1\}$
$b$	2	0	14	14	$\langle b_1 \rangle$	$b_1$	2	$\{p_1, n_1\}$
$c$	3	0	14	14	$\langle c_1, c_2 \rangle$	$c_1$	3	$\{p_1, n_1, n_2\}$
$d$	4	0	14	14	$\langle d_1, d_2 \rangle$	$c_2$	1	$\{p_2, p_3, n_2\}$
$e$	5	0	14	14	$\langle e_1 \rangle$	$d_1$	1	$\{p_2, p_3, n_2, n_3\}$
$f$	6	0	14	14	$\langle f_1 \rangle$	$d_2$	8	$\{n_3\}$
						$e_1$	1	$\{p_3, n_4\}$
						$f_1$	1	$\{p_3, n_4\}$

Figure 5.11: Task and segment specifications.

We use Lemmas 5.1 and 5.2 to compute the worst case response times of all segments in  $\mathcal{S}$ , and hence of all the tasks in  $\Gamma$ .

Segment  $a_1$  will experience local blocking due to  $b_1$  and global blocking due to  $c_1$ ,

$$\begin{aligned} B^L(a_1) &= E'(b_1) = \text{wait}(b_1) + E_{b_1} \\ B^G(a_1) &= E'(c_1) = \text{wait}(c_1) + E_{c_1} \\ B(a_1) &= \max\{B^L(a_1), B^G(a_1)\} = \max\{E'(b_1), E'(c_1)\} \end{aligned}$$

Since  $b_1$  requires only local resources, according to Corollary 5.1,  $\text{wait}(b_1) = 0$ . We can compute  $\text{wait}(c_1)$  using Lemma 5.6. For each partial segment requirements graphs derived from  $G$  we need to compute the set of segments which  $c_1$  depends on and sum up their execution times.  $\text{wait}(c_1)$  is then equal to the maximum of these sums. According to Figure 5.8, the only dependency sets for  $c_1$  among all partial segment requirements graphs segment  $c_1$  is  $\{d_1, e_1, f_1\}$ . Therefore,

$$\text{wait}(c_1) = E_{d_1} + E_{e_1} + E_{f_1}.$$

and

$$\begin{aligned} E'(b_1) &= 0 + 2 = 2 \\ E'(c_1) &= 1 + 1 + 1 + 3 = 6. \end{aligned}$$

The worst-case response time of segment  $a_1$  is given, according to Lemma 5.2, by

$$WCRT(a_1) = A(a_1) + w(a_1)$$

with

$$w(a_1) = B(a_1) + E'(a_1) + \sum_{\tau_{x,y} \mid \pi_x < \pi_a \wedge (R_{x,y} \cap R_{a_1} \cap \mathcal{R}^L \neq \emptyset)} \left\lceil \frac{w(a_1) + J(a_1)}{T_x} \right\rceil E'(\tau_{x,y})$$

Since  $a_1$  is the first segment in the sequence  $R_a$ , its activation time  $A(a_1) = 0$ , and it will suffer no jitter, i.e.  $J(a_1) = 0$ . Since it requires only local resources, according to Corollary 5.1,  $wait(a_1) = 0$ . Since  $a_1$  belongs to the highest priority task, it will not be preempted while executing. Therefore,

$$WCRT(a_1) = \max\{E'(b_1), E'(c_1)\} + E_{a_1} = \max\{2, 6\} + 2 = 8.$$

Similarly, we can compute the worst-case response time for local segment  $b_1$

$$\begin{aligned} B^L(b_1) &= 0 \\ B^G(b_1) &= E'(c_1) \\ B(b_1) &= E'(c_1) \\ WCRT(b_1) &= E'(c_1) + E_{b_1} + E_{a_1} = 6 + 2 + 2 = 10 \end{aligned}$$

Since the local segment  $c_1$  requires a global resource  $n_2$ , according to (5.16) its blocking time is bounded by  $B^L(c_1)$ . Since  $c_1$  has a lower priority than  $a_1$  and  $b_1$ , it can be preempted by both of them. Therefore,

$$\begin{aligned} B^L(c_1) &= 0 \\ B(c_1) &= 0 \\ WCRT(c_1) &= E'(c_1) + E_{a_1} + E_{b_1} = 6 + 2 + 2 = 10 \end{aligned}$$

Segment  $c_2$  is a global segment. Its worst-case response time is therefore given by Lemma 5.1. According to Figure 5.8, the dependency sets for  $c_2$  are  $\{d_1, e_1, f_1\}$  and  $\{e_1, f_1\}$ . Therefore,

$$\begin{aligned} wait(c_2) &= \max\{E_{d_1} + E_{e_1} + E_{f_1}, E_{e_1} + E_{f_1}\} = 1 + 1 + 1 = 3 \\ E'(c_2) &= wait(c_2) + E_{c_2} = 3 + 1 = 4 \\ WCRT(c_2) &= WCRT(c_1) + E'(c_2) = 10 + 4 = 14 \end{aligned}$$

Segment  $d_1$  is a global segment. According to the dependencies in Figure 5.8 we get

$$\begin{aligned} wait(d_1) &= \max\{E_{c_1} + E_{e_1} + E_{f_1}, E_{c_2} + E_{e_1} + E_{f_1}\} \\ &= \max\{3 + 1 + 1, 1 + 1 + 1\} = 5 \\ WCRT(d_1) &= wait(d_1) + E_{d_1} = 5 + 1 = 6 \end{aligned}$$

Segment  $d_2$  is a global segment. According to Figure 5.8 it does not depend on any other segments. Therefore,

$$\begin{aligned} wait(d_2) &= 0 \\ WCRT(d_2) &= WCRT(d_1) + wait(d_2) + E_{d_2} = 6 + 8 = 14 \end{aligned}$$

Similarly, we can compute the worst-case response times of segments  $e_1$  and  $f_1$ ,

$$\begin{aligned} WCRT(e_1) &= 6 \\ WCRT(f_1) &= 6 \end{aligned}$$

According to Corollary 5.2, the worst-case response times of tasks  $a, b, c, d, e, f$  are 8, 10, 14, 14, 6, 6, respectively. Since the worst-case response time of each task is

smaller or equal to its deadline, the system is schedulable. The utilization of the system is

$$U(a) + U(b) + U(c) + U(d) + U(e) + U(f) = \frac{2}{14} + \frac{2}{14} + \frac{3+1}{14} + \frac{1+8}{14} + \frac{1}{14} + \frac{1}{14} = \frac{19}{14}$$

which is greater than 1, so we are exploiting some of the available concurrency.  $\square$

## 5.7 Evaluation

In this section we demonstrate the effectiveness of PSRP in exploiting the inherent parallelism of a platform comprised of multiple heterogeneous resources. We consider a task set where some task segments require several processors at the same time and also share nonpreemptive resources. We schedule it using two approaches: (i) using PSRP, and (ii) by collapsing all the processors into one virtual processor and applying uniprocessor scheduling (which we refer to as "Collapsed"). To the best of our knowledge, the second approach is currently the best alternative to PSRP for scheduling parallel tasks which can execute on arbitrary subsets of processors and share nonpreemptive resources. We compute the WCRT of the complete task for the two approaches. The difference in response times represents potential utilization gain, which can be exploited by e.g. background tasks or tighter timing requirements.

The simulated task set  $\Gamma$  represents a multimedia application, where video frames are captured periodically with period  $T$  and subsequently processed by a set of filters. Some of the filters are computationally intensive, but can exploit functional parallelism and execute on several processors in parallel. The video frames are stored in a shared global memory. Each parallel filter loads the necessary frame data from the global memory into its local buffer, operates on it, and writes the result back to the global memory. The data is transferred using a DMA controller. The simulated platform corresponds to a PC with a multicore processor. For simplicity we assume no caches.

*PSRP approach:* The platform consists of  $M$  processors  $p_j$ ,  $1 \leq j \leq M$ , a global memory  $m$ , local memories accessible by individual processors (or groups of processors) where  $m_i$  represents the memory region in a local memory allocated to task  $\tau_i$ , and a DMA controller  $dma$  for transferring data between the global and local memories. It can be expressed in terms of our model as  $\mathcal{P} = \{p_1, p_2, \dots, p_M\}$  and  $\mathcal{N} = \{dma, m, m_1, m_2, \dots, m_{|\Gamma|}\}$ .

We consider several scenarios. In each scenario we divide the processors into  $H$  groups  $\mathcal{P}_g$ ,  $1 \leq g \leq H$ . Each group contains  $W$  processors, with  $H * W = M$ . On each group of processors we execute a set of  $K$  parallel tasks. Each parallel task  $\tau_i$  belonging to group  $g$  is specified by  $S(\tau_i) = \langle (0.5, \{dma, m, m_i\}), (5, \mathcal{P}_g \cup \{m_i\}), (0.5, \{dma, m, m_i\}) \rangle$ . On each processor  $p_j \in \mathcal{P}$  we also execute a sequential task  $\tau_i$  with  $S(\tau_i) = \langle (2, \{p_j, m_i\}) \rangle$ . All tasks share the same period  $T$ ,  $\forall \tau_i \in \Gamma$ :  $D_i = T_i$ , and the parallel tasks have higher priority than the sequential tasks.

*Collapsed approach:* We can model the Collapsed approach by replacing all processors by one preemptive resource  $p$  and having each segment require at least the resource  $p$ . For a scenario with  $H$ ,  $W$  and  $K$  defined

above, the “collapsed” task set then consists of  $H * K$  tasks  $\tau_i$  with  $S(\tau_i) = \langle (0.5, \{p, dma, m, m_i\}), (5, \{p, m_i\}), (0.5, \{p, dma, m, m_i\}) \rangle$ , and  $H * W$  tasks  $\tau_i$  with  $S(\tau_i) = \langle (2, \{p\}) \rangle$ .

Figure 5.12 compares the maximum WCRT among all tasks in  $\Gamma$  between the PSRP and Collapsed approaches for  $H = 2$ . We vary the number of tasks per processor group  $K$  and the number of processors required by parallel tasks  $W$ . We have computed the WCRT for the PSRP approach using the analysis presented in this chapter, and for the Collapsed approach using the Fixed Priority Preemptive Scheduling analysis Audsley et al. (1993).

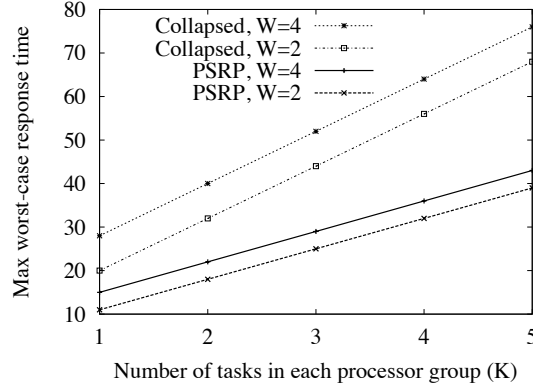


Figure 5.12: Comparison of WCRTs for cases (i) and (ii) for  $H = 2$  and varying  $W$  and  $K$ .

The results show that PSRP experiences lower WCRT than the Collapsed approach. Moreover, since the difference in WCRT increases for larger values of  $K$  and  $W$ , the benefits of PSRP increase with larger task sets and more parallelism, i.e. when tasks execute on more processors in parallel. The results therefore demonstrate that PSRP indeed outperforms the Collapsed approach.

### 5.7.1 Time complexity of the analysis

The worst-case response time of all tasks can be computed using the following procedure:

1. Construct the segment requirements graph  $G$ .
2. Compute the partial segment requirements graphs  $partial(G)$ .
3. For each partial segment requirements graph compute its connected tasks.
4. For each global segment  $\tau_{i,j} \in \mathcal{S}^G$  compute its waiting time  $wait(\tau_{i,j})$ .

5. For each task  $\tau_i \in \Gamma$ , starting with the highest priority one, traverse its required segments sequence  $R_i$ , starting with the first one, and compute its worst-case response time.

**Lemma 5.8.** *The worst-case response time of all tasks in  $\Gamma$  can be computed in exponential time.*

*Proof.* The time complexity is given by the sum of the complexities of the individual steps of the above procedure.

