Adaptive CPU Resource Management for Multicore Platforms

Romero Segovia, Vanessa

2011

Document Version:
Publisher's PDF, also known as Version of record

Adaptive CPU Resource Management for Multicore Platforms

Vanessa Romero Segovia

Department of Automatic Control
Lund University
Lund, September 2011

To dreams that come true far away from home

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT--3252--SE

Printed in Sweden,
Lund University, Lund 2011

Abstract

The topic of this thesis is adaptive CPU resource management for multicore platforms. The work was done as a part of the resource manager component of the adaptive resource management framework implemented in the European ACTORS project. The framework dynamically allocates CPU resources for the applications. The key element of the framework is the resource manager that combines feedforward and feedback algorithms together with reservation techniques. The reservation techniques are supported by a new Linux scheduler through hard constant bandwidth server reservations. The resource requirements of the applications are provided through service level tables. Dynamic bandwidth allocation is performed by the resource manager which adapts applications to changes in resource availability, and adapts the resource allocation to changes in application requirements. The dynamic bandwidth allocation allows to obtain real application models through the tuning and update of the initial service level tables.

Acknowledgements

I want to begin by thanking the Department of Automatic Control for accepting me as one of its members and thereby opening up new vistas of achieving my valued goal in life —To continuously strive for excellence in the things that I like the most.

I extend my sincere thanks to my supervisor Karl-Erik for his never ending patience and support of my ideas needed to accomplish this work. My thanks also to Tore for his constant motivation and encouragement that helped me make the right decisions during the course of this work.

I would like to express my heartfelt gratitude to my dear parents in Peru who have been a continuous source of inspiration and supported me in the pursuit of my dreams.

I would also like to thank my dear friends Niranjana and Songwei, for having shared my joys and sorrows, despite the geographical distance separating us.

It would be an incomplete acknowledgement without mentioning the enduring support of my beloved husband and soul mate, Patrick, who has always been there in every journey of my life reassuring me of the important things in life.

This research has partially been supported by the EC ICT FP7 project ACTORS (ICT-216586), the EC NoE ArtistDesign, and the Linneaus Center LCCC.

Contents

1

Introduction

1.1 Motivation

The need for adaptivity in embedded systems is becoming more urgent with the continuous evolution towards much richer feature sets and demands for sustainability.

The European FP7 project ACTORS (Adaptivity and Control of Resources for Embedded Systems) [1] has developed an adaptive CPU resource management framework. The framework consists of three layers: the application, the resource manager, and the scheduler layer. The target systems of the framework are Linux-based multicore platforms and is mainly intended for soft real-time applications.

The ideas presented in this thesis are driven by the desire to automatically allocate the available CPU resources at runtime, and to adapt the allocated resources to the real needs of the applications. This work considers the resource manager as the key element of the ACTORS framework. As a result it focuses all its efforts in the development of different methods and algorithms for the resource manager.

The methods and algorithms described combine feedforward and feedback techniques. The last ones have shown to be suitable to manage the uncertainty related to the real CPU requirements of the applications at runtime. In this way the resource manager is able to adapt the applications to changes in resource availability, and to adapt how the resources are distributed when the application requirements change.

1.2 Outline

This thesis is organized as follows: Chapter 2 provides the relevant background and describes related research. Chapter 3 presents the ACTORS framework and gives an overview of its different layers. The inputs and outputs of the resource manager are explained in Chapter 4. Chapter 5 introduces a feedforward algorithm that allows the registration of applications and assigns the service level at which they must execute. Chapter 6 continues with the registration process and shows different algorithms that allow the bandwidth distribution of the registered applications. Different control strategies that perform bandwidth adaption are shown in Chapter 7. Chapter 8 shows how the implemented control strategies can be used to obtain a model of the application at runtime. Adaption towards changes in resource availability and/or the changes in the relative importance of applications with respect to others is described in Chapter 9. A brief description of different applications that use the resource manager framework is shown in Chapter 10. Chapter 11 concludes the thesis.

1.3 Publications

The thesis is based on the following publications:

V. R. Segovia, K.-E. Årzén, S. Schorr, R. Guerra, and G. Fohler, "Adaptive Resource Management Framework for Mobile Terminals - the ACTORS Approach," in *Proceedings of the Workshop on Adaptive Resource Management, WARM*, Stockholm, April 2010.

V. R. Segovia and K.-E. Årzén, "Towards Adaptive Resource Management of Dataflow Applications on Multi-core Platforms," in *Proceedings Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems, ECRTS*, Brussels, July 2010.

K.-E. Årzén, V. R. Segovia, M. Kralmark, S. Schorr, A. Meher, and G. Fohler, "ACTORS Adaptive Resource Management Demo," in *Proceedings of the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems, APRES*, Chicago, April 2011.

K.-E. Årzén, V. R. Segovia, S. Schorr, and G. Fohler, "Adaptive Resource Management Made Real," in *Proceedings of the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems, APRES*, Chicago, April 2011.

E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzén, V. R. Segovia, and C. Scordino, "Resource management on multicore systems: The ACTORS approach," *IEEE Micro*, vol. 31, no. 3, May/June 2011.

V. R. Segovia and K.-E. Årzén, "Adaptive Bandwidth Resource Management in Multicore Platforms," in *Proceedings of the 12th Real-Time in Sweden Conference, RTiS*, Västerås, June 2011.

V. R. Segovia, M. Kralmark, M. Lindberg, and K.-E. Årzén, "Processor thermal control using adaptive bandwidth resource management," in *Proceedings of the 18th World Congress of the International Federation of Automatic Control, IFAC*, Milan, August 2011.

The contributions by the author in the publications above mainly concern the different algorithms implemented by the resource manager to allocate bandwidth resources to the applications, and to adapt the allocated bandwidth to the real needs of the applications.

The author has also contributed to the following ACTORS deliverables:

K.-E. Årzén, P. Faure, G. Fohler, M. Mattavelli, A. Neundorf, V. R. Segovia, and S. Schorr, "D1f: Interface Specification", Jan. 2011. `http://www3.control.lth.se/user/karlerik/Actors/M36/d1f-main.pdf`

K.-E. Årzén, V. R. Segovia, E. Bini, J. Eker, G. Fohler, and S. Schorr, "D3a: State Abstraction", January 2011. `http://www3.control.lth.se/user/karlerik/Actors/M36/d3a-main.pdf`

K.-E. Årzén, V. R. Segovia, M. Kralmark, A. Neundorf, S. Schorr, R. Guerra, and G. Fohler, "D3b: Resource Manager", January 2011. `http://www3.control.lth.se/user/karlerik/Actors/M36/d3b-main.pdf`

K.-E. Årzén, G. Fohler, V. R. Segovia, S. Schorr, "D3c: Resource Framework", January 2011. http://www3.control.lth.se/user/karlerik/Actors/M36/d3c.pdf

1.4 Contributions

This thesis contains the following contributions:

- A feedforward algorithm that assigns service levels to applications according to their bandwidth requirements, the QoS provided at each service level, and their relative importance values.
- Different policies for performing the bandwidth distribution of an application on a multicore platform.
- Bandwidth controllers that dynamically adapt the allocated CPU resources based on resource utilization and/or achieved QoS feedback, and that derive at runtime tuned models of the applications.

2

Background

Embedded systems play an important role in a very large proportion of advanced products designed in the world. A surveillance camera or a cell phone are classical examples of embedded systems in the sense that they have limited resources in terms of memory, CPU, power consumption, etc, but still they are highly advanced and very dynamic systems.

Different types of applications may execute in these systems. Basically they can be distinguished based on their real-time requirements as hard real-time applications and soft real-time applications. Hard real-time applications are those where missing one deadline may lead to a fatal failure of the system, so temporal and functional feasibility of the system must be preserved even in the worst case. On the other hand, for soft real-time applications failure to meet a deadline does not necessarily lead to a failure of the system, the meeting of deadlines is desirable for performing reasons.

Well-developed scheduling theory is available to determine whether an application can meet all its deadlines or not. If sufficient information is available about worst-case resource requirements, for instance worst-case execution times (WCET), then the results from classical schedulability theory can be applied.

Fixed Priority Scheduling with preemption is the most common scheduling method. Tasks are assigned priorities and at every point in time the ready task with the highest priority runs. The priorities assignment can be done using Rate Monotonic Scheduling (RMS). For RMS the tasks priorities are assigned according to their periods, the

smaller the period the higher the priority. Schedulability is guaranteed as long as the processor utilization U is below 0.69 [2]. For overload conditions low priority tasks can suffer from starvation, while the highest priority task has still guaranteed access to the processor. Fixed Priority Scheduling is supported by almost all available real-time operating systems.