**Step 1:** A segment requirements graph  $G = (V, E)$  is a tripartite graph which contains  $|V| = |\mathcal{R}| + |\mathcal{S}|$  vertices and  $|E| = |\mathcal{I}|$  edges, where  $\mathcal{I} = \bigcup_{\tau_{i,j} \in \mathcal{S}} R_{i,j}$  is the set of all segment resource requirements. It can be constructed in  $O(|\mathcal{R}| + |\mathcal{S}| + |\mathcal{I}|)$  time.

**Step 2:** According to Lemma 5.5, the set of all partial segment requirements graphs  $partial(G)$  contains  $|partial(G)| \leq \prod_{\tau_i \in \Gamma} |S_i|$  graphs. If we define  $S_{max}$  as the longest sequence of segments among all tasks, then  $|partial(G)| \leq |S_{max}|^{|\Gamma|}$ . Each partial graph can be computed by removing segment-nodes and connecting edges from the original graph  $G$ , which can be done in  $O(|\mathcal{R}| + |\mathcal{S}| + |\mathcal{I}|)$  time. This entire step will therefore take  $O(|S_{max}|^{|\Gamma|} * (|\mathcal{R}| + |\mathcal{S}| + |\mathcal{I}|))$  time.

**Step 3:** Each partial graph can be decomposed into connected subgraphs by performing a breadth first search in  $O(|\mathcal{R}| + |\mathcal{S}| + |\mathcal{I}|)$  time. This entire step will therefore take  $O(|S_{max}|^{|\Gamma|} * (|\mathcal{R}| + |\mathcal{S}| + |\mathcal{I}|))$  time.

**Step 4:** By precomputing the sum of the execution times of each connected subgraph in all partial segment requirements graphs (which can be done while constructing the connected subgraphs in Step 3), we can compute the  $\sum_{\tau_{x,y} \in \delta(\tau_{i,j}, g)} E_{x,y}$  term in (5.13) in  $O(1)$  time, by subtracting its execution time from the sum for each partial segment requirements graph. This has to be repeated for each partial segment requirements graph to compute the max term in (5.13). This entire step will therefore take  $O(|S_{max}|^{|\Gamma|})$  time.

**Step 5:** The worst-case response time of a local segment in (5.19) relies on solving the recursive equation (5.20), which (according to Lehoczky et al. (1989)) can be done for all local segments in pseudo-polynomial time.

Steps 1-4 show that the worst-case response time for all global segments can be computed in exponential time, while step 5 shows that for all local segments it can be done in pseudo-polynomial time. Therefore, using Corollary 5.2, the worst-case response time for all tasks can be computed in exponential time.  $\square$

## 5.8 Discussion

In this chapter we addressed the problem of multi-resource scheduling of parallel tasks with real-time constraints. We proposed a new resource model, which classifies different resources (such as bus, processor, shared variable, etc.) as either a preemptive

or non-preemptive multi-unit resource. We then presented a new scheduling algorithm called PSRP and the corresponding schedulability analysis. Simulation results based on an example application show that it can exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources.

In the remainder of this section we discuss the pros and cons of the proposed system model and PSRP.

### 5.8.1 Multi-unit preemptive resources

We can model a homogenous multiprocessor containing  $n$  cores in two ways: as a preemptive resource  $p$  with  $N_p = n$ , or as  $n$  preemptive resources  $p_1, p_2, \dots, p_n$ , with capacities  $N_{p_i} = 1$ , for all  $1 \leq i \leq n$ . Existing literature on parallel-task scheduling on multiprocessors assumes the first option, where each task segment specifies a requirement for a number of units of a multi-unit resource  $p$ . The system is then responsible for allocating tasks to processors during runtime. This model will ignore potentially large migration overheads, e.g. in memory intensive applications as data locality cannot be guaranteed. Using the second approach, our model allows to partition the task set upfront, e.g. optimizing data locality.

### 5.8.2 Nonpreemptive execution on preemptive resources

When a preemptive resource  $p$  is required by a segment which requires also another preemptive resource, then  $p$  is marked as a global resource, resulting in non-preemptive execution on  $p$ . This may appear overly pessimistic, especially compared to the work by Kato and Ishikawa (2009) who describe a preemptive gang scheduling algorithm. However, they assume independent tasks. In multiprocessor scheduling with shared resources it is critical to keep the holding time of global non-preemptive resources as short as possible (which is the rationale between the spin-lock based approach to locking global resources in MSRP). If we were to schedule all preemptive resources preemptively, then segments requiring several preemptive resources in a chained fashion (illustrated in Figure 5.5) would increase the resource holding time. We therefore decided to limit preemptive execution to preemptive resources which are required by segments which do not require any other preemptive resource.

### 5.8.3 Pessimistic analysis for local segments

Theorem 5.2 describes the WCRT of a local segment. It treats all local segments alike, whether they require global resources or not. However, only a local segment which requires *only* local resources can be preempted by higher priority segments while it is executing. A segment which requires at least one global resource will be scheduled non-preemptively on all preemptive resources it requires. We can therefore lower the bound on WCRT of local segments which require global resources by ignoring the interference of higher priority tasks during the execution of those segments. For this purpose we can adopt the schedulability analysis for Fixed-Priority with Deferred Preemption Scheduling by Bril et al. (2007).

#### 5.8.4 Pessimistic bound for the waiting time

Lemma 5.6 presents a bound for the waiting time. This bound, however, is pessimistic, as illustrated by the following example.

**Example 5.6.** Figure 5.13 illustrates the source of the pessimism for  $\text{wait}(b)$ . According to PSRP, segment  $b$  may be delayed by  $a$  or  $c$ , but not both. Lemma 5.6, however, assumes that in the worst-case  $b$  will have to wait for both  $a$  and  $c$ , which is pessimistic in case  $a$  and  $c$  do not share a common resource.

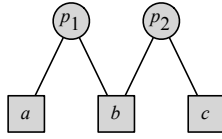


Figure 5.13: A segment requirements graph for a system comprised of  $\mathcal{P} = \{p_1, p_2\}$ ,  $\mathcal{N} = \emptyset$ ,  $\mathcal{S} = \{a, b, c\}$  with  $R_a = \{p_1\}$ ,  $R_b = \{p_1, p_2\}$ , and  $R_c = \{p_2\}$ .

□

The pessimism stems from the definition of dependency. According to Definition 5.8, a segment depends on another segment iff there exists a partial segment requirements graph  $g$ , such that the two segments belong to the same connected subgraph of  $g$ . Hence, segment  $b$  in Figure 5.13 will have to wait for both  $a$  and  $c$ , since all of them belong to the same subgraph.

#### 5.8.5 Nested global critical sections

FMLP supports nested critical sections by means of resource groups. The resource groups partition the set of resources into independent subsets. Consequently, a task trying to access resource  $r$  may become blocked on *all* resources in the resource group  $G(r)$ .

Under PSRP, if a task has nested critical sections we can move the inner critical sections outwards until they overlap exactly with the outer most critical section. This new task can be expressed in our system model, where segments require several resources at the same time. Each segment can be blocked only on resources which it requires (rather than the complete resource group). PSRP therefore provides a more flexible approach for dealing with nested global critical sections than FMLP. In the worst case, under FMLP a segment requiring resource  $r$  will be blocked by every task for the duration of the longest segment which requires a resource from  $G(r)$ , while under PSRP a segment will be indirectly blocked by all dependent segments (see Figure 5.5).

Under FMLP every time a job is resumed it may block on a local resource. This is the same for PSRP, where we have to include the blocking time for each segment (rather than once per task).



## Chapter 6

# Grasp

Modern real-time systems are becoming increasingly more complex, with many tasks executing concurrently on many processors, making it difficult to understand the system behavior. A popular trend in coping with the vast number of tasks and the resulting interferences between them is to hide tasks inside components and to integrate the system from those components. This approach requires hierarchical scheduling, which has been covered extensively in the literature for uniprocessor systems. Recently, the real-time literature has been investigating applying hierarchical scheduling to multiprocessor platforms. In this chapter we address the problem of how to provide insight into complex interaction patterns between jobs executing in a hierarchical multiprocessor system.

Several approaches are available for tackling the complexity of modern software systems. Ideally, every system would be meticulously documented, providing a formal yet concise description of the emergent system behavior. However, this is a long and costly process without immediate effects (such as additional functionality) and is therefore not common in practice. Examples of poorly documented code and system designs are abundant. The description of the dynamic system behavior therefore needs to be extracted from existing systems. There are modeling and verification tools available, which rely on the developers analyzing the implementation and constructing its model. These tools then employ formal methods to verify the behavior of the extracted model against an abstract model. The state of the art modeling and verification techniques, however, are not scalable and therefore can be applied to verify only a small portion of the entire system.

Visualization tools offer an interesting alternative. Existing systems can be instrumented to generate runtime traces, which can then be analyzed by engineers and researchers, leveraging their expertise and human capacity to recognize patterns, to gain insight into the system behavior. The challenge here lies in presenting the information in an intuitive way, enabling the user to extract the essential properties of the analyzed system.

Grasp is a toolset for tracing and visualizing the behavior of complex real-time systems. Its main strength lies in providing many different visualizations for various

real-time primitives and scheduling techniques in a consistent and intuitive way. Its flexible architecture allows to easily extend it with new visualization and analysis techniques.

We have been using Grasp extensively within our group during our research of embedded real-time systems and the development of various extensions of a commercial real-time operating system  $\mu C/OS-II$ , including a hierarchical scheduling framework and slot shifting. The usage of Grasp has also been reported in (Åsberg et al., 2010, 2011; van den Heuvel et al., 2010) where it was used to gain insights into new approaches for hierarchical scheduling in Linux and VxWorks operating systems. Recently Grasp was used in the context of the Smart Objects For Intelligent Applications (SOFIA) European research project to visualize the communication patterns between sensor nodes in a smart home environment.

In this chapter we focus on tracing multiprocessor systems. The challenge here lies in presenting the execution and communication between jobs running on different processors in an intuitive and compact way. Moreover, if the timestamps of events occurring on a processor are recorded in its local time, special care must be taken to synchronize the individual traces. While some custom-built clusters such as IBM Blue Gene offer sufficiently accurate global clocks, most distributed systems can only rely on local clocks (Becker et al., 2011). If the local clocks drift too far apart it may lead to inaccuracies or even errors during the trace analysis or visualization, as the causality between events may appear to be broken (e.g. messages arriving before they were sent).

The Grasp toolset together with example traces is available for Linux, Mac and Windows at <http://www.win.tue.nl/~mholende/grasp>.

## Contributions

In this chapter we focus on visualizing the timing of job execution and communication in the context of multiprocessor systems. In particular, we demonstrate Grasp's ability to visualize

- partitioned and global multiprocessor scheduling,
- migrating tasks and jobs,
- communication between jobs via shared memory and message passing,
- hierarchical scheduling in combination with multiprocessor scheduling,
- time synchronization between traces from different processors.

We also describe Grasp's interface for synchronizing timestamps in traces generated in a distributed system.

## Publications

The ability of Grasp to visualize hierarchical scheduling was presented in (Holenderski et al., 2010c). The visualization of multiprocessor systems was introduced

in (Holenderski et al., 2011a), with the time synchronization between traces from different processors described in (Holenderski et al., 2012b).

## 6.1 Related work

Existing visualization tools for real-time systems are specialized to visualize a fixed set of behaviors. For example, the Tracealyzer Mughal and Javed (2008) and TimeDoctor TimeDoctor (2011) are targeting only non-hierarchical uniprocessor systems. Making a step towards distributed systems is not trivial. Grasp, on the other hand, supports multiprocessor systems with two level virtualization.

There are several trace visualization tools which support the development of parallel programs on uniform parallel-processor platforms, such as VAMPIR Nagel et al. (1996), Paje Kergommeaux et al. (2000), Jedale Hunold et al. (2010), or Scalasca Geimer et al. (2010). They illustrate the execution of parallel jobs and communication between them, but they are limited to flat systems. To the best of our knowledge no visualization tools currently support the visualization of hierarchical scheduling in a uniprocessor or multiprocessor setting.

Traces accepted by most tools are lists of timed events, often in a binary format. Grasp, on the other hand, has adopted the idea of treating the trace as a script. On the one hand, Grasp traces are more verbose and require more storage space, compared to the binary format. On the other hand, they allow for large degree of flexibility, making it easy to add new events to the event model without changing the core implementation of the visualization and analysis components. This makes Grasp ideal for rapid prototyping of new visualization techniques. We have exploited this flexibility during the development of Grasp's various visualization and analysis features.

There are several tracing tools available, mainly for the Linux platform, which generate traces. Examples include the Data Stream Kernel Interface (DSKI) Buchanan et al. (1998), Ftrace Ftrace (2008), and Dtrace Dtrace (2008). DSKI is a platform independent interface standard to support collection of a variety of performance data from the operating system internals. It has been implemented on Linux. Ftrace and Dtrace are integrated in many Linux distributions. They exhibit low performance overhead and low memory footprint. In order to leverage their popularity, we have implemented a converter from the sched\_switch tracer output of Ftrace, allowing to use Grasp in many Linux and Unix environments.

When tracing is used to analyze the behavior of distributed systems, then there is the additional problem of synchronizing the times of events occurring on different nodes. Time differences among distributed clocks can be characterized in terms of their relative offset and drift. If we assume constant drift, then the local time can be mapped onto the global (or master) time via linear interpolation. Existing time synchronization algorithms compute the interpolation based on the timestamps of messages exchanged back and forth with a master node Cristia (1989); Maillet and Tron (1995) or with other nodes Mills (1992); Biberstein et al. (2008); Becker et al. (2009). The timestamp synchronization in Grasp is based on Maillet and Tron (1995).

The authors of Becker et al. (2009, 2011) observe that while linear offset interpolation might prove satisfactory for short runs, measurement errors and time-dependent drifts may create inaccuracies and violate causality relations during longer runs (e.g. a message is received before it was sent). They propose a method for fixing these errors by postponing certain events in the trace. However, while maintaining the causality relations in traces, their methods change the timing of the events. As Grasp is targeting real-time systems, it relies on linear interpolation for synchronizing traces and if it detects inconsistencies in the ordering of events then it notifies the user that the constant drift assumption was violated.

## 6.2 Grasp overview

The Grasp toolset is composed of three entities: the Grasp Recorder, the Grasp Trace and the Grasp Player, as shown in Figure 6.1.

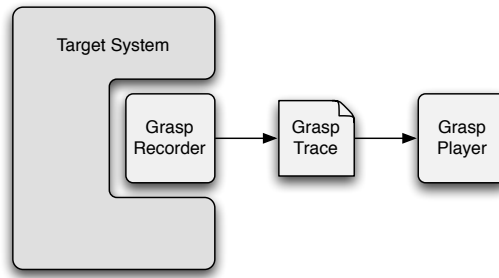


Figure 6.1: Overview of the Grasp architecture.

The Grasp Recorder is embedded in the target system and is responsible for generating a trace. The generated Grasp Trace contains the raw data from a particular system run. The Grasp Player reads in a trace and displays it in an intuitive way.

### 6.2.1 Grasp Recorder

The Grasp Recorder is implemented as a library providing functions to initialize the recorder, log events, and finalize the recorder. Calls to the event logging methods are inserted at several places inside the kernel to log common events, such as context switches, arrival of tasks, or server replenishment. The recorder also provides a function to log custom events, which programmers may call inside their applications.

Designing and implementing an instrumentation infrastructure which exhibits low performance and memory overheads can be a daunting task. Therefore, rather than designing a custom Grasp Recorder and integrating it within the target system, one can implement a converter for existing trace format, leveraging existing instrumentation and tracing tools, as we have done for the sched\_switch tracer output of Ftrace (Ftrace, 2008).

### 6.2.2 Grasp Trace

The Grasp Trace is a Tcl (Welch et al., 2003) script. The decision for treating the Grasp Trace as a script results in large degree of flexibility. The Grasp Player basically provides a set of commands which can be called from within a Grasp Trace. A trace can therefore be a simple list of commands, but it can also be a complete system simulator, or anything in between. This allows to embed various extensions (or plugins) inside a trace, resulting in a self-contained trace which can be visualized by any Grasp Player, independent of the plugins it provides. It can also be used to reduce the size of very large traces, by automatically generating or factoring out common or repeating parts. Also, a trace may call methods in the player's public API to override its default settings, making sure that the trace is visualized as intended by its creator. The greatest benefit of the trace being a script, however, is the simple plugin infrastructure discussed in the next section.

A typical Grasp Trace event has the following structure:

**plot** *time event arguments*

which means that *event* has occurred at time *time*. The *arguments* parameter is a list and describes the instance of the *event*. Every *event* defines its own signature, i.e. the number and the semantics of the *arguments* which it accepts. Usually an event accepts a list of required arguments followed by a list of *-key value* pairs for optional arguments. In the remainder of this chapter we will often ignore the **plot time** part, as it is common for most events. Also, we will ignore optional arguments for customizing the trace visualization, such as assigning names or colors to tasks.

There are several basic events for tracing job execution:

- **newTask** *task* creates a new task, where *task* is a new identifier used in later events.
- **jobArrived** *job task* indicates that *job* belonging to *task* has arrived, where *job* is a new identifier used in later events, and *task* is the identifier of a task created previously with **newTask**.
- **jobStarted** *job* indicates that *job* has started.
- **jobPreempted** *job* indicates that *job* has been preempted.
- **jobBlocked** *job* indicates that *job* has been blocked (e.g. trying to access a locked shared resource).
- **jobResumed** *job* indicates that *job* has been resumed.
- **jobCompleted** *job* indicates that *job* has completed.

An example trace is shown in Figure 6.2.

```

newTask task1 -priority 7 -name "Task 1"
newTask task2 -priority 8 -name "Task 2"
plot 5 jobArrived job2.1 task2
plot 5 jobResumed job2.1
plot 20 jobArrived job1.1 task1
plot 20 jobPreempted job2.1 -target job1.1
plot 20 jobResumed job1.1
plot 35 jobCompleted job1.1 -target job2.1
plot 35 jobResumed job2.1
plot 50 jobCompleted job2.1

```

Figure 6.2: Example of a Grasp Trace.

### 6.2.3 Grasp Player

The Grasp Player is the main contribution of Grasp. It basically provides an execution environment for the script inside of a Grasp Trace. As the Grasp Player is also written in Tcl, its operation is very simple: it loads the definitions of all methods which can be called inside a trace, and then evaluates the trace script. Figure 6.3 shows an example of a trace of a video processing algorithm. The visualization correlates the contents of the frame buffers with the system execution, allowing to inspect their content at different times in relation to the dynamic events occurring during runtime.

The Grasp Player comes with a powerful set of features, including the visualization of task execution in flat and hierarchical systems, uni- and multiprocessor scheduling, intervals in slot shifting, measurement of execution and response times, automatic verification of certain trace properties, command line interface, and exporting to postscript (useful for creating high quality figures for research articles, e.g. Figures 6.5, 6.6, 6.7, and 6.10).

### Plugins

The Grasp Player provides a simple yet versatile infrastructure for extending it with custom visualization and analysis plugins. For example, the Grasp Recorder extension and Grasp Player visualization plugin for intervals in slot shifting was implemented by a student within two hours, extending the budget visualization for servers in hierarchical scheduling.

A plugin has three interfaces at its disposal:

(i) A plugin can define and implement its own methods which can be called within a trace. The Buffer visualization in Figure 6.3 is an example of such a plugin. It defines methods for tracing the content of buffers via events for adding and removing messages from a buffer:

- **newBuffer** *buffer* creates a new buffer, where *buffer* is a new identifier used in later events.

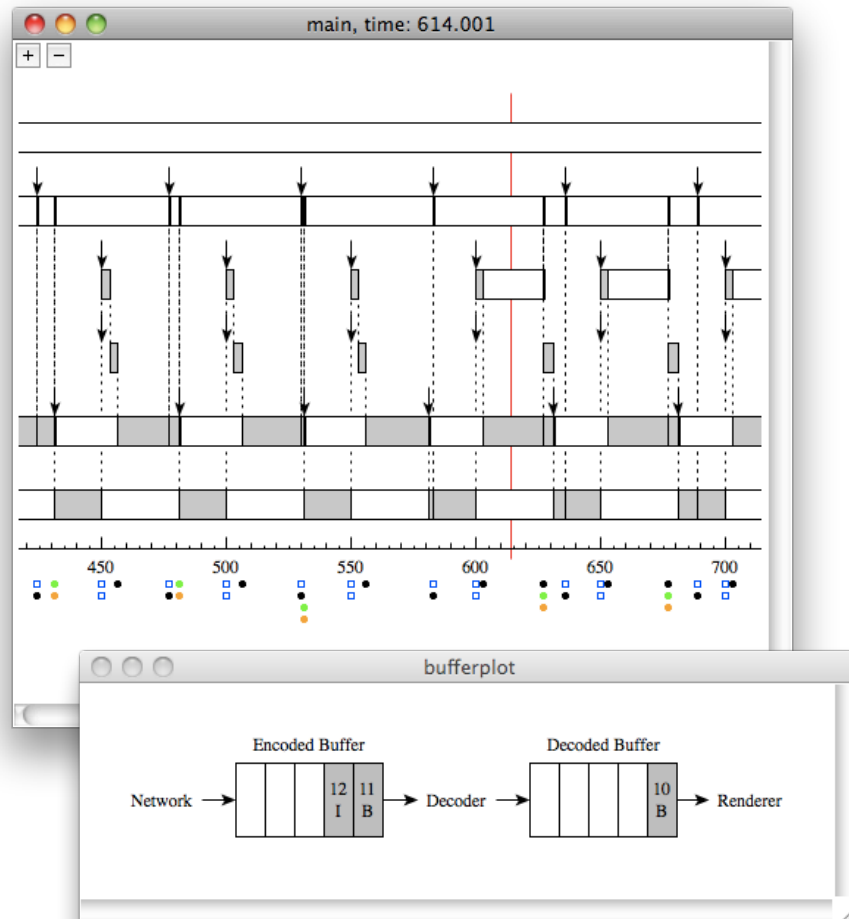


Figure 6.3: Example illustrating a video processing application comprised of several tasks (including Network, Decoder and Renderer tasks) executing on a single processor and communicating individual frames of an MPEG video via two shared buffers. As the mouse cursor moves across the trace, the contents of the buffers changes. The figure shows the contents of the buffers at time 614, including the sequence number and the kind of the video frames.

- **bufferplot** *time* **write** *buffer* *message* indicates that *message* was added at *buffer*'s tail at time *time*.
- **bufferplot** *time* **read** *buffer* indicates that a message was removed from the *buffer*'s head at time *time*.

(ii) Alternatively, a plugin can register handlers for a set of virtual events, which are generated when the traced events are processed. The Grasp Player provides a method allowing a plugin to register a script which will be evaluated whenever a particular event occurs. For example, the Measurement plugin registers a handler for the **jobArrived** and **jobCompleted** events, to compute the response time of jobs.

(iii) The Grasp Player also provides a set of player events. For example, a plugin can register a script which will be called upon the **TimeChanged** event, which is generated when the mouse cursor is moved across the trace. This player event is used by the Buffer plugin to illustrate the buffer content at the time pointed to by the mouse cursor (e.g in Figure 6.3).

The simple plugin infrastructure is made possible by the Grasp Trace being a script. Other visualization tools rely on a “dispatch” method which is called for each event in the trace to dispatch the corresponding event handler. Extending such tools with new events requires to modify the dispatch method (or to limit the syntax of traced events). As the Grasp Trace simply calls methods provided by the Grasp Player, there is no need for a dispatch method. Extending the Grasp Player with a plugin requires simply to place the plugin script inside of the plugins directory (which is automatically included when the player starts).

### Automatic verification

The plugin infrastructure can be leveraged to implement various verification tools for automatically analyzing the system behavior in a trace. For example, the BudgetCheck plugin shows a warning when a server exceeds its budget, and the MutexCheck plugin verifies proper nesting of mutex locking events inside a trace.

For any given target system, if a particular behavior is expected, then a “test-suite” plugin may be implemented to verify that for a specific scenario the target system satisfies the desired properties, e.g. after a maintenance activity.