There are also multiple dynamic priority scheduling algorithms. In these algorithms the priorities are determined at scheduling time. An example of such scheduling algorithm is Earliest Deadline First (EDF). For EDF the ready task with the earliest deadline is scheduled to run. EDF can guarantee schedulability up to a processor utilization of 1.0 [2], which means that it can fully exploit the available processing capacity of the processor. Under overload conditions there are no guarantees that tasks will meet their deadlines. EDF is implemented in several research operating systems and scheduling frameworks.

2.1 Threads versus Reservations

Today most embedded systems are designed and implemented in a very static fashion, assigning resources using priorities and deadlines, and with a very large amount of testing. The fundamental problem with state-of-the-art technologies such as threads and priorities is the lack of behavioral specifications and relations with resource demands.

For advanced embedded systems third party software has come to play an important role. However, without a proper notion of resource needs and timing constraints, integration of real-time components from several different vendors into one software framework is complicated. Threads and priorities do not compose [3], and even worse, priorities are global properties, possibly causing completely unrelated software components to interfere.

Resource reservations techniques constitute a very powerful mechanism that addresses the problems described above. It enforces temporal isolation and thereby creates groups of threads that have the properties of the atomic thread. This removes the need to know the structure of third party software.

Resource Reservation Techniques

In order to be able to guarantee timely behavior for real-time applications, it is necessary to shield them from other potentially misbehaving applications. One approach is to use resource reservations to isolate tasks from each other.

Resource reservation techniques implement temporal protection by reserving for each task τ_i a specified amount of CPU time Q_i in every interval P_i. The term Q_i is also called maximum budget, and P_i is called reservation period.

There are different reservation based scheduling algorithms, for instance the Constant Bandwidth Server (CBS) [4, 5], which is based on EDF, the Weighted Fair Scheduling [6], which has its origins in the networking field and also the Lottery scheduling [7], which has a static approach to reservations.

The Constant Bandwidth Server The Constant Bandwidth Server (CBS) is a reservation based scheduling method, which takes advantage of dynamic priorities to properly serve aperiodic requests and better exploit the CPU.

A CBS server S is characterized by the tuple (Q^S, P^S), where Q^S is the server maximum budget and P^S is the server period. The server bandwidth is denoted as U^S and is the ratio Q^S/P^S. Additionally, the server S has two variables: a budget q^S and a deadline d^S.

The value q^S lies between 0 and Q^S, it is a measure of how much of the reserved bandwidth the server has already consumed in the current period P^S. The value d^S at each instance is a measure of the priority that the algorithm provides to the server S at each instance. It is used to select which server should execute on the CPU at any instance of time.

Consider a set of tasks τ_i consisting of a sequence of jobs $J_{i,j}$ with arrival time $r_{i,j}$. Each job is assigned a dynamic deadline $d_{i,j}$ that at any instant is equal to the current server deadline d^S. The algorithm rules are defined as follow:

- At each instant a fixed deadline $d_{S,k} = r_{i,j} + P^S$ with $d_{S,0} = 0$ and a server budget $q^S = Q^S$ is assigned.
- The deadline $d_{i,j}$ of $J_{i,j}$ is set to the current server deadline $d_{S,k}$. In case the server deadline is recalculated the job deadline is also

recalculated.

- Whenever a served job $J_{i,j}$ executes, q^S is decreased by the same amount.
- When q^S becomes 0, the server variables are updated to $d_{S,k} = r_{i,j} + P^S$ and $q^S = Q^S$.
- In case $J_{i,j+1}$ arrives before $J_{i,j}$ has finished, then $J_{i,j+1}$ is put in a FIFO queue.

Hard CBS A problem with the CBS algorithm is that it has a soft reservation replenishment rule. This means that the algorithm guarantees that a task or job executes at least for Q^S time units every P^S, allowing it to execute more if there is some idle time available. Such kind of rule does not allow hierarchical scheduling, and is affected by some anomalies in the schedule generated by problems like the Greedy Task [8] and the Short Period [9].

A hard reservation [10, 8, 9] instead is an abstraction that guarantees the reserved amount of time to the server task or job, such that the task or job executes at most Q^S units of time every P^S.

Consider $q^r_{i,j}$ as the remaining computational need for the job $J_{i,j}$ once the budget is exhausted. The algorithm rules are defined as follow:

- Whenever $q^r_{i,j} \geq Q^S$, the server variables are updated to $d_{S,k+1} = d_{S,k} + P^S$ and $q^S = Q^S$.
- On the other hand if $q^r_{i,j} < Q^S$, the server variables are updated to $d_{S,k+1} = d_{S,k} + q^r_{i,j}/U^S$ and $q^S = q^r_{i,j}$.

In general resource reservation techniques provide a more suitable interface for allocating resources such as CPU to a number of applications. According to this method, each application is assigned a fraction of the platform capacity, and it runs as if it were executing alone on a less performing virtual platform [11], independently of the behavior of the other applications. In this sense, the temporal behavior of each application is not affected by the others and can be analyzed in isolation.

A virtual platform consists of a set of *virtual processors* or reservations, each of them executing a portion of an application. A virtual processor is a uni-processor reservation characterized by a bandwidth

$\alpha \leq 1$. The parameters of the virtual processor are derived as a function of the computational demand to meet the application deadline.

Hierarchical Scheduler

When using resource reservation techniques such as the Hard CBS, the system can be seen as a two-level hierarchical scheduler [12] with a global scheduler and local schedulers. Figure 2.1 shows the structure of a hierarchical scheduler.

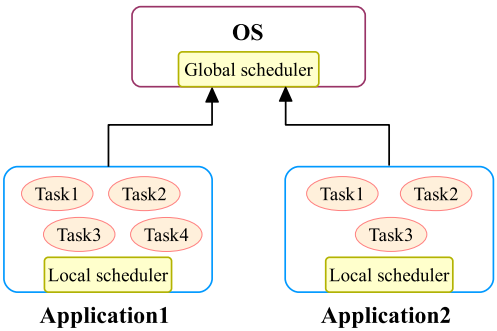


Figure 2.1: Hierarchical scheduler structure.

The global scheduler that is at the top level selects which application is executed next and for how long. Thus, it assigns each application a fraction of the total processor time distributed over the time line according to a certain policy. The local scheduler that belongs to each application selects which task is scheduled next.

In particular for two-level hierarchical scheduler the ready queue has either threads or servers, and the servers in turn contain threads or servers (for higher level schedulers).

2.2 Adaptivity in Embedded Systems

The need for adaptivity in embedded systems is becoming more pressing with the ongoing evolution towards much richer feature sets and demands for sustainability. Knowing the exact requirements of different applications at design time is very difficult. From the application

side, the resource requirements may change during execution. Tasks sets running concurrently can change at design time and runtime, this could be the result of changes in the required feature set or user installed software when deployed. From the system side, the resource availability may also vary at runtime. The systems can be too complex to know everything in detail, this implies that not all software can be analyzed. As a result, the overall load of the system is subject to significant variations, which could degrade the performance of the entire system in an unpredictable fashion.

Designing a system for worst-case requirements is in many cases not economically feasible, for instance in consumer electronics, mobile phones, etc. For these systems, using the classical scheduling theory based on worst-case assumptions, a rigid offline design and a priory guarantees would keep resources unused for most of the time. As a consequence, resources that are already scarce would be wasted reducing in this way the efficiency of these systems.

In order to prevent performance and efficiency degradation, the system must be able to react to variations in the load as well as in the availability of resources. Adaptive real-time systems addresses these issues. Adaptive real-time systems are able to adjust their internal strategies in response to changes in the resource availability, and resource demands to keep the system performance at an acceptable level.

Adaptive Resource Management

Adaptivity can be achieved using methods for managing CPU resources together with feedback techniques. The management algorithms can range from simple such as the adaption of task parameters like the task periods, to highly sophisticated and more reliable frameworks that utilize resource reservation techniques. The use of virtualization techniques such as the resource reservation-based scheduling provide spatial and temporal separation of concerns and enforce dependability and predictability. Reservations can be composed, are easier to develop and test, and provide security support, making them a good candidate to manage CPU resources. The feedback techniques provide the means to evaluate and counteract if necessary the consequences of the scheduling decisions made by the management methods.

In order to be able to adapt to the current state of the resource requirements of the application as well as the resource availability

of the system, the current state must be known. Thus, *sensors* are required to gather information such as the CPU resource utilization, deadline misses, etc. This information is then used to influence the operation of the system using *actuators*, which can be task admission control, modification of task weights or priorities, or modification of reservation parameters such as the budget/bandwidth and the period. These schemes resemble a control loop with sensors, actuators and a plant which is to be controlled.

There are a variety of approaches how to apply control theory to scheduling [13, 14, 15]. Of particular interest is feedback control in combination with resource reservation techniques. The motivation behind this is the need to cope with incorrect reservations, to be able to reclaim unused resources and distribute them to more demanding tasks, and to be able to adjust to dynamic changes in resource requirements. Hence, a monitoring mechanism is needed to measure the actual demands and a feedback mechanism is needed to perform the reservation adaptation.

2.3 Multicores

The technology improvements in the design and development of microprocessors has always aimed at increasing their performance from one generation to the next. Initially for single processors the tendency was to reduce the physical size of chips, this implied an increment in the number of transistors per chip. As a result, the clocks speeds increased producing a dangerous level of heat dissipation across the chip [16].

Many techniques are used to improve single core performance . In the early nineties performance was achieved by increasing the clock frequency. However, processor frequency has reached a limit. Other techniques include superscalar processors [17] that are able to issue multiple instructions concurrently. This is achieved through pipelines where instructions are pre-fetched, split into sub-components and executed out-of-order. The approach is suitable for many applications, however it is inefficient for applications that contain code difficult to predict. The different drawbacks of these techniques, the increased available space, and the demand for increased thread level parallelism [18] for which many applications are better suited led to the development of

multicore microprocessors.

Nowadays performance is not only synonym of higher speed, but also of power consumption, temperature dissipation, and number of cores. Multicore processors are often run at slower frequencies, but have much better performance than a singlecore processor. However, with increasing the number of cores comes issues that were previously unforeseen. Some of these issues include memory and cache coherence as well as communication between the cores.

Multicore Scheduling Algorithms

One of the large challenges of multicore systems, is that the scheduling problem now consists of both mapping the tasks to a processor and scheduling the tasks within a processor. There are still many open problems regarding the scheduling issues in multicore systems. Analyzing multiprocessor systems is not an easy task. As pointed out by Liu [19]: "few of the results obtained for a single processor generalize directly to the multiple processor case: bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors".

An application can be executed over a multicore platform using partitioned or global scheduling algorithm. For partitioned scheduling any task of the application is bound to execute on a given core. The problem of distributing the load over the computing units is analogous to the bin-packing problem, which is known to be NP-hard [20]. There are good heuristics that are able to find acceptable solutions [21, 22, 23, 24]. However, their efficiency is conditioned by their computational complexity, which is often too high.

For global scheduling any task can execute on any core belonging to the execution platform. This option is preferred for highly varying computational requirements. With this method, there is a single system-wide queue from which tasks are extracted and scheduled on the available processors.

Multicore Reservations

Multicore platforms also need resource reservation techniques, according to which the capacity of a processor can be partitioned into a set

of reservations. The idea behind *multicore reservation* is the ability to reserve *shares* of a multicore platform, so that applications can run in isolation without interfering on each other. Despite the simple formulation of the problem, the multicore nature of the problem introduces a considerably higher complexity than the singlecore version of the problem.

2.4 Linux

The Linux scheduler is a priority based scheduler that schedules tasks based upon their static and dynamic priorities. Each time the Linux scheduler runs, every task on the run queue is examined and its goodness value is computed. This value results from the combination of the static and dynamic priorities of a task. The task with the highest goodness is chosen to run next. Ties in goodness result in the task that is closest to the front of the queue running first.

The Linux scheduler may not be called for intervals of up to 0.4 seconds when there are compute bound tasks running. This means that the currently running task has the CPU to itself for periods of up to 0.4 seconds, this will also depend upon the priority of the task and whether it blocks or not. This is convenient for throughput since there are few computationally unnecessary context switches. However, this can destroy interactivity because Linux only reschedules when a task blocks or when the dynamic priority of the task reaches zero. As a result, under the Linux default priority based scheduling method, long scheduling latencies can occur.

Linux Trends in Embedded Systems

Traditionally embedded operating systems have employed proprietary software, communication protocols, operating systems and kernels for their development. The arrival of Linux has been a major factor in changing embedded landscape. Linux provides the potential of an open multivendor platform with an exploding base of software and hardware support.

The use of embedded Linux mostly for soft real-time applications but also for hard ones, has been driven by the many benefits that it

provides with respect to traditional proprietary embedded operating systems. Embedded Linux is a real-time operating system that comes with royalty-free licenses, advanced networking capabilities, a stable kernel, support base, and the ability to modify and redistribute the source code.

Developers are able to access the source code and to incorporate it into their products with no royalty fees. Many manufacturers are providing their source code at no cost to engineers or other manufacturers. Such is the case of Google with its Android software for cellular phones available for free to handset makers and carriers who can then adapt it to suit their own devices.

As further enhancements have been made to Linux it has quickly gained momentum as an ideal operating system for a wide range of embedded devices scaling from PDAs, all the way up to defense command and control systems.

2.5 Related Works

This section presents some of the projects as well as different research topics related to the ACTORS project and consequently to this work.

The MATRIX Project

The Matrix [25, 26] project has developed a QoS framework for real-time resource management of streaming applications on heterogeneous systems. The Matrix is a concept to abstract from having detailed technical data at the middleware interface. Instead of having technical data referring to QoS parameters like: bandwidth, latency and delay, it only has discrete portions that refer to levels of quality. The underlying middleware must interpret these values and map them on technical relevant QoS parameters or *service levels*, which are small in number such as *high*, *medium*, *low*.

The FRESCOR Project

The European Frescor [27] project has developed a framework for real-time embedded systems based on contracts. The approach integrates advanced flexible scheduling techniques provided by the AQuoSA [28]

scheduler directly into an embedded systems design methodology. The target platform is singlecore processor. The bandwidth adaptation is layered on top of a soft CBS server. It is achieved by creating a contract model that specifies which are the application requirements with respect to the flexible use of the processing resources in the system. The contract also considers the resources that must be guaranteed if the component is to be installed into the system, and how the system can distribute any spare capacity to achieve the highest usage of the available resources.

Other Adaptive QoS Frameworks

Comprehensive work on application-aware QoS adaptation is reported in [29, 30]. Both approaches separate between the adaptations on the system and application levels. Architectures like [29] give an overall management system for end-to-end QoS, covering all aspects from user QoS policies to network handovers. While in [29] the application adjustment is actively controlled by a middle-ware control framework, in [30] this process is left to the application itself, based on requests from the underlying system.

Classical control theory has been examined for QoS adaptation. [31] shows how an application can be controlled by a task control model. The method presented in [32] uses control theory to continuously adapt system behavior to varying resources. However, a continuous adaptation maximizes the global quality of the system but it also causes large complexity of the optimization problem. Instead, we propose adaptive QoS provision based on a finite number of discrete quality levels.

The variable-bandwidth servers proposed in [33] integrate directly the adaptation into the bandwidth servers. Resource reservations can be provided also using other techniques than bandwidth servers. One possibility is to use hypervisors [34], or to use resource management middleware or resource kernels [10]. Resource reservations are also partly supported by the mainline Linux completely fair scheduler or CFS.

Adaptivity with respect to changes in requirements can also be provided using other techniques. One example is the elastic task scheduling [35], where tasks are treated as springs that can be compressed in order to maintain schedulability in spite of changes in task rate. Another possibility is to support mode changes through different types

of mode change protocols [36]. A problem with this is that the task set parameters must be known both before and after the change.

3

Resource Manager Overview

3.1 Overall Structure

In ACTORS the main focus was automatic allocation of available CPU resources to applications not only at design time, but also at runtime, based on the demands of the applications as well as the current state of the system. In order to do this, ACTOR proposes a software architecture [37] consisting of three layers:

- The application layer
- The scheduler layer
- The resource manager layer

Figure 3.1 shows the overall structure of the ACTORS software architecture. The resource manager is a key component in the architecture, that collects information from the other layers through interfaces, and makes decisions based on this information and the current state of the system.

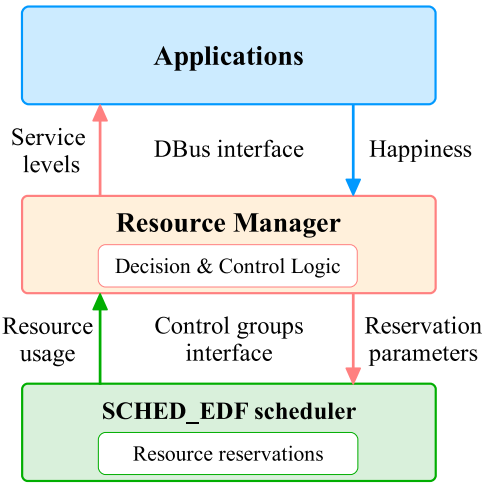


Figure 3.1: Overall structure of the ACTORS software architecture.