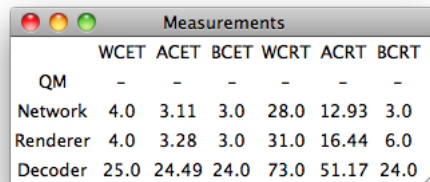
### Measurements

The Grasp player measures the execution and response time of jobs and provides a summary of the average, best case and worst case execution and response times for all jobs of a task. This information is shown on demand, by clicking on a job or a task label, or by selecting “Measurements” from the menu, shown in Figure 6.4.

The Grasp plugins also allow to easily implement custom measurement tools, as we did for measuring the mode change latencies for the simulations in Chapter 4. To measure the mode change latencies we have added a simple plugin with three new events:

- **latencyStart** starts a latency measurement.
- **latencyStop** stops a previously started latency measurement.
- **latencySummary** collects all the measured latencies, uses the *gnuplot* tool (Gnuplot, 2010) to plot them on a graph, and automatically writes the graph to a postscript file.





	WCET	ACET	BCET	WCRT	ACRT	BCRT
QM	-	-	-	-	-	-
Network	4.0	3.11	3.0	28.0	12.93	3.0
Renderer	4.0	3.28	3.0	31.0	16.44	6.0
Decoder	25.0	24.49	24.0	73.0	51.17	24.0

Figure 6.4: Example of trace measurements, summarizing the worst-case (WCET), average-case (ACET) and best-case (BCET) execution times, and the worst-case (WCRT), average-case (ACRT) and best-case (BCRT) response times for all application tasks.

The first two events are generated during the simulation whenever a mode change occurs. The latter event is generated at the end of the simulation.

## 6.3 Multiprocessor scheduling

In this section we present Grasp's support for multiprocessor systems. The multiprocessor support is implemented by extending a subset of events for tracing job execution with an optional **-processor** argument.

Our goal was to support various concepts commonly found in multiprocessor scheduling. Grasp supports partitioned as well as global multiprocessor scheduling with task and job migration, and communication between jobs on shared and distributed memory platforms. In this section we discuss each of these features in more detail.

### 6.3.1 Creating a processor

Similar to other objects in a trace, such as tasks or servers, a processor needs to be created before it can be referred to in other trace events.

- **newProcessor** *processor* creates a new processor, where *processor* is an identifier which can be added to other trace events to support multiprocessor visualization.

### 6.3.2 Partitioned and global scheduling

In partitioned scheduling, each task is assigned to a particular processor and during runtime all of its jobs execute on that processor. In global scheduling, different jobs of the same task may execute on different processors.

### Partitioned scheduling

When a task is created, it can be assigned to a particular processor:

- **newTask** *task* **-processor** *processor* creates a new *task* and assigns it to the *processor*.

All subsequent job events will be mapped to the *processor* (unless the processor argument is overridden, as discussed in the next section). Figure 6.5 shows an example of a trace on partitioned multiprocessor platform.

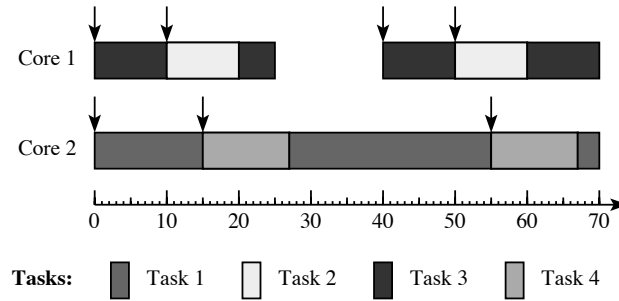


Figure 6.5: Example illustrating the execution of five tasks on a partitioned multiprocessor platform consisting of two cores.

Figure 6.5 shows the system behavior in a collapsed view, where the execution of all tasks is collapsed on a single timeline. Alternatively, the Grasp Player also supports an expanded view, where each processor is shown in a separate window illustrating the interactions between the local tasks, as shown for a single processor in Figure 6.3.

Note that the Grasp Player provides many details upon a mouse click. For example, when the mouse is clicked on top of a downward pointing arrow, a message is shown telling which task has arrived. These features are difficult to visualize on paper.

### Global scheduling

In global scheduling we can distinguish between task and job migration (also referred to as restricted- and full-migration scheduling, respectively (Carpenter et al., 2004)). When only task migration is allowed, then tasks are allowed to migrate between processors, however, each job must execute on one processor. When job migration is allowed, then jobs may migrate between processors, i.e. they can halt on one processor and resume on another. Grasp supports both task and job migration by having the **jobArrived**, **jobStarted**, and **jobResumed** events accept an optional **-processor** argument. In a trace containing only task migration only the **jobArrived** event will specify the **-processor** argument. In a trace containing job

migration also the **jobStarted** and **jobResumed** events will specify the **-processor** argument. Figure 6.6 illustrates job migration by having the first job of task 1 arrive at time 15 on core 2 and later at time 22 migrate to core 1.

```
newProcessor core1 -name "Core 1"
newProcessor core2 -name "Core 2"
newTask task1 -name "Task 1"
...
plot 15 jobArrived job1.1 task1 -processor core2
plot 15 jobPreempted job4.1
plot 15 jobResumed job1.1
...
plot 22 jobPreempted job1.1 -processor core1
plot 22 jobPreempted job3.1
plot 22 jobResumed job1.1 -processor core1
plot 22 jobResumed job4.1
...
```

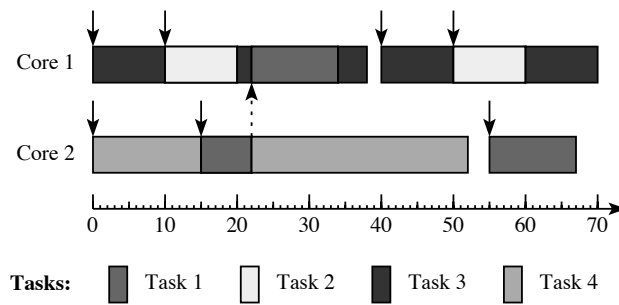


Figure 6.6: Example showing a partial trace and the corresponding visualization, illustrating the migration of a job. At time 22 the first job of task 1 migrates from core 2 to core 1, indicated by the dashed arrow.

### 6.3.3 Communication between jobs

Depending on the memory architecture in a multiprocessor system, jobs can communicate via shared memory or via message passing.

#### Shared memory

When jobs executing on different processors communicate via shared memory, it is critical to maintain the data consistency of the shared data structures. A common approach is using mutexes. Grasp provides events for acquiring and releasing a mutex, as shown in the example in Figure 6.7. The relevant events are:

- **jobAcquiredMutex** *job mutex* indicates that *job* has acquired *mutex*.

- **jobReleasedMutex** *job mutex* indicates that *job* has released *mutex*.

The arguments *job* and *mutex* are identifiers for a previously created job and mutex, respectively.

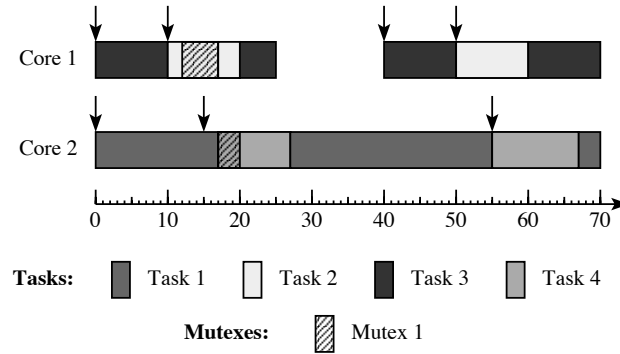


Figure 6.7: Example showing tasks 2 and 4 using a mutex to communicate via shared memory.

Figure 6.7 shows an example of two tasks communicating via shared memory. At time 12 task 2 locks Mutex 1 guarding a shared memory location. When task 4 arrives at time 15 it finds the shared mutex in a locked state and is suspended. At time 17, when task 2 unlocks the mutex, task 4 is able to resume and lock Mutex 1 to read the data communicated from task 2.

### Message passing

On a distributed memory platform jobs communicate via message passing. A popular example is the Message Passing Interface (MPI) (Pacheco, 1996). We reuse the Buffer plugin for this purpose. Depending on the communication paradigm (one to one, broadcast, multicast), we create the appropriate message buffers.

When the mouse cursor is dragged inside of the Grasp Player window, the contents of the buffers is animated, reflecting their state at the current time, indicated by the long vertical red line. Clicking on a buffer element reveals more message details (in case they were provided in the trace).

Figure 6.8 shows an example of tasks 1 and 2 communicating via message passing. At time 12, there are 2 messages A and B from task 2 inside of a message buffer, waiting for task 1 to read them.

## 6.4 Hierarchical scheduling

An interesting and unique feature of Grasp is the built in support for visualizing behavior of servers in a hierarchical real-time system. An example is shown in Figure 6.9. There are four events for tracing the budget of a server:

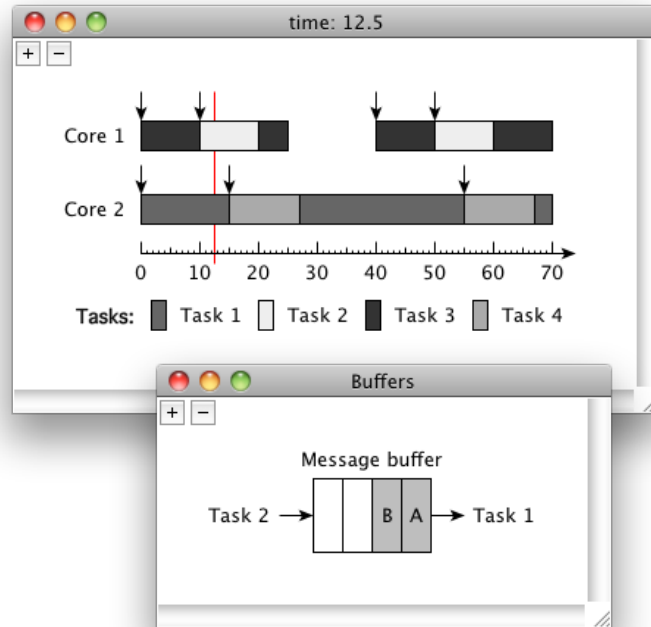


Figure 6.8: Example showing tasks 1 and 2 using a buffer to communicate via message passing.

- **serverReplenished** *server budget* indicates that *server*'s remaining budget was replenished to *budget*.
- **serverResumed** *server* indicates that a task has started consuming *server*'s budget.
- **serverPreempted** *server* indicates that a task has stopped consuming *server*'s budget.
- **serverDepleted** *server budget* indicates that *server*'s remaining budget has been depleted.

These four events are sufficient to visualize the behavior of most servers in the real-time literature. We have extended  $\mu\text{C}/\text{OS-II}$  with the polling (Lehoczky et al., 1987), periodic idling (Davis and Burns, 2005), deferrable (Strosnider et al., 1995), and constant bandwidth (Abeni and Buttazzo, 1998) servers. Figure 6.9 shows the visualization of a trace of a system comprised of two tasks mapped to a polling and a deferrable server. The task execution is shown on top, with the server capacities illustrated underneath.

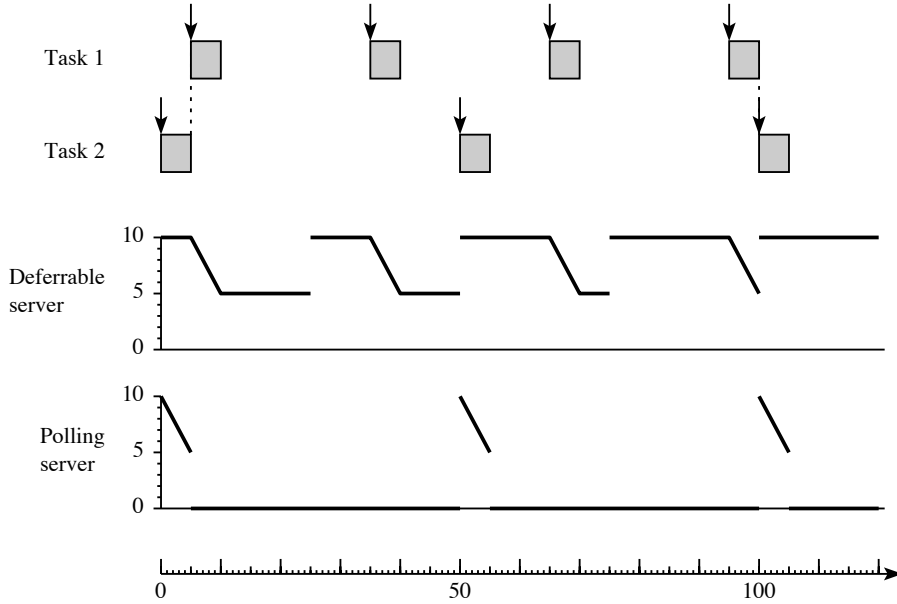


Figure 6.9: Example of a trace visualization for hierarchical scheduling. Task 1 is assigned to the Deferrable server, and Task 2 is assigned to the Polling server.