3.2 Application Layer

The ACTORS application layer will typically contain a mixture of different application types. These applications will have different characteristics and real-time requirements. Some applications will be implemented in the dataflow language CAL whereas others use conventional techniques.

In general, it is assumed that the applications can provide support for resource and quality adaption. This implies that an application supports one or several service levels, where the application consumes different amount of resources at each service level. Applications supporting several service levels are also known as adaptive applications. On the other hand, applications which support only one service level are known as non-adaptive applications.

Applications which register and work together with the resource manager are defined as ACTORS-aware applications, these applications can be adaptive or non-adaptive. Applications which do not provide any information to the resource manager are defined as ACTORS-unaware applications, these applications are non-adaptive.

CAL Applications

A CAL application is an application written in CAL [38], which is a dataflow and actor-oriented language. An actor is a modular component that encapsulates its own state, and interacts with other actors through input and output ports. This interaction with other actors is carried out asynchronously by consuming (reading) input tokens, and producing (writing) output tokens. The output port of an actor is connected via a FIFO buffer to the input port of another actor. The computations within an actor are performed through firings, or actions which include consumption of tokens, modification of internal state, and production of tokens. A CAL network or network of actors is obtained by connecting actor input and output ports. Figure 3.2 illustrates the structure of a CAL application.

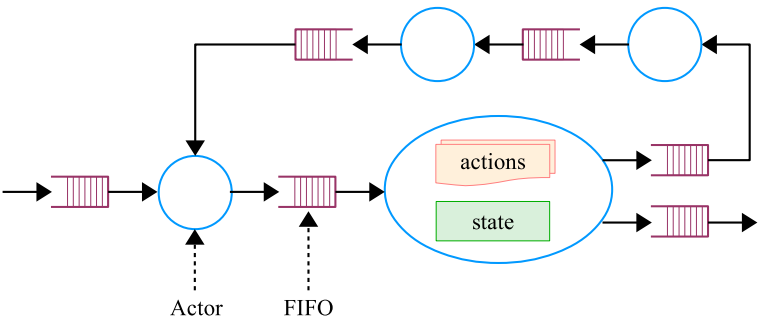


Figure 3.2: CAL application.

A CAL network can correspond to a synchronous data flow (SDF) model [39], or a dynamic data flow (DDF) model [40]. For the first type of network the number of tokens consumed and produced during each firing is constant, making it possible to determine the firing order statically.

ACTORS distinguishes between dynamic and static CAL applications. In general dynamic CAL applications correspond to most multimedia streaming applications, where the execution is highly data-dependent. This makes it impossible to schedule the network statically. Static CAL applications contains actions with constant token consumption and production rates, for instance a feedback control application.

In this case the data flow graph can be translated into a static precedence relation.

The execution of a CAL application is governed by the *CAL run-time system*. The run-time system consists of two parts, the *actor activator* and the *run-time dispatcher*. The actor activator activates actors as input data becomes available by marking them as ready for execution. The dispatcher repeatedly selects an active actor in a round-robin fashion and then executes it until completion.

The run-time system assumes that the actor network is statically partitioned. For each partition there is a thread that performs the actor activation and dispatching.

The run-time is not only responsible for the execution of the CAL actors within applications, but also of the *system actors*. A system actor is an actor that is implemented directly in `C`. The purpose of these actors is to provide a means for communication between the CAL application and the external environment. System actors are used for input-output communication, for access to the system clock, and for communication with the resource manager. Normally each system actor has its own thread.

Legacy Applications

A legacy applications is an ACTORS-unaware application. This means that it is not necessary for the application to modify its internal behavior based on which service level that it executes under, and hence its resource consumption.

The current way of executing a legacy application is through the use of a wrapper. The wrapper enables the resource manager to handle a legacy application as an application with one or several service levels and one virtual processor. The wrapper periodically checks if any application threads have been created or deleted and adds or removes those from the virtual processor.

3.3 Scheduler Layer

The scheduler is the kernel component which schedules and allocates resources to each process according to a scheduling policy or algorithm.

As one of the important parts of the kernel, its main job is to divide the CPU resources among all active processes on the system.

In order to fit the requirements specified by the ACTORS architecture, the scheduling algorithm needs to implement a resource reservation mechanism [41, 42] for CPU time resources.

According to the resource reservation mechanism, each application is assigned a fraction of the platform capacity, and it runs as if it were executing alone on a slower *virtual platform* (see Figure 3.1), independently of the behavior of other applications. A virtual platform consists of a set of *virtual processors*, each executing a part of an application. A virtual processor is parametrized through a budget Q_i and a period P_i. In this way, the tasks associated with the virtual processor execute for an amount of time equal to Q_i every period P_i.

SCHED_EDF

SCHED_EDF [43] is a new real-time scheduling algorithm that has been developed within the ACTORS project. It is a hierarchical partitioned EDF scheduler for Linux where SCHED_EDF tasks are executed at the highest level, and ordinary Linux tasks at the secondary level. This means, that ordinary tasks may only execute if there is no SCHED_EDF tasks that want to execute.

The SCHED_EDF provides support for reservations or virtual processors through the use of hard CBS (Constant Bandwidth Server). A virtual processor may contain one or several SCHED_EDF tasks.

Some of the characteristics of SCHED_EDF are:

- SCHED_EDF allows the creation of virtual processors for periodic and non periodic process.
- SCHED_EDF permits the modification of virtual processors parameters.
- SCHED_EDF provides support for multicore platforms.
- SCHED_EDF has a system call that allows to specify in which core the process should execute.
- SCHED_EDF reports the resource usage per virtual processor to userspace.

- `SCHED_EDF` allows the migration of virtual processors between cores at runtime.

The last characteristic allows monitoring the resource usage of the threads executing within a virtual processor. This information can be used by the resource manager in order to redistribute the CPU resource among the applications if necessary.

3.4 Resource Manager Layer

The resource manager constitutes the main part of the ACTORS architecture. It is a user space application, which decides how the CPU resources of the system should be distributed among the applications. The resource manager interacts with both the application and the scheduler layer at run-time, this interaction allows it to gather information from the running applications as well as from new applications that would like to execute on the system, and to be aware of the current state of the system.

The resource manager communicates with the applications using a D-Bus [44] interface, which is a message bus system that enables applications on a computer to talk to each other. In the case of the scheduler, the resource manager communicates using the control groups API of Linux. Here, the control groups provide a mechanism for aggregating partitioned sets of tasks, and all their future children, into hierarchical groups with specialized behavior.

The main tasks of the resource manager are to accept applications that want to execute on the system, to provide CPU resources to these applications, to monitor the behavior of the applications over time, and to dynamically change the resources allocated during registration based on the current state of the system, and the performance criteria of the system. This is the so called resource adaptation.

Figure 3.3 shows in more detail the structure of the ACTORS architecture. Here, the resource manager has two main components, a global supervisor, and several bandwidth controllers. The supervisor implements feedforward algorithms which allow the acceptance, or registration, of applications. The bandwidth controllers implement a feedback algorithm, which monitors the resource consumption of the running

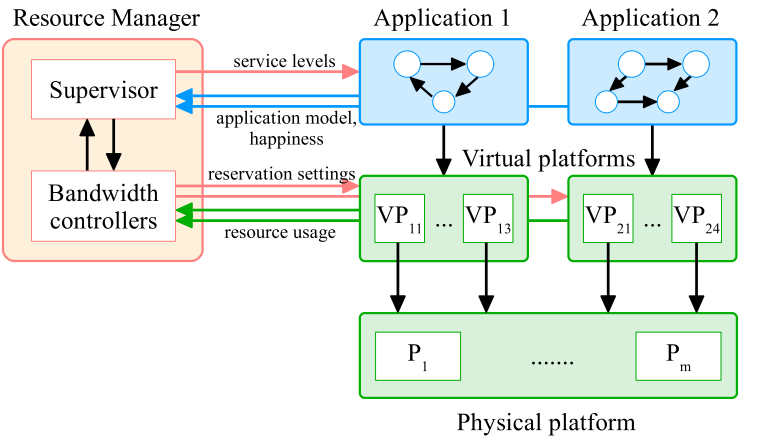


Figure 3.3: ACTORS software architecture

applications, and dynamically redistributes the resources if necessary. A detailed description of these two components will be done in Chapters 5, 6 and 7.

Resource Manager Implementation

The resource manager is implemented in `C++`. It consists of two threads which themselves are `SCHED_EDF` tasks executing within a fixed-size virtual processor within core 0. The resource manager communicates with the applications through a D-Bus interface and with the underlying `SCHED_EDF` using the control groups API of Linux. The first thread handles incoming D-Bus messages containing information provided by the applications. The second thread periodically samples the virtual processors, measures the resource consumption, and invokes the bandwidth controllers.

3.5 Assumptions and Delimitations

The current version of the resource manager makes a number of assumptions and have several limitations. These are summarized below.

Homogeneous Platform: The resource manager assumes that the execution platform is homogeneous, that is, all cores are identical and that it does not matter on which core that a virtual processor executes. In reality this assumption rarely holds. Also, in a system where the cores are identical, it is common that the cores share L2 caches pairwise. This is for example the case for x86-based multicore architectures. A consequence of this is that if we have two virtual processors with a large amount of communication between them it is likely that the performance, for instance, throughput, would be better if they are mapped to two physical cores that share cache. This is, however, currently not supported by the resource manager.

Single Resource Management: The current version of the resource manager only manages the amount of CPU time allocated to the applications, that is, a single resource. Realistic applications also require access to other resources than the CPU, for example memory. However, in some sense the CPU is the most important resource, since if a resource does not receive CPU time it will not need any other resource.

Temporal isolation: The `SCHED_EDF` operating system supports temporal isolation through the use of constant bandwidth servers. However, `SCHED_EDF` currently does not support reservation-aware synchronization protocols, for instance, bandwidth ceiling protocols [45]. Thus, temporal isolation is not guaranteed for threads that communicate with other threads, Synchronization is currently implemented using ordinary POSIX mutex locks. One example of this is the mutual exclusion synchronization required for the FIFO buffers in the CAL dataflow applications.

Best Effort Scheduling: Although the resource management framework can be used also for control applications as will be described in Chapter 10 is has primarily been developed for multimedia application which commonly have soft real-time requirements and are focused on maximizing the throughput. The underlying operating system, that is, Linux together with `SCHED_EDF` is not altogether well-specified. A consequence of this is that the scheduling approach adopted is best effort scheduling.

4

Resource Manager Inputs and Outputs

The communication between the different layers of the ACTORS architecture is based on interfaces between the layers. The information flowing through these interfaces has different characteristics, but in general one can distinguish between static and dynamic information. Considering that the resource manager is the key element of the architecture, it also constitutes the pivot from where the information flows in or out to the other layers.

4.1 Static Inputs

Static inputs include information which is not considered to change during runtime, or at least not very often. This information is mainly provided by the application at registration time, and the developer at system start time.

Service Level Table

In order to be able to run or execute in the ACTORS software architecture, every application must register with the resource manager. The registration allows the resource manager to be aware of the resource requirements, quality of service, and structure of the applications running on the system. These particular characteristics of each application are described in the service level table.

The service level table provides information about the different service levels supported by the applications. Additionally it specifies the resource requirements and the quality of service that can be expected at each service level.

All the values in the service level table are expressed as integer values. The service level index is a number that can take any value beginning from 0, where 0 corresponds to the highest service level. The quality of service or QoS, takes values between 0 and 100. It corresponds to the QoS that can be expected at a certain service level. The resource requirements are specified as a tuple consisting of two parameters: the bandwidth, and the time granularity. The bandwidth is an indicator of the amount of resources required by an application, but it is not enough to capture all of the time properties of an application. These properties can be included in the service level table through the time granularity value. This value provides the time horizon within which the resources are needed. The time granularity is expressed in micro seconds [μs].

The service level table may include information about how the total bandwidth should be distributed among the individual virtual processors of the application for each service level. These values are also known as the bandwidth distribution or BWD. The bandwidth distribution values may be absolute or relative. If it is relative then the bandwidth distribution values for each service level sums to 100, whereas if it is absolute then it sums to the total bandwidth.

Additionally to the service levels supported by each application, an extra service level is automatically added to all applications when they register. This service level is know as the extra service level or x. The resource requirements at this service level are the lowest that can be assigned during registration. The functionality of this service level will be explained in Chapter 5.

Table 4.1 shows the service level table for an application named A1. The table contains the service level index (SL), the quality of service (QoS), the bandwidth (BW), the time granularity, and the bandwidth distribution (BWD). In the table at service level 0 the application A1 provides a QoS of 100%. The total bandwidth required and the granularity at this service level correspond to 200% and 50μs respectively. The total bandwidth is evenly splited among the four virtual processors that contain the application tasks, this is expressed by the bandwidth

distribution values.

Table 4.1: Service level table of application A1

Application	SL	QoS [%]	BW [%]	Granularity [μs]	BWD [%]
A1	0	100	200	50	[50, 50, 50, 50]
	1	80	144	90	[36, 36, 36, 36]
	2	50	112	120	[28, 28, 28, 28]
	3	30	64	250	[16, 16, 16, 16]
	x	1	4	100000	[1, 1, 1, 1]

The values defined in the service level table of each application, except for the extra service level x, are specified by the application developer, and can be seen as an initial model of the application. How certain or trustful these values are is something that can be evaluated by the different algorithms implemented by the resource manager first after the application has been executing for some period of time.

Importance Values

The application importance specifies the relative importance or priority of an application with respect to others. The importance values only play a role when the system is overloaded, that is, when it is not possible for all registered applications to execute at their highest service level.

The importance is expressed as a non-negative integer value and it is specified by the system developer. In case the value is not explicitly specified which is the most common case, the resource manager provides a default importance value of 10.

Table 4.2 shows an example of an importance table, which has three applications. The highest value represents the highest importance.

The importance values are provided in a file that is read by the resource manager during start up.

Number of Virtual Processors

The number of virtual processors is a value provided implicitly through the bandwidth distribution. For the resource manager this value is an

Table 4.2: Importance table

Application	Importance
mplayer	100
tetris	75
firefox	10

indicator of the topology of the application. The number can be greater than the number of online physical cores of the system.

Thread Groups

In addition to the service level table each application also needs to provide information about how many thread groups it consists of, and which threads that belong to these groups. Each thread group will eventually be executing within a separate virtual processor.

4.2 Dynamic Inputs

Dynamic inputs includes online information about the state of the allocated resources, that is, how they are being used, and about the level of satisfaction obtained with the allocated resources. This information is provided by the scheduler and the application layers.

Used Budget

The used budget value is the cumulative budget used by the threads in each of the virtual processors of an application since its creation. This value is measured in nano seconds.

Exhaustion Percentage

The exhaustion percentage value is the cumulative number of server periods that the virtual processor budget has been completely consumed. A high value indicates that the application was throttled by the CBS server and that it is likely that is requires more bandwidth.

Cumulative Period

The cumulative period value represents the total number of server periods fully elapsed, that is, the number of times that the deadline of the reservation has been postponed.

Together the used budget, the exhaustion percentage, and the cumulative period, provide information about the state of the resources allocated to each application, that is how they are being used by the application.

The used budget, the exhaustion percentage, and the cumulative period values are provided by the scheduler layer, and are read periodically by the resource manager, with a sampling period that is a multiple of the period of each running application.

Happiness

The happiness value represents the level of satisfaction, or the perceived quality, obtained with the allocated resources at a specific service level. The value is provided to the resource manager only by applications which implement mechanisms that monitor their quality, and determine whether it corresponds to what can be expected for the current service level.

For simplicity the happiness value is a binary value, that is, it can only take one of two values, 0 which means that the application is not able to provide the quality of service promised at the current service level, and 1 otherwise. Unless the application reports that it is unhappy with the allocated resources, the resource manager assumes that the application is happy.

4.3 Dynamic Outputs

Dynamic outputs include online parameters produced by the resource manager, which are provided to the application and the scheduler layer.

Assigned Service Level

The assigned service level value is used to inform an application at which service level it must execute.

The assigned service level value of each running application is generated by the resource manager, based on the service level table provided during registration time, the current state of the system, and the system objective. A more detailed description of the algorithm used to calculate this value will be part of Chapter 5.

Assigned Server Budget and Period

The assigned server budget and server period parametrize each virtual processor created by the resource manager. The assigned server budget defines the maximum budget that can be used by the application tasks running inside a virtual processor every period.

The period is directly given in the service level table of each application through the timing granularity value. It may depend on the service level. The assigned server budget value is initially defined by the resource manager at the creation of the virtual processor, that is, at the registration of a new application, and redefined whenever the algorithms inside the resource manager consider that the assigned server budget does not match the real resource needs of the application process. Chapters 6 and 7 will provide more information about when the assigned server budget is calculated, and under which conditions it can be recalculated.

Affinity

The affinity value decides to which physical processor a virtual processor should be allocated. Considering that the ACTORS software architecture is mainly oriented to multicore systems, there are several ways how the resource manager can specify these values. A more detailed description about the algorithm used to set the affinity value can be found in Chapter 6.

5

Service Level Assignment

One of the objectives of the architecture proposed by ACTORS (see Figure 3.3) is to be able to optimally distribute the CPU resources among the running applications. This distribution must be done systematically according to a *performance criteria*, which defines what optimality means, and to *policies* that specify when and how this must be done.