## 6.5 Hierarchical multiprocessor scheduling

In Section 6.4 we have introduced Grasp’s support for hierarchical scheduling in uniprocessor systems. In this section we discuss the combination of hierarchical and multiprocessor scheduling.

Using the standard hierarchical scheduling support, the Grasp Player is not aware of the task-to-server mapping, nor of the desired behavior of particular server types (such as periodic-idling or deferrable server). The hierarchical scheduling events pertain only to the replenishment, depletion and consumption of server’s budget. The target system is responsible for generating the correct behavior. However, the Grasp Player can be easily extended with a verification plugin, making sure that the server behavior is according to its specification, e.g. that only tasks assigned to the server consume its budget, or that a periodic idling server always idles its budget away.

The fact that Grasp is not aware of the mapping between servers and tasks allows to easily trace systems where tasks consume budgets from *several* servers, and systems where a server is serving its budget to *several* tasks executing at the same time on different processors (Shin et al., 2008). The latter is accomplished by allowing several **serverResumed** events to occur in a trace without a corresponding **serverPreempted** event in between. This provides a very simple way for tracing budget consumption in a multiprocessor setting: whenever a task assigned to a bud-

get is resumed on any processor, the corresponding server is also resumed. Similarly, a server is preempted whenever a task consuming its budget is preempted on any processor.

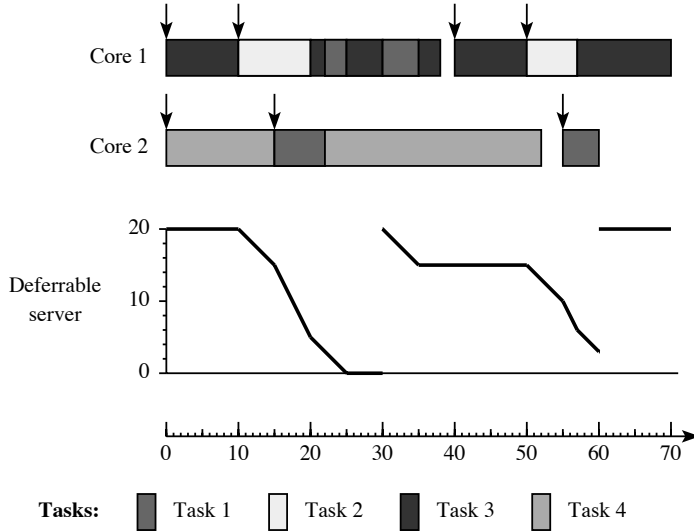


Figure 6.10: Example showing a trace visualization of a hierarchical multiprocessor system, where a deferrable server with period 30 and capacity 20 is serving its budget to tasks 1 and 2.

Figure 6.10 shows an example visualization of such a system, where a deferrable server with period 30 and capacity 20 is serving its budget to tasks 1 and 2. Tasks 3 and 4 are not bound to any server. At time 10, when task 2 arrives, it starts consuming server's budget. At time 15, when task 1 arrives, it also starts consuming server's budget. The budget is consumed at twice the rate until task 2 completes at time 20.

## 6.6 Timestamp synchronization

In Section 6.2.2, we mentioned that the events comprising a Grasp Trace do not have to be totally ordered by time. In a distributed multiprocessor system this allows to record traces on each processor individually, and then to simply concatenate the traces to form a single system trace, without the need for interleaving the events. This system trace file can then be loaded into the Grasp Player as any other trace file.

In the absence of a global time, i.e. if each processor records its local events using an asynchronous local clock, we need to synchronize the events which were recorded on different processors. The clock synchronization algorithm used by Grasp is based on the work presented in (Maillet and Tron, 1995). Unlike Maillet and Tron

(1995), however, Grasp does not enforce to synchronize with a single master node. It allows nodes to synchronize their time with an arbitrary node, creating clusters of synchronized nodes.

Let  $P_i$  (for  $i \in \{1..p\}$ ) be a processor in our target system, and let  $C_i(t)$  be the value of its local clock at physical time  $t \in \mathbb{R}$ . If we assume constant drift of each local clock, then we can express each local clock function as

$$C_i(t) = \alpha_i + \beta_i t,$$

where  $\alpha_i$  is the initial offset at  $t = 0$  and  $\beta_i$  is the drift with respect to the physical time  $t$ . We can then use the algorithm presented by Maillet and Tron (1995) to approximate the  $\alpha_i$  and  $\beta_i$  parameters, and use the  $C_i(t)$  functions to map events recorded on different processors onto the same timeline.

The algorithm relies on processors exchanging synchronization messages, as illustrated in Figure 6.11.

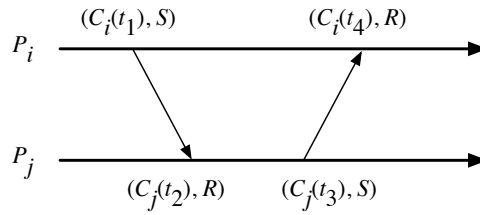


Figure 6.11: Two processors  $P_i$  and  $P_j$  exchange messages to synchronize their clocks.

The tuples indicate the recorded events:  $(C_i(t), S)$  indicates the transmission and  $(C_i(t), R)$  the reception of a synchronization message at local time  $C_i(t)$ . Processor  $P_i$  sends a message to  $P_j$  and records the event  $(C_i(t_1), S)$  using its local clock time. Upon reception of the message,  $P_j$  records the event  $(C_j(t_2), R)$  using its own local time and immediately sends a message back to  $P_i$ , recording the sending event  $(C_j(t_3), S)$ . Upon reception of the message,  $P_i$  records the event  $(C_i(t_4), R)$ .

Grasp provides the following events for synchronizing time between events generated on different processors:

- **syncSent** *source target id* indicates that the *source* processor has sent a message *id* to the *target* processor.
- **syncReceived** *source target id* indicates that the *target* processor has received a message *id* from the *source* processor.

The *id* parameter is used to match the transmission and reception events, in case several synchronization messages are sent between the same nodes. It also allows Grasp to identify lost synchronization messages.

Figure 6.12 shows an example of a system trace, which was obtained by concatenating two traces recorded on processors *core1* and *core2*. Note that each synchronization event is recorded with the local time on the processor where it is generated.



```
newProcessor core1
...
plot 15 syncSent core1 core2 m1
plot 28 syncReceived core1 core2 m2
...
newProcessor core2
...
plot 21 syncReceived core1 core2 m1
plot 22 syncSent core1 core2 m2
...
```

Figure 6.12: Example showing a partial trace illustrating the use of synchronization events.

When the Grasp Player loads the complete system trace, it has enough information to synchronize the events.

Grasp does not pose a minimum bound on the number of synchronization messages. However, the larger the number of messages and the longer the interval between the messages, the higher the accuracy of the approximation of the  $\alpha_i$  and  $\beta_i$  parameters. Of course, the accuracy comes at the cost of additional overhead for exchanging the messages and storing the corresponding events.



## Chapter 7

# Conclusion

This thesis addressed the problem of multi-resource management in embedded real-time systems. It focused on three research questions. The first question concentrated on how to design an efficient hierarchical scheduling framework for supporting independent development and analysis of component based systems, to provide temporal isolation between components. The second question investigated how to change the mapping of resources to tasks and components during run-time efficiently and predictably, and how to analyze the latency of such a system mode change in systems comprised of several scalable components. The third question dealt with the scheduling and analysis of a set of parallel-tasks with real-time constraints which require simultaneous access to several different resources.

For providing temporal isolation we chose a reservation-based approach. We first focused on processor reservations, where timed events play an important role. Common examples are task deadlines, periodic release of tasks, budget replenishment and budget depletion. Efficient timer management is therefore essential. We investigated the overheads in traditional timer management techniques and presented a mechanism called Relative Timed Event Queues (RELTEQ), which provides an expressive set of primitives at a low processor and memory overhead. We then leveraged RELTEQ for an efficient, modular and extendible design for enhancing a real-time operating system with periodic tasks, polling, idling periodic and deferrable servers, and a two-level fixed-priority Hierarchical Scheduling Framework (HSF). The HSF design provides temporal isolation and supports independent development of components by separating the global and local scheduling, and allowing each server to define a dedicated scheduler. Furthermore, the design addresses the system overheads inherent to an HSF and prevents undesirable interference between components. It limits the interference of inactive servers on the system level by means of wakeup events and a combination of inactive server queues with a stopwatch queue. Our implementation is modular and requires only a few modifications of the underlying operating system. Experimental results based on an implementation in the  $\mu\text{C}/\text{OS-II}$  real-time operating system demonstrated a reduction of system overheads and the effect of limited interference between subsystems.

We then investigated scalable components operating in a memory-constrained system. We first showed how to reduce the memory requirements in a streaming multimedia application, based on a particular priority assignment of the different components along the processing chain. Then we investigated adapting the resource provisions to tasks during runtime, referred to as mode changes. We presented a novel mode change protocol called Swift Mode Changes, which relies on Fixed Priority with Deferred preemption Scheduling to reduce the mode change latency bound compared to existing protocols based on Fixed Priority Preemptive Scheduling.

We then presented a new partitioned parallel-task scheduling algorithm called Parallel-SRP (PSRP), which generalizes MSRP for multiprocessors, and the corresponding schedulability analysis for the problem of multi-resource scheduling of parallel tasks with real-time constraints. We showed that the algorithm is deadlock-free, derived a maximum bound on blocking, and used this bound as a basis for a schedulability test. We presented an example which demonstrated how PSRP can exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources.

Finally, we presented Grasp, which is a visualization toolset aiming to provide insight into the behavior of complex real-time systems. Its flexible plugin infrastructure allows for easy extension with custom visualization and analysis techniques for automatic trace verification. Its capabilities include the visualization of hierarchical multiprocessor systems, including partitioned and global multiprocessor scheduling with migrating tasks and jobs, communication between jobs via shared memory and message passing, and hierarchical scheduling in combination with multiprocessor scheduling. For tracing distributed systems with asynchronous local clocks Grasp also supports the synchronization of traces from different processors during the visualization and analysis.

## Future work

Our RELTEQ implementation is based on a linked-list implementation of the event queues. We have indicated the challenges of a tree-based implementation due to the relative time representation. In the future we want to investigate in more detail other advanced data structures for implementing RELTEQ queues.

Our HSF implementation, as well as several HSF implementations in the literature (Behnam et al., 2008; Kato et al., 2010), implicitly assume that server budget is accounted for at tick granularity, which is also the granularity of timed events in the system. Consequently, it can be shown that the utilization of a server may approach 1, independent of its period or capacity. In the future we want to investigate how we can limit the excess utilization of servers by accounting their budget with finer granularity. This can be accomplished e.g. by reading the counter of a higher resolution timer when a server is switched in or out, instead of relying on the tick counter.

In the investigation of temporal protection and memory constrained systems we have assumed independent tasks. An interesting future work is to extend these approaches to dependent tasks which share non-preemptive resources.

The PSRP algorithm can handle non-preemptive resources with arbitrary capacity, however, it is limited to single-unit preemptive resources. While it is sufficient for partitioned parallel-task scheduling, where processors are allocated to task segments up front, in the future we want to extend PSRP to also handle multi-unit preemptive resources, which would result in a novel global parallel-task scheduler for tasks which share non-preemptive resources.