The resource manager plays a key role in this distribution because it is able to dynamically communicate with the applications and the scheduler layer. Thus, it is aware at any time of the resource requirements of the running applications as well as the availability of system resources. This places the resource manager in a position of decision maker in the system, with the ability of implementing algorithms that provide the desired optimal distribution.

5.1 Problem Description

To carry out the distribution of CPU resources the resource manager needs to define some rules or policies. They specify who can take part in the distribution, when it should take place, and which are the minimum requirements on the information that must be provided. The policies specify that:

- Only accepted applications or the ones in process of being accepted by the resource manager can take part in the distribution. The process of acceptance is also known as registration.

- The registration is the first step that must be done by every application that wants to run on system.

- The distribution is executed when an application registers, unregisters, and whenever the performance criterion as well as the system conditions require it.

- The unregistration takes place when the application has finished its execution and therefore it does not need to use the resources of the system anymore.

- The information required includes the importance table, the service level tables of the applications, and the performance criteria.

In addition to these policies a performance criterion must be considered. This criterion defines the optimality of the distribution. One criterion could be to maximize the quality of service provided by the system, although this could sound like a misconceived idea since the quality of service is a relative measurement. Consider for instance the experience perceived by the user when running different applications. In such a case the system quality of service can be interpreted as the sum of the quality of service of the different running applications.

Another criterion could be to save energy, in systems such as mobile phones this could be an important issue. This would relegate the quality of service provided to a second place. However, a certain quality of service that matches the energy constraints must be guaranteed. For the purpose of the present work, only the first criterion will be considered.

The starting point of the resource distribution is when an application registers with the resource manager. At this moment the application provides its resource requirements through its service level table. It is then the task of the resource manager to allocate resources to the application.

The quality of service as well as the consumed resources are directly associated with the service levels that an application supports. Thus, the best way to allocate resources to the application would be to define the service level at which it must execute. This is also known as the service level assignment of an application.

5.2 BIP Formulation

The problem previously described can be formulated as an optimization problem. The objective of this optimization is to select the service level of each application so that the weighted sum of the quality of service is maximized. This optimization problem is subject to the constraint that the total CPU resources are limited.

Since from all the service levels provided by each application only one will be assigned, the problem formulation can be done such that the decision variables represent the selection of a particular service level. Additionally, the constraint defined by the maximum assignable CPU resources on the system can be expressed as a linear combination of the decision variables.

The particular characteristics of the formulation described place it in the category of a Binary Integer Programming (BIP) Problem [46]. This is a special case of integer linear programming, which constrains the decision variables to be binary. In general a BIP problem can be formulated as follows:

$$\begin{aligned} \min \quad & \{c^T x : x \in P \cap X\} \\ & Ax \leq b \\ & Gx = d \\ & x_i \in \{0,1\}, \quad i = 1,\ldots,n \end{aligned} \tag{5.1}$$

where A and G are real coefficient matrices of dimensions mxn and pxn respectively. The objective and the constraints are affine functions. The feasible set of the BIP problem is specified by $P \cap X$, where P is a given polyhedron, and X is a combinatorial discrete set that are defined as:

$$\begin{aligned} P &:= \{x \in \Re^n : Ax \leq b, x \geq 0\} \\ X &:= \{x \in Z^n : 0 \leq x \leq 1\}, \quad X \subseteq Z^n \end{aligned}$$

BIP problems are convex optimization problems with a feasible finite set containing at most 2^n points. In general they can be very difficult to solve, but they can be efficiently solved under certain conditions such as when the constraint matrix is totally unimodular, and the

right-hand side vector of the constraints belongs to the integers. They can be solved using different algorithms, the performance of any particular method is highly problem-dependent. This methods include enumeration techniques, including the branch and bound procedure [47], cutting plane techniques [48], and group theoretic techniques [49].

Service Level Assignment Formulation

The service level assignment can now be formulated as a BIP problem [50]. The service level $j \in M = \{0, \ldots, SL_i - 1\}$, where SL_i is the number of service levels supported by application $i \in N = \{1, \ldots, n\}$, is represented as a column vector x_{ij} containing boolean variables, where the variable is 1 if the corresponding service level has been selected and 0 otherwise. The quality of service and the bandwidth of each application are represented by the row vectors q_{ij} and α_{ij} of corresponding size. The problem can now be formulated as follows:

$$\max \sum_{i=1}^{n} \sum_{j=0}^{m} w_i q_{ij} x_{ij} \tag{5.2}$$

$$\sum_{i=1}^{n} \sum_{j=0}^{m} \alpha_{ij} x_{ij} \leq C$$

$$\forall i, \sum_{j=0}^{m} x_{ij} = 1$$

In the formulation C corresponds to the total assignable bandwidth of the system, and w_i to the importance value of application i. The set N contains all the running applications which include the registered applications, as well as the one in process of registration. The set M contains the service levels supported by each application.

The first constraint guarantees that the total sum of the bandwidth at the assigned service level of each application does not exceed the total capacity of the system. The last constraint ensures that all applications get registered with the resource manager, this means that applications that have lower importance values, and do not contribute significantly to the performance criterion will be accepted to run at the default lowest service level x that is defined automatically by the resource manager.

Assigning the lowest service level x is a way for the resource manager to inform an application that it cannot meet its resource requirements. Then, it is up to the application to decide whether to proceed at the lowest service level with a very small amount of resources, or to terminate itself. Because of the presence of the service level x, the optimization problem in all practical situations always has a feasible solution.

The formulation in Equation 5.2 assumes that the resource manager accepts all applications that want to run on the system. In case this does not represent an important issue, the last constraint can be relaxed by changing it to an inequality constraint. Thus, the resource manager will be able to shut down some applications in order to allow the registration of applications with higher importance values.

5.3 Example

In this section a simple example is introduced to show how the service level assignment is performed. The scenario includes four applications named A1, A2, A3, and A4. For illustration reasons the importance value of the applications is shown as an extra column named I in the service level table of the applications.

Table 5.1 shows the service level tables of all the applications. One can observe that the applications support different number of service levels and have different resource requirements. All the applications except application A4 provide the BWD parameter, that is, it has more than one virtual processor.

Implementation Considerations

The physical platform employed is a four core machine. The BIP optimization problem is solved using the `GLPK` [51] linear programming toolkit. To ensure a proper behavior of the operating system 10% bandwidth of each processor is reserved for system applications including the 10% for the resource manager itself. Thus, 360% of the bandwidth is available to applications executing under the control of the resource manager.

Solving an ILP problem online in a real-time system may sound as a quite bad approach due to the potential inefficiency. However, in this

Table 5.1: Service level table of applications A1, A2, A3, and A4.

Application name	I	SL	QoS [%]	BW [%]	Granularity [μs]	BWD [%]
A1	10	0	100	200	50	[50, 50, 50, 50]
		1	90	150	90	[35, 35, 45, 35]
		2	70	100	120	[25, 25, 25, 25]
		3	60	50	250	[10, 10, 20, 10]
		x	1	4	100000	[1, 1, 1, 1]
A2	100	0	100	180	50	[60, 60, 60]
		1	80	140	90	[27, 27, 26]
		2	50	100	120	[17, 17, 16]
		x	1	3	100000	[1, 1, 1]
A3	1000	0	100	120	50	[30, 30, 30, 30]
		1	60	80	90	[20, 20, 20, 20]
		x	1	4	100000	[1, 1, 1, 1]
A4	200	0	100	100	50	
		1	90	90	90	
		2	60	60	120	
		x	1		100000	

case there are several factors that avoids this problem. The resource manager thread that performs the optimization is also executing within a `SCHED_EDF` reservation. Hence, it will not disturb applications that already have been admitted to the system, but will only delay the registration of the new application. Also, provided that the new application has been correctly implemented using a separate thread for the D-Bus communication, not even this application will be blocked. Instead it will continue executing under the normal Linux scheduling class during the registration process, provided that the `SCHED_EDF` threads do not consume all the CPU time. Also, the size of the optimization problem is quite limited. The largest application set so far used with the

resource manager is the control demonstrator described in Chapter 10. It consists of 8 applications with 2-4 service levels each. In this case the registration process takes 1-2 seconds.

Service Level Assignment

The result of the service level assignment is presented for two cases. In the first case no relaxation of the second constraint in Equation 5.2 is allowed, and in the second case relaxation is allowed.

At time t_0 no applications are running on the system. At time t_1 application A1 wants to execute on the system, therefore the registration process begins. After solving Equation 5.2 the resource manager assigns service level 0 to A1. At time t_2 application A2 begins the registration process. Since this application is more important than application A1, and it contributes significantly to the objective function, the resource manager assigns the highest service level to A2, and decreases the service level of A1 from 0 to 1. When application A3 registers at time t_3, the resource manager assigns service level 0 to A3 and A2 and reduces the service of A1 to 3.

The results are shown in Table 5.2. The table shows that the assigned service level for applications A1, A2, and A3 will be the same if relaxation of the constraint is considered or not. Additionally the table

Table 5.2: Service level assignment of applications A1, A2, A3, and A4 with and without relaxation of the second constraint of Equation 5.2.

	$\forall i, \sum x_{(i)} = 1$ or $\sum x_{(i)} \leq 1$				$\forall i, \sum x_{(i)} = 1$	$\forall i, \sum x_{(i)} \leq 1$
	t_0	t_1	t_2	t_3	t_4	t_4
A1	-	0	1	3	4	-
A2	-	-	0	0	2	1
A3	-	-	-	0	0	0
A4	-	-	-	-	0	0
W_{QoS}	-	1000	11500	110600	125000	128000
A_{BW}	-	200	330	350	325	360

shows the weighted quality of service (W_{QoS}) and the total assigned bandwidth (A_{BW}) after each service level assignment.

Depending on which constraint is employed different results in the service level assignment can be observed at time t_4 when application A4 registers with the resource manager. When the equality constraint is used, all four applications remain in the system, however, application A1 gets to execute at the default lowest service level that can be assigned for this application, that is, service level 4 that only provides 4% bandwidth of the system to the application, while application A2 gets service level 2. On the other hand if the inequality constraint is used, the resource manager will unregister application A1, and give service level 1 to A2.

A careful observation of the table at time t_4 shows that the weighted QoS is greater when the inequality constraint is used, this at the price of shutting down the application A1. Naturally when using the equality constraint application A1 will still be running on the system consuming a minimum amount of resources, this may imply a poor performance of the application, then again whenever applications A2, A3 or A4 finish their execution the application A1 will recover. A deeper evaluation of this behavior will be done in Chapter 7.

Advantages and Disadvantages of the Formulation

The formulation presented in Equation 5.2 is very simple, and uses little information to produce a solution. However, it is this lack of more detailed information which constitutes its weakest point. For instance consider the example previously explained. The solution of the problem did not consider the bandwidth distribution parameter, this parameter is very important specially when defining how each of the virtual processors of the applications must be assigned to each processor on the system. Although the maximum assignable bandwidth of the system is 360%, the maximum assignable bandwidth in each core is only 90%. Therefore the solution provided by Equation 5.2 is not necessarily schedulable.

Although this could look like a major drawback, one has to keep in mind that solving BIP problems can be very difficult and very time and resource consuming. Thus, the idea behind this formulation is "divide and conquer", first the resource manager will use this simple formulation to assign a possible service level, and later on with the

help of additional techniques will produce the final assigned service levels which respect all schedulability conditions.

6

Bandwidth Distribution

In the previous chapter it was mentioned that the optimal distribution of CPU resources among the running applications begins at registration time. At this point the resource manager assigns the service level at which each application present on the system must execute. This service level assignment is formulated as a BIP optimization problem. This formulation includes the new application that has requested the registration as well as the applications already registered.

After the service level assignment the resource manager is aware of the total amount of resources or bandwidth that each application requires. The next natural step would be to distribute the total bandwidth. This process is known as the bandwidth distribution and includes two subproblems.

The first subproblem is how the resource manager should divide the total bandwidth of an application between its virtual processors. This can be easily solved using the BWD values from the service level table in case they have been provided. Otherwise, the total bandwidth is split evenly between the virtual processors (VP) of the application.

The second subproblem is how the virtual processors should be mapped or distributed onto the physical cores. The complexity of this problem is increased by the multicore nature of the platform, and the particular partitioning of the applications (BWD). The resource manager handles this problem using different distribution policies.

6.1 Distribution Policies

There are different ways how the resource manager can map the virtual processors of an application onto the physical cores. Basically the resource manager implements two different policies, the balanced distribution, and the packed distribution.

Balanced Distribution

The balanced distribution policy is primarily developed for multimedia applications implemented using dataflow techniques. For multimedia applications the main objective is often to maximize throughput. In order to achieve this it is desirable that all the cores do productive work as much as possible and avoid unproductive work, for instance, context switching. Hence, the run time system used for these types of applications contains one thread per physical core. In order to be able to control the computing resources assigned to these threads they are each executing within a virtual processor. A consequence of this is that the number of virtual processors typically equals the number of physical cores. In order to avoid context switching the virtual processors are mapped to different physical cores. In order to enable dynamic frequency/voltage scaling (DVFS), which on certain architectures cannot be applied to the individual cores but only to all the cores, the distribution policy further tries to perform the mapping so that the load on all the cores is balanced as much as possible.

The policy works as follows. First the physical cores are sorted according to their amount of free bandwidth space in descending order and the virtual processors are sorted according to their bandwidth. If the number of virtual processors of the application being registered is equal to or less than the number of physical cores the mapping is simply performed according to this order. Should the number of virtual processors be larger than the number of physical cores then a resorting of the physical cores is performed each a time a number of virtual processors equal to a multiple of the number of physical cores has been mapped.

Figure 6.1 shows the balanced distribution for an application named A1 which has five tasks, each of them within a VP. The generated load is balanced among the four processors (cores). Since the application contains more VPs than the number of cores, two of the VPs will have

the same affinity.

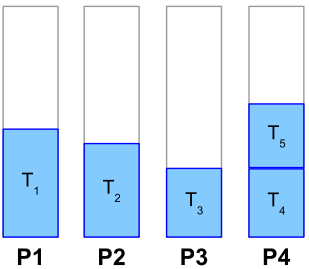


Figure 6.1: Balanced distribution for application A1

The balanced distribution is done only for the new application. In this way the assigned affinity of the currently executing applications is kept constant. Only the size of their VPs is adjusted, that is, increased or decreased accordingly to their assigned service level.

BIP Formulation The balanced distribution can be expressed as a heuristic first fit problem with the objective to evenly maximize the usage of all the cores on the system. This can formulated as a BIP problem [50], where the decision variables are contained in the matrix x of dimension mxn where m is the number of available cores and n is the number of virtual processors of the new application. The value of x_{ij} is 1 if the virtual processor $j \in N = \{1, \ldots, n\}$ of the new application is assigned to core $i \in M = \{1, \ldots, m\}$ and 0 otherwise. The bandwidth requirements of each virtual processor is given by the vector v. The problem can now be stated as follows:

$$\max \sum_{i=1}^{m} \sum_{j=1}^{n} c_i v_j x_{ij} \tag{6.1}$$

$$\sum_{j=1}^{n} v_j x_{ij} \leq c_i$$

$$\forall i, \sum_{j=1}^{n} x_{ij} \leq 1$$

$$\forall j, \sum_{i=1}^{m} x_{ij} = 1$$

In the formulation c_i is the free bandwidth on core i. The second constraint implies that each core can have at the most one VP from the same application, while the third one enforces that a VP can be assigned to only one core. If an application contains more VPs than there are cores, the resource manager will pack some of them together, beginning with the smallest ones. This packing is done so that the problem matches the formulation proposed by Equation 6.1. Once the formulation produces a solution, the packed VPs are unpacked and assigned the same affinity.

The formulation described by Equation 6.1 can be implemented as a first fit bin packing algorithm. The algorithm sorts the VPs of the new application being registered from large to small, and the cores from full to empty. Then it performs the distribution according to the following pseudo algorithm:

Algorithm 1 BalancedDistribution

Require: Sort VPs (large to small) $\wedge$ Ps (full to empty).
Ensure: BalancedDistribution.

```
1: j ⇐ −1
2: for i = 0 to nVPs do {nVPs is number of Virtual Processors}
3:     j ⇐ (j + 1) mod nPs        {nPs is number of Processors}
4:     if j = 0 then
5:         resort Ps from full to empty
6:     end if
7:     if VP[i] fits in P[j] then
8:         map VP[i] to P[j]
9:         reduce space left in P[j]
10:    else
11:        BalancedDistribution failed
12:    end if
13: end for
```

The balanced distribution respects the assigned affinity of the currently executing applications not only during registration of new applications but also when an application unregisters. Similar to the

registration case, the size of the VPs of the running applications is adjusted according to their new assigned service level.

Packed Distribution

Another way to perform the bandwidth distribution is to select the affinity of the virtual processors of an application such that they fit in as few cores as possible. This is also known as the packed distribution. Figure 6.2 shows the packed distribution version of the example presented in the Balanced Distribution subsection. One can notice that this time the number of cores used for the distribution is less than in the balanced distribution case.