We have indicated that our schedulability analysis of PSRP is pessimistic. The analysis is pessimistic with respect to local segments, because it ignores the fact that local segments which require at least one global resource will execute non-preemptively and therefore will not suffer interference from higher priority tasks. A possible future direction for eliminating this pessimism is to combine our analysis with the analysis of Fixed Priority with Deferred Preemption scheduling.

Our schedulability analysis is also pessimistic with respect to our definition of dependency between segments, which assumes that all segments in a connected subgraph of a partial segment requirements graph depend on each other. In the future we can reduce this pessimism by considering an alternative definition of dependency, which will e.g. distinguish between dependency chains in a subgraph which share common resources and those which do not.



# Bibliography

- L. Abeni, G. Buttazzo. *Integrating multimedia applications in hard real-time systems*. In *Real-Time Systems Symposium (RTSS)*, pp. 4–14. 1998.
- H. Ahn, S. Cho, H. Na, H. Han. *Access pattern based stream buffer management scheme for portable media players*. *IEEE Transactions on Consumer Electronics*, vol. 55(3):pp. 1522–1529, 2009.
- G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. In *Spring joint computer conference*, pp. 483–485. 1967.
- J. H. Anderson, J. M. Calandrino. *Parallel real-time task scheduling on multicore platforms*. In *Real-Time Systems Symposium (RTSS)*, pp. 89–100. 2006.
- M. Åsberg, M. Behnam, F. Nemati, T. Nolte. *Towards hierarchical scheduling in AUTOSAR*. In *Emerging Technologies Factory Automation (ETFA)*, pp. 1–8. 2009.
- M. Åsberg, T. Nolte, S. Kato. *Towards hierarchical scheduling in linux/multi-core platform*. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–4. 2010.
- M. Åsberg, T. Nolte, S. Kato. *A loadable task execution recorder for hierarchical scheduling in linux*. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, vol. 1, pp. 380–387. 2011.
- N. Audsley, A. Burns, M. Richardson, K. Tindell, A. J. Wellings. *Applying new scheduling theory to static priority pre-emptive scheduling*. *Software Engineering Journal*, vol. 8(5):pp. 284–292, 1993.
- AUTOSAR. *Automotive open system architecture (AUTOSAR)*. 2011. URL <http://www.autosar.org>.
- T. P. Baker. *Stack-based scheduling for realtime processes*. *Real-Time Systems*, vol. 3(1):pp. 67–99, 1991.
- D. Becker, M. Geimer, R. Rabenseifner, F. Wolf. *Extending the scope of the controlled logical clock*. *Cluster Computing*, pp. 1–19, 2011.

- D. Becker, R. Rabenseifner, F. Wolf, J. C. Linford. *Scalable timestamp synchronization for event traces of message-passing applications*. *Parallel Computing*, vol. 35:pp. 595–607, 2009.
- M. Behnam, T. Nolte, I. Shin, M. Åsberg, R. J. Bril. *Towards hierarchical scheduling on top of VxWorks*. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 63–72. 2008.
- M. Behnam, I. Shin, T. Nolte, M. Nolin. *Sirap: a synchronization protocol for hierarchical resource sharing in real-time open systems*. In *International Conference on Embedded Software (EMSOFT)*, pp. 279–288. 2007.
- M. Biberstein, Y. Harel, A. Heilper. *Clock synchronization in cell be traces*. In *EuroPar 2008 – Parallel Processing*, vol. 5168 of *Lecture Notes in Computer Science*, pp. 3–12. Springer Berlin / Heidelberg, 2008.
- A. Block, H. Leontyev, B. B. Brandenburg, J. H. Anderson. *A flexible real-time locking protocol for multiprocessors*. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 47–56. 2007.
- B. B. Brandenburg, J. H. Anderson. *A comparison of the M-PCP, D-PCP, and FMLP on LITMUS<sup>RT</sup>*. In *International Conference on Principles of Distributed Systems (OPODIS)*, pp. 105–124. 2008a.
- B. B. Brandenburg, J. H. Anderson. *An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in litmus<sup>RT</sup>*. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 185–194. 2008b.
- B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, J. H. Anderson. *Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?* In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 342–353. 2008.
- D. Brash. *The ARM architecture version 6 (ARMv6)*. White Paper, 2002. URL [www.arm.com/pdfs/ARMv6\\_Architecture.pdf](http://www.arm.com/pdfs/ARMv6_Architecture.pdf).
- R. J. Bril, J. J. Lukkien, W. Verhaegh. *Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption*. *Real-Time Systems*, vol. 42(1):pp. 63–119, 2009.
- R. J. Bril, J. J. Lukkien, W. F. J. Verhaegh. *Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 269–279. 2007.
- B. Buchanan, D. Niehaus, S. Sheth, Y. Wijata. *The data stream kernel interface*. Tech. Rep. ITTC-FY98-TR11510-04, University Of Kansas, 1998.
- A. Burns. *Preemptive priority based scheduling: An appropriate engineering approach*. In S. Son, ed., *Advances in Real-Time Systems*, pp. 225–248. Prentice-Hall, 1994.



- A. Burns. *Defining new non-preemptive dispatching and locking policies for ada*. Reliable Software Technologies — Ada-Europe 2001, pp. 328–336, 2001.
- A. Burns, M. Nicholson, K. Tindell, N. Zhang. *Allocating and scheduling hard real-time tasks on a parallel processing platform*. Tech. Rep. YCS-94-238, University of York, UK, 1994.
- A. Burns, A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Third edition ed. Addison Wesley Longman, 2001.
- G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 2nd ed. Springer, 2004.
- G. Buttazzo, P. Gai. *Efficient EDF implementation for small embedded systems*. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. 2006.
- J. M. Calandrino, J. H. Anderson, D. P. Baumberger. *A hybrid real-time scheduling approach for large-scale multicore platforms*. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pp. 247–258. 2007.
- A. Carlini, G. C. Buttazzo. *An efficient time representation for real-time embedded systems*. In *ACM Symposium on Applied Computing*, pp. 705–712. 2003.
- J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, S. Baruah. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chap. A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms, pp. 30–1 – 30–19. Chapman and Hall/CRC, 2004.
- S. Chodrow, F. Jahanian, M. Donner. *Run-time monitoring of real-time systems*. In *Real-Time Systems Symposium (RTSS)*, pp. 74–83. 1991.
- F. Cristia. *Probabilistic clock synchronization*. Distributed Computing, vol. 3:pp. 146–158, 1989.
- R. I. Davis, A. Burns. *Hierarchical fixed priority pre-emptive scheduling*. In *Real-Time Systems Symposium (RTSS)*, pp. 389–398. 2005.
- R. I. Davis, A. Burns. *A survey of hard real-time scheduling for multiprocessor systems*. ACM Computing Surveys, vol. 43(4):pp. 35:1–35:44, 2011.
- E. W. Dijkstra. *Een algoritme ter voorkoming van de dodelijke omarming*, 1964. URL <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF>. Circulated privately.
- E. W. Dijkstra. *The mathematics behind the Banker’s Algorithm*. In *Selected Writings on Computing: A Personal Perspective*, pp. 308–312. Springer-Verlag, 1982.
- Dtrace. <http://wikis.sun.com/display/dtrace/dtrace>. 2008.

- A. Easwaran, I. Shin, I. Lee. *Optimal virtual cluster-based multiprocessor scheduling*. Real-Time Systems, vol. 43:pp. 25–59, 2009.
- D. R. Engler, M. F. Kaashoek, J. O’Toole, Jr. *Exokernel: an operating system architecture for application-level resource management*. In *Symposium on Operating Systems Principles (SOSP)*, pp. 251–266. 1995.
- A. Eswaran, R. R. Rajkumar. *Energy-aware memory firewalling for qos-sensitive applications*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 11–20. 2005.
- A. Eswaran, A. Rowe, R. Rajkumar. *Nano-RK: An energy-aware resource-centric RTOS for sensor networks*. In *Real-Time Systems Symposium (RTSS)*, pp. 256–265. 2005.
- Evidence. *ERIKA Enterprise: Open Source RTOS for single- and multi-core applications*. 2010. URL <http://www.evidence.eu.com>.
- D. Faggioli, M. Trimarchi, F. Checconi, C. Scordino. *An EDF scheduling class for the linux kernel*. In *Real-Time Linux Workshop*. 2009.
- D. G. Feitelson. *Distributed hierarchical control for parallel processing*. Computer, vol. 23(5):pp. 65–77, 1990.
- D. G. Feitelson, L. Rudolph. *Gang scheduling performance benefits for fine-grain synchronization*. Journal of Parallel and Distributed Computing, vol. 16(4):pp. 306 – 318, 1992.
- F. H. Fitzek, M. Reisslein. *MPEG4 and H.263 video traces for network performance evaluation*. IEEE Network, vol. 15(6):pp. 40–53, 2001.
- Ftrace. <http://www.kernel.org/doc/Documentation/trace/ftrace.txt>. 2008.
- P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, P. Marceca. *A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform*. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 189–198. 2003.
- P. Gai, G. Lipari, M. D. Natale. *Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip*. In *Real-Time Systems Symposium (RTSS)*, pp. 73–83. 2001.
- J. J. G. García, J. C. P. Gutiérrez, M. G. Harbour. *Schedulability analysis of distributed hard real-time systems with multiple-event synchronization*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 15–24. 2000.
- T. Geelen. *Dynamic loading in a real-time system: An overlaying technique using virtual memory*. Tech. rep., Eindhoven University of Technology, 2005.

- M. Geimer, F. Wolf, B. J. N. Wylie, D. Becker, D. Böhme, W. Frings, M.-A. Hermanns, B. Mohr, Z. Szebenyi. *Recent developments in the scalasca toolset*. In *International Workshop on Parallel Tools for High Performance Computing*, pp. 39–51. Springer, 2010.
- Gnuplot. <http://www.gnuplot.info>. 2010.
- J. Goossens, V. Berten. *Gang FTP scheduling of periodic and parallel rigid real-time tasks*. In *Real-Time and Network Systems (RTNS)*, pp. 189–196. 2010.
- R. Gopalakrishnan, G. M. Parulkar. *Bringing real-time scheduling theory and practice closer for multimedia computing*. SIGMETRICS Perform. Eval. Rev., vol. 24(1):pp. 1–12, 1996.
- A. N. Habermann. *Prevention of system deadlocks*. Commun. ACM, vol. 12:pp. 373–382, 1969.
- P. B. Hansen. *Operating system principles*. Prentice-Hall, 1973.
- B. G. Haskell, A. Puri, A. N. Netravali. *Digital Video: An introduction to MPEG-2*. Chapman & Hall, Ltd., 1996.
- J. W. Havender. *Avoiding deadlock in multitasking systems*. IBM Systems Journal, vol. 7(2):pp. 74–84, 1968.
- C. Hentschel, R. Bril, Y. Chen, R. Braspenning, T.-H. Lan. *Video quality-of-service for consumer terminals - a novel system for programmable components*. IEEE Transactions on Consumer Electronics, vol. 49(4):pp. 1367–1377, 2003.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Swift mode changes in memory constrained real-time systems*. In *International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 262–269. 2009a.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Swift mode changes in memory constrained real-time systems*. Tech. Rep. CS-09-08, Eindhoven University of Technology, 2009b.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems*. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2011a.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Reducing memory requirements in a multimedia streaming application*. In *International Conference on Consumer Electronics (ICCE)*. 2011b.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Reducing memory requirements in a multimedia streaming application*. IEEE Transactions on Consumer Electronics, vol. 57(1):pp. 145–152, 2011c.

- M. Holenderski, R. J. Bril, J. J. Lukkien. *An efficient hierarchical scheduling framework for the automotive domain*. In S. M. Babamir, ed., *Real-Time Systems, Architecture, Scheduling, and Application*. InTech, 2012a.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems*. Journal of Systems Architecture, 2012b.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Parallel-task scheduling on multiple resources*. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 2012c.
- M. Holenderski, W. Cools, R. J. Bril, J. J. Lukkien. *Multiplexing real-time timed events*. In *Emerging Technologies and Factory Automation (ETFA)*, pp. 1718–1721. 2009c.
- M. Holenderski, W. Cools, R. J. Bril, J. J. Lukkien. *Extending an open-source real-time operating system with hierarchical scheduling*. Tech. Rep. CS-report 10-10, Eindhoven University of Technology, 2010a.
- M. Holenderski, C. G. Okwudire, R. J. Bril, J. J. Lukkien. *Memory management for multimedia quality of service in resource constrained embedded systems*. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. 2010b.
- M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien. *Grasp: Tracing, visualizing and measuring the behavior of real-time systems*. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pp. 37–42. 2010c.
- R. C. Holt. *Some deadlock properties of computer systems*. ACM Comput. Surv., vol. 4:pp. 179–196, 1972.
- S. Hunold, R. Hoffmann, F. Suter. *Jedule: A tool for visualizing schedules of parallel applications*. In *International Conference on Parallel Processing Workshops (ICPPW)*, pp. 169–178. 2010.
- R. Inam, J. Maki-Turja, M. Sjödin, S. M. H. Ashjaei, S. Afshar. *Support for hierarchical scheduling in freertos*. In *Emerging Technologies Factory Automation (ETFA)*, pp. 1–10. 2011.
- D. Isovici, G. Fohler. *Quality aware MPEG-2 stream adaptation in resource constrained systems*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 23–32. 2004.
- D. Jarnikov. *QoS Framework for Video Streaming in Home Networks*. Ph.D. thesis, Eindhoven University of Technology, 2007.
- D. Jarnikov, P. van der Stok, C. Wust. *Predictive control of video quality under fluctuating bandwidth conditions*. In *International Conference on Multimedia and Expo (ICME)*, vol. 2, pp. 1051–1054. 2004.

- S. Kato, Y. Ishikawa. *Gang EDF scheduling of parallel task systems*. In *Real-Time Systems Symposium (RTSS)*, pp. 459–468. 2009.
- S. Kato, R. Rajkumar, Y. Ishikawa. *A loadable real-time scheduler framework for multicore platforms*. In *Submitted to Real-Time Computing Systems and Applications (RTCSA)*. 2010.
- J. C. D. Kergommeaux, B. D. O. Stein, M. S. Martin. *Paje: An extensible environment for visualizing multi-threaded program executions*. LNCS 1900, 2000.
- D. Kim, Y.-H. Lee, M. Younis. *SPIRIT- $\mu$ Kernel for strongly partitioned real-time systems*. In *Real-Time Computing Systems and Applications (RTCSA)*, pp. 73–80. 2000.
- S. Kim, T. Kim, E. G. Im, H. Han. *Efficient reuse of local regions in memory-limited mobile devices*. IEEE Transactions on Consumer Electronics, vol. 56(3):pp. 1297–1303, 2010.
- J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 1998.
- J. J. Labrosse. *MicroC/OS-II: The Real Time Kernel*. CMP Books, 2nd ed., 2002.
- K. Lakshmanan, D. de Niz, R. Rajkumar. *Coordinated task scheduling, allocation and synchronization on multiprocessors*. In *Real-Time Systems Symposium (RTSS)*, pp. 469–478. 2009.
- K. Lakshmanan, S. Kato, R. Rajkumar. *Scheduling parallel real-time tasks on multi-core processors*. In *Real-Time Systems Symposium (RTSS)*, pp. 259–268. 2010.
- T. Lan, Y. Chen, Z. Zhong. *Mpeg2 decoding complexity regulation for a media processor*. In *Workshop on Multimedia Signal Processing (MMSP)*, pp. 193–198. 2001.
- J. Lehoczky, L. Sha, Y. Ding. *The rate monotonic scheduling algorithm: exact characterization and average case behavior*. In *Real Time Systems Symposium (RTSS)*, pp. 166–171. 1989.
- J. P. Lehoczky, L. Sha, J. K. Strosnider. *Enhanced aperiodic responsiveness in hard real-time environments*. In *Real-Time Systems Symposium (RTSS)*, pp. 261–270. 1987.
- X. Li, M. Malek. *Analysis of speedup and communication/computation ratio in multiprocessor systems*. In *Real-Time Systems Symposium (RTSS)*, pp. 282–288. 1988.
- G. Lipari, E. Bini. *A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation*. In *Real-Time Systems Symposium (RTSS)*, pp. 249–258. 2010.
- C. L. Liu, J. W. Layland. *Scheduling algorithms for multiprogramming in a hard-real-time environment*. J. ACM, vol. 20(1):pp. 46–61, 1973.

- E. Maillet, C. Tron. *On efficiently implementing global time for performance evaluation on multiprocessor systems*. Parallel Distributed Computing, vol. 28:pp. 84–93, 1995.
- F. H. Martin. *Apollo 11: 25 years later*. Apollo 11 Lunar Surface Journal, 1994.
- S. McCartney. *Eniac: The Triumphs and Tragedies of the World's First Computer*. Walker & Company, 1999.
- R. McIlroy, P. Dickman, J. Sventek. *Efficient dynamic heap allocation of scratchpad memory*. In *International Symposium on Memory Management (ISMM)*, pp. 31–40. 2008.
- F. Menichelli, M. Olivieri. *Static minimization of total energy consumption in memory subsystem for scratchpad-based systems-on-chips*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17(2):pp. 161–171, 2009.
- C. Mercer, R. Rajkumar, J. Zelenka. *Temporal protection in real-time operating systems*. In *IEEE Workshop on Real-Time Operating Systems and Software*, pp. 79–83. 1994.
- D. L. Mills. *Network Time Protocol (Version 3)*. The Internet Engineering Task Force—Network Working Group, 1992. RFC 1305.
- A. K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.
- M. I. Mughal, R. Javed. *Recording of Scheduling and Communication events on Telecom Systems*. Master's thesis, Mälardalen University, 2008.
- W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, K. Solchenbach. *Vampir: Visualization and analysis of MPI resources*. Supercomputer, vol. 12:pp. 69–80, 1996.
- T. Nakajima. *Resource reservation for adaptive QoS mapping in real-time mach*. Parallel and Distributed Processing, pp. 1047–1056, 1998.
- T. Nolte. *Compositionality and CPS from a platform perspective*. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, vol. 2, pp. 57–60. 2011.
- T. Nolte, I. Shin, M. Behnam, M. Sjödin. *A synchronization protocol for temporal isolation of software components in vehicular systems*. IEEE Transactions on Industrial Informatics, vol. 5(4):pp. 375–387, 2009.
- F. O'Brien. *The Apollo Guidance Computer: Architecture and Operation*. Praxis Publishing Ltd., 2010.
- S. Oikawa, R. Rajkumar. *Portable RK: a portable resource kernel for guaranteed and enforced timing behavior*. In *Real-Time Technology and Applications Symposium (RTAS)*, pp. 111–120. 1999.

- OpenCores. *Openrisc 1000: Architectural simulator*. 2010. URL <http://opencores.org/openrisc,or1ksim>.
- J. Ousterhout. *Scheduling techniques for concurrent systems*. In *International Conference on Distributed Computing Systems*, pp. 22–30. 1982.
- P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.
- L. Palopoli, T. Cucinotta, L. Marzario, G. Lipari. *Aquosa—adaptive quality of service architecture*. *Software – Practice and Experience*, vol. 39(1):pp. 1–31, 2009.
- D. Pollock, D. Zöbel. *Conformance testing of priority inheritance protocols*. In *International Conference on Real-Time Systems and Applications (RTCSA'00)*, p. 404. 2000.
- R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. *Resource kernels: A resource-centric approach to real-time and multimedia systems*. In *Conference on Multimedia Computing and Networking (CMCN)*, pp. 150–164. 1998.
- R. Rajkumar, L. Sha, J. Lehoczky. *Real-time synchronization protocols for multiprocessors*. In *Real-Time Systems Symposium (RTSS)*, pp. 259–269. 1988.
- J. Real. *Protocolos de cambio de mundo para sistemas de tiempo real (mode change protocols for real time systems)*. Ph.D. thesis, Technical University of Valencia, 2000.
- J. Real, A. Crespo. *Mode change protocols for real-time systems: A survey and a new proposal*. *Real-Time Systems*, vol. 26(2):pp. 161–197, 2004.
- S. Saewong, R. R. Rajkumar, J. P. Lehoczky, M. H. Klein. *Analysis of hierarchical fixed-priority scheduling*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, p. 173. 2002.
- A. Saifullah, K. Agrawal, C. Lu, C. Gill. *Multi-core real-time scheduling for generalized parallel task models*. In *Real-Time Systems Symposium (RTSS)*, pp. 217–226. 2011.
- L. Sha, R. Rajkumar, J. Lehoczky, K. Ramamritham. *Mode change protocols for priority-driven preemptive scheduling*. Tech. Rep. CMU/SEI-88-TR-34, Carnegie Mellon University, 1988.
- L. Sha, R. Rajkumar, J. P. Lehoczky. *Priority inheritance protocols: An approach to real-time synchronization*. *IEEE Transactions on Computers*, vol. 39(9):pp. 1175–1185, 1990.
- I. Shin, A. Easwaran, I. Lee. *Hierarchical scheduling framework for virtual clustering of multiprocessors*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 181–190. 2008.

- I. Shin, I. Lee. *Periodic resource model for compositional real-time guarantees*. In *Real-Time Systems Symposium (RTSS)*, pp. 2–13. 2003.
- I. Shin, I. Lee. *Compositional real-time scheduling framework with periodic model*. *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 7:pp. 30:1–30:39, 2008.
- J. K. Strosnider, J. P. Lehoczky, L. Sha. *The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*. *IEEE Transactions on Computers*, vol. 44(1):pp. 73–91, 1995.
- TimeDoctor. <http://sourceforge.net/projects/timedoctor/>. 2011.
- K. Tindell, A. Alonso. *A very simple protocol for mode changes in priority preemptive systems*. Tech. rep., Universidad Politecnica de Madrid, 1996.
- K. Tindell, J. Clark. *Holistic schedulability analysis for distributed hard real-time systems*. *Microprocess. Microprogram.*, vol. 40(2-3):pp. 117–134, 1994.
- M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien. *Protocol-transparent resource sharing in hierarchically scheduled real-time systems*. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. 2010.
- M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, J. J. Lukkien. *Constant-bandwidth supply for priority processing*. *IEEE Transactions on Consumer Electronics*, vol. 57(2):pp. 873–881, 2011.
- M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, J. J. Lukkien. *Virtual timers in hierarchical real-time systems*. In *Work in Progress session of the Real-time Systems Symposium (RTSS)*, pp. 35–38. 2009.
- M. Weffers-Albu. *Behavioral analysis of real-time systems with interdependent tasks*. Ph.D. thesis, Technische Universiteit Eindhoven, 2008.
- B. Welch, K. Jones, J. Hobbs. *Practical Programming in Tcl and Tk*. Prentice Hall, 2003.
- T. Wiegand, G. Sullivan, G. Bjontegaard, A. Luthra. *Overview of the h.264/avc video coding standard*. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13(7):pp. 560 –576, 2003.
- C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, C. Hentschel. *Qos control strategies for high-quality video processing*. *Real-Time Systems*, vol. 30(1-2):pp. 7–29, 2005.
- K. S. Yim, H. Bahn, K. Koh. *A flash compression layer for smartmedia card systems*. *IEEE Transactions on Consumer Electronics*, vol. 50(1):pp. 192 – 197, 2004.
- D. Zöbel, D. Polock, A. van Arkel. *Testing for the conformance of real-time protocols implemented by operating systems*. *Electronic Notes in Theoretical Computer Science*, vol. 133:pp. 315–332, 2005.



# Symbol index

$\mathcal{R}$	Set of all resources	5
$\mathcal{P}$	Set of preemptive resources	6
$\mathcal{N}$	Set of nonpreemptive resources	6
$\mathcal{T}$	Time domain	7
$\Gamma$	Set of all tasks	7
$\mathcal{S}$	Set of all segments	7
$R_{i,j}$	Set of resource requirements of segment $\tau_{i,j}$	7
$E_{i,j}$	Worst-case execution time of segment $\tau_{i,j}$	7
$E_i$	Worst-case execution time of task $\tau_i$	8
$\mathcal{C}$	Set of all components	9
$\pi_c$	Priority of component $c$	9
$T_c$	Period of component $c$	9
$O_c$	Offset (or phasing) of component $c$	9
$D_c$	Deadline of component $c$	9
$R_c$	Set of resource requirements of component $c$	9
$P_c$	Set of resource provisions of component $c$	9
$E_c$	Time capacity of component $c$	9
$\sigma^S(t, s, r)$	Number of units of resource $r$ owned by segment $s$ at time $t$	13
$\eta^S(t, s, r)$	Returns 1 if segment $s$ is scheduled on resource $r$ at time $t$ , 0 otherwise	13
$\sigma^C(t, c, r)$	Number of units of resource $r$ owned by component $c$ at time $t$	14
$\eta^C(t, c, r)$	Returns 1 if component $c$ is scheduled on resource $r$ at time $t$ , 0 otherwise	14
$\beta(t, c)$	Remaining budget of component $c$ at time $t$	14
$\gamma(c)$	Set of tasks requiring component $c$	25
$\lambda(\tau_i)$	Set of components required by task $\tau_i$	25
$\nu(t, r)$	Number of units of resource $r$ available at time $t$	25
$\alpha(t, \tau_i)$	Segment of task $\tau_i$ which is active at time $t$	25
$R_{\alpha(t, \tau_i)}$	Set of resources required by task $\tau_i$ at time $t$	25
$s.sq$	Server queue of server $s$	36
$s.se$	Stopwatch event of server $s$	36

$s.vq$	Virtual server queue of server $s$	39
$s.we$	Wakeup event of server $s$	38
$\mathcal{M}$	Set of system mode identifiers	66
$\Gamma^S$	Set of scalable tasks	66
$\mathcal{C}^S$	Set of scalable components	66
$\tilde{E}_i^k$	Execution time needed by task $\tau_i$ to process the $k^{\text{th}}$ frame	64
$\hat{E}_i$	Worst-case execution time of task $\tau_i$	65
$\vec{E}_{a..b}^k$	Execution time needed by task chain $\tau_{i..j}$ to process the $k^{\text{th}}$ frame	64
$\tau_i^k$	Specification of scalable task $\tau_i$ during system mode $k$ , where $\tau_i = (\tau_i^1, \tau_i^2, \dots, \tau_i^k, \dots, \tau_i^{ \mathcal{M} })$	66
$c^k$	Specification of scalable component $c$ during system mode $k$ , where $c = (c^1, c^2, \dots, c^k, \dots, c^{ \mathcal{M} })$	66
$\acute{E}(c)$	Worst-case pre- and post-processing time of component $c$ .	81
$\phi^\Gamma(x, y)$	Set of tasks <i>directly</i> involved in the mode change from mode $x$ to mode $y$	82
$\phi^C(x, y)$	Set of components <i>directly</i> involved in the mode change from mode $x$ to mode $y$	82
$\Phi(x, y)$	Set of tasks <i>directly and indirectly</i> involved in the mode change from mode $x$ to mode $y$	83
$\kappa(r)$	Set of segments requiring resource $r$	108
$\mathcal{R}^L$	Set of local resources	108
$\mathcal{R}^G$	Set of global resources	108
$\mathcal{S}^L$	Set of local segments	109
$\mathcal{S}^G$	Set of global segments	109
$partial(G)$	Set of partial segment requirements graphs derived from graph $G$	119
$\delta(\tau_{i,j}, g)$	Set of segments which segment $\tau_{i,j}$ can reach in the partial segment requirements graph $g$	119
$E'(\tau_{i,j})$	Execution time of segment $\tau_{i,j}$ extended with its waiting time	122
$A(\tau_{i,j})$	Worst-case activation time of segment $\tau_{i,j}$	122

# Acronyms

AUTOSAR	Automotive Open System Architecture
DPCP	Distributed Priority Ceiling Protocol
DSKI	Data Stream Kernel Interface
ECU	Electronic Control Unit
EDF	Earliest Deadline First
FCFS	First Come First Serve
FMLP	Flexible Multiprocessor Locking Protocol
FMLP P-SP	Flexible Multiprocessor Locking Protocol for Partitioned Static-Priority scheduling
FPDS	Fixed-Priority Deferred Scheduling
FPNS	Fixed Priority Non-preemptive Scheduling
FPFS	Fixed-Priority Preemptive Scheduling
FPS	Fixed-Priority Scheduling
HSF	Hierarchical Scheduling Framework
ICTOH	Implicit Circular Timers Overflow Handler
ISR	Interrupt Service Routine
MPCP	Multiprocessor Priority Ceiling Protocol
MPI	Message Passing Interface
MSRP	Multiprocessor-SRP
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol
PSRP	Parallel-SRP
QM	Quality Manager
RELTEQ	Relative Timed Event Queues
RM	Resource Manager
RTOS	Real-Time Operating System
SCS	Spread-Cognizant Scheduling
SDK	Software Development Kit
SRP	Stack Resource Policy
TCB	Task Control Block
TSC	Time Stamp Counter
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time



# Accomplishments

This appendix contains a list of papers of which Mike Holenderski is an author, a list of Master thesis and internship projects which Mike has supervised, and courses which Mike has assisted with.

## Peer-reviewed publications

- M. Bergsma, M. Holenderski, R. J. Bril, J. J. Lukkien. *Extending RTAI/Linux with fixed-priority scheduling with deferred preemption*. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. 2009.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth*. In *Work in Progress session of the Euromicro Conference on Real-Time Systems (ECRTS)*. 2008.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Swift mode changes in memory constrained real-time systems*. In *International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 262–269. 2009a.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems*. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2011a.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Reducing memory requirements in a multimedia streaming application*. In *International Conference on Consumer Electronics (ICCE)*. 2011b.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Reducing memory requirements in a multimedia streaming application*. *IEEE Transactions on Consumer Electronics*, vol. 57(1):pp. 145–152, 2011c.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *An efficient hierarchical scheduling framework for the automotive domain*. In S. M. Babamir, ed., *Real-Time Systems, Architecture, Scheduling, and Application*. InTech, 2012a.

- M. Holenderski, R. J. Bril, J. J. Lukkien. *Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems*. Journal of Systems Architecture, 2012b.
- M. Holenderski, R. J. Bril, J. J. Lukkien. *Parallel-task scheduling on multiple resources*. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 2012c.
- M. Holenderski, W. Cools, R. J. Bril, J. J. Lukkien. *Multiplexing real-time timed events*. In *Emerging Technologies and Factory Automation (ETFA)*, pp. 1718–1721. 2009b.
- M. Holenderski, C. G. Okwudire, R. J. Bril, J. J. Lukkien. *Memory management for multimedia quality of service in resource constrained embedded systems*. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. 2010a.
- M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien. *Grasp: Tracing, visualizing and measuring the behavior of real-time systems*. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pp. 37–42. 2010b.
- M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, J. J. Lukkien. *Constant-bandwidth supply for priority processing*. In *International Conference on Consumer Electronics (ICCE)*. 2011a.
- M. M. H. P. van den Heuvel, M. Holenderski, R. J. Bril, J. J. Lukkien. *Constant-bandwidth supply for priority processing*. IEEE Transactions on Consumer Electronics, vol. 57(2):pp. 873–881, 2011b.
- M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, J. J. Lukkien. *Virtual timers in hierarchical real-time systems*. In *Work in Progress session of the Real-time Systems Symposium (RTSS)*, pp. 35–38. 2009.

## Program committees

During his Ph.D., Mike was on the following program committees:

- Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2012

## Supervision

During his Ph.D., Mike was involved in supervising the following Master of computer Science thesis:

- *Extending RTAI Linux with FPDS* by Mark Bergsma,
- *Support for fast mode-changes of applications* by Martijn van den Heuvel,

- *Extending uC/OS-II with FPDS and Reservations* by Wim Cools,
- *Modeling and measuring the performance of a surveillance camera* by Norbert Verhagen.

and the following internships:

- *Memory sharing for queues of media processing chains* by Ashu Gebreweld,
- *Dynamic memory re-allocation for swift mode changes* by Chidi Okwudire.

## Courses

During his Ph.D., Mike was involved in

- assisting with the “Real-time systems architectures” course in the Master of Computer Science program in the years 2009/2010 and 2010/2011,
- assisting with the “Real-time, embedded and concurrent programming” workshop in the Professional Doctorate in Engineering program in the years 2010/2011 and 2011/2012,
- setting up and subsequently assisting with the “Real-time systems architectures (in automotive domain)” course in the Master of Embedded Systems program in the year 2011/2012.





## Summary

This thesis addresses the problem of online multi-resource management in embedded real-time systems. It focuses on three research questions. The first question concentrates on how to design an efficient hierarchical scheduling framework for supporting independent development and analysis of component based systems, to provide temporal isolation between components. The second question investigates how to change the mapping of resources to tasks and components during run-time efficiently and predictably, and how to analyze the latency of such a system mode change in systems comprised of several scalable components. The third question deals with the scheduling and analysis of a set of parallel-tasks with real-time constraints which require simultaneous access to several different resources.

For providing temporal isolation we chose a reservation-based approach. We first focused on processor reservations, where timed events play an important role. Common examples are task deadlines, periodic release of tasks, budget replenishment and budget depletion. Efficient timer management is therefore essential. We investigated the overheads in traditional timer management techniques and presented a mechanism called Relative Timed Event Queues (RELTEQ), which provides an expressive set of primitives at a low processor and memory overhead. We then leveraged RELTEQ to create an efficient, modular and extensible design for enhancing a real-time operating system with periodic tasks, polling, idling periodic and deferrable servers, and a two-level fixed-priority Hierarchical Scheduling Framework (HSF). The HSF design provides temporal isolation and supports independent development of components by separating the global and local scheduling, and allowing each server to define a dedicated scheduler. Furthermore, the design addresses the system overheads inherent to an HSF and prevents undesirable interference between components. It limits the interference of inactive servers on the system level by means of wakeup events and a combination of inactive server queues with a stopwatch queue. Our implementation is modular and requires only a few modifications of the underlying operating system.

We then investigated scalable components operating in a memory-constrained system. We first showed how to reduce the memory requirements in a streaming multimedia application, based on a particular priority assignment of the different components along the processing chain. Then we investigated adapting the resource provisions to tasks during runtime, referred to as mode changes. We presented a novel mode change protocol called Swift Mode Changes, which relies on Fixed Priority with

Deferred preemption Scheduling to reduce the mode change latency bound compared to existing protocols based on Fixed Priority Preemptive Scheduling.

We then presented a new partitioned parallel-task scheduling algorithm called Parallel-SRP (PSRP), which generalizes MSRP for multiprocessors, and the corresponding schedulability analysis for the problem of multi-resource scheduling of parallel tasks with real-time constraints. We showed that the algorithm is deadlock-free, derived a maximum bound on blocking, and used this bound as a basis for a schedulability test. We then demonstrated how PSRP can exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources.

Finally, we presented Grasp, which is a visualization toolset aiming to provide insight into the behavior of complex real-time systems. Its flexible plugin infrastructure allows for easy extension with custom visualization and analysis techniques for automatic trace verification. Its capabilities include the visualization of hierarchical multiprocessor systems, including partitioned and global multiprocessor scheduling with migrating tasks and jobs, communication between jobs via shared memory and message passing, and hierarchical scheduling in combination with multiprocessor scheduling. For tracing distributed systems with asynchronous local clocks Grasp also supports the synchronization of traces from different processors during the visualization and analysis.

# Curriculum Vitae

Michał Holenderski was born on January 12, 1982 in Warsaw, Poland. After finishing the International Secondary School in Eindhoven, Netherlands, in 2000, he studied Computer Science at the Eindhoven University of Technology (TU/e), where he received his Bachelor of Science in 2003 and Master of Science in 2007. During his Master studies he did two six-months internships: at the National University of Singapore (NUS), and at Sun Microsystems Labs (SunLabs) in Menlo Park, USA. At NUS he conducted research into distributing machine learning in a wireless sensor network. At SunLabs he investigated programming FLEET, an experimental asynchronous data-driven processor architecture, which was also the topic of his Master thesis. In 2007 Michał started a PhD at the Systems Architecture and Networking (SAN) group at the TU/e, of which the results are presented in this thesis. In 2011 he joined the SAN group as a post-doctoral researcher, and founded a startup which focuses on developing software for mobile platforms.