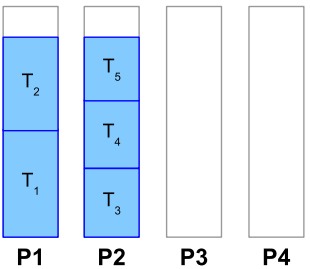


Figure 6.2: Packed distribution for application A1

The motivation for the packed distribution is to utilize as few physical cores as possible, making it possible to switch off or power down the unused cores using power management techniques.

The name packed distribution comes from the fact that the algorithm tries to pack as many virtual processors as possible in the same core. First it sorts the VPs of the application being registered from large to small, and the cores from full to empty. Then it performs the distribution according to pseudo algorithm 2.

The algorithm will always try to fit the virtual processors into cores in the same core order. The packed distribution is done only for the new application respecting the assigned affinity of the already registered applications.

For the packed distribution policy the unregistration of an application may trigger new affinity assignments for the VPs of the running

Algorithm 2 PACKEDDISTRIBUTION

Require: Sort VPs (large to small) $\wedge$ Ps (full to empty).
Ensure: PackedDistribution if $found = 1$.

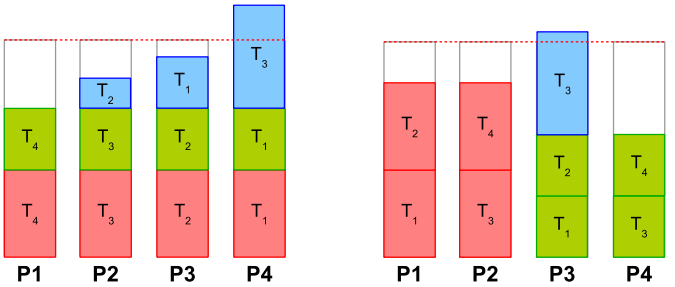
```
1: for i = 0 to nVPs do {nVPs is number of Virtual Processors}
2:     found ⇐ 0
3:     for j = 0 to nPs do {nPs is number of Processors}
4:         if VP[i] fits in P[j] then
5:             map VP[i] to P[j]
6:             reduce space left in P[j]
7:             found ⇐ 1
8:             break
9:         end if
10:    end for
11:    if found = 0 then
12:        PackedDistribution failed
13:    end if
14: end for
```

applications. This ensures that the VPs of the applications are packed in as few cores as possible also after the unregistration.

6.2 Handling Infeasible Distributions

The solution produced by the balanced or the packed distribution may or may not be feasible in terms of schedulability. This means, that the particular partitioning for the assigned service level (how the bandwidth is distributed among the VPs) of each application might not match the free space available on the system. In the case of a non-feasible solution the registration process fails. Figure 6.3 shows the infeasible distribution solutions of the balanced and the packed distribution policies. The scenario has three applications represented by different colors. The infeasibility occurs when the third application (blue) tries to register.

As mentioned in the last section of Chapter 5 this could be the result of not including the BWD values of the applications in the service level assignment formulation described in Equation 5.2. To avoid this situation the resource manager additionally implements two mecha-



(a) (b)

Figure 6.3: Infeasible distribution for the (a) balanced and the (b) packed distributions.

nisms that always produce a feasible solution. The mechanisms are repetitive service level assignment, and compression and decompression algorithm. They are self-contained and can be used independently from each other.

Repetitive Service Level Assignment

Infeasibility occurs when the particular partitioning of the application at *the current assigned service level* cannot be mapped to the system cores. The repetitive service level assignment algorithm addresses the problem by performing a new service level assignment. This new service level assignment does not contain the assigned service level combination that resulted in the infeasible solution. This is repeated until a feasible solution which can be mapped onto the cores is found.

There are different ways to avoid producing the undesired service level combination. A simple way consists of adding a constraint that ensures that the new optimal value of the cost function is always less than it was at the previous optimization. Equation 6.2 shows the new constraint, where Z_O is the old optimal cost function value.

$$\sum_{i=1}^{n}\sum_{j=0}^{m} w_i q_{ij} x_{ij} < Z_O \tag{6.2}$$

Notice that the value of the objective function produced by the undesired combination could also be obtained by another combination

which would not necessarily lead to an infeasible solution. This means that the objective may contain several local maxima. Figure 6.4 shows a simple illustration of this phenomenon, where the curve contains two local maxima defined as a and b. The formulation proposed by Equation 6.2 does not observe this possibility. It directly bounds the upper limit of the new value of the objective function. Hence, it discards the other local maxima that could have produce a feasible solution.

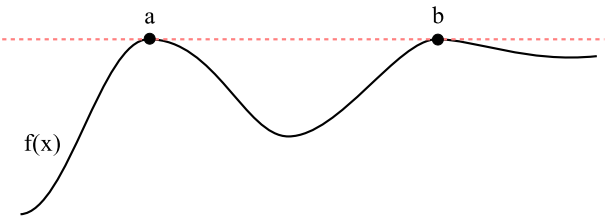


Figure 6.4: A function f(x) with two local maxima elements at points a and b

An advantage with this approach is that the number of constraints remains constant.

A more elegant way to avoid the service level combinations can be achieved by dynamically adding constraints to the formulation described by Equation 5.2. These constraints will include information about the service level assigned to each application that leads to infeasibility. For instance, consider three applications A1, A2, and A3 with three service levels for each of them. Assume that the service level assignment that leads to infeasibility corresponds to 0, 0, and 1 for A1, A2, and A3 respectively. For this case the new constraint added to the formulation would correspond to:

$$x_{10} + x_{20} + x_{31} < 3$$

No matter which of these methods that is used, the repetitive service level assignment will eventually produce a feasible solution. The only difference between them lay on the optimality of the results.

The repetitive service level algorithm has different effects in each of the distribution policies. For the balanced distribution algorithm it respects the assigned affinity of the already registered applications,

and only affects the affinity of the new application. For the packed distribution the algorithm sets the affinity of the applications beginning with the highest importance application. This may lead to a totally new distribution.

Compression and Decompression Algorithm

Another way to handle the infeasible solution produced by Equation 6.1 is through the compression and decompression algorithm. The objective of this algorithm is to always provide a schedulable solution, where the particular partitioning of the new application matches the available free space of the system. Depending on the information collected from the new application that is, the QoS provided at the assigned service level, and the importance with respect to the other applications, the algorithm might trigger a new service level assignment for the new application or even for the currently executing ones. The algorithm can be described as follows:

- Each virtual processor of each application has a nominal bandwidth B_{jn}, which corresponds to the bandwidth distribution value assigned to the virtual processor j at the current service level. The index n means that this is a nominal value.
- Each virtual processor j has a maximum and minimum bandwidth B_{jmax} and B_{jmin}, which correspond to the bandwidth values assigned to the same virtual processor j at the next and previous service level respectively, that is

$$B_{jmin} \leq B_{jn} \leq B_{jmax}$$

- A new bandwidth $\hat{B}_j > B_{jn}$ can be assigned to a virtual processor j as long as the following condition is fulfill

$$\forall_i, \sum_j B_{ij} \leq 1 \qquad i \in P \tag{6.3}$$

  where P corresponds to the set of online processors on the system.
- If Equation 6.3 does not hold then the bandwidth assigned to the virtual processors of the other applications executing in the same

processor must be reduced or compressed according to

$$\hat{B}_j = B_{jn} - (B_n - B_d)\frac{s_j}{S} \tag{6.4}$$

$$B_n = \sum_{\tau_j \in \Gamma_c} B_{jn} \qquad \forall B_n > B_d$$

$$S = \sum_{\tau_j \in \Gamma_c} s_j \qquad \forall_{\tau_j \in \Gamma_c} s_j = g(I_j)$$

$$B_d = B_M - B_f \qquad \forall \hat{B}_j < B_{jmin} \Rightarrow \hat{B}_j = B_{jmin}$$

where Γ_c is the set of VPs which bandwidth can be reduced or compressed, Γ_f is the set of VPs which bandwidth cannot be reduced, B_M is the maximum assignable bandwidth on the system, $g(I_j)$ is a function of the importance value of the application, and s_j is a scaling factor which is inversely proportional to the importance of the application.

In addition to this, the following policies are followed before compressing the bandwidth assigned to the currently executing applications as well as the new application:

- The applications which bandwidth will be compressed are the ones for which the importance times the QoS at the currently assigned service level is smaller than the one of the application that has requested more bandwidth than what is available on the system. If the compressed bandwidth of the applications is greater than B_{jmin} (the assignable bandwidth at the next lower service level), then the application keeps its assigned service level, otherwise the service level is decreased.
- In case the importance times the QoS of the new application at the currently assigned service level is smaller than the ones of all the other applications, then the new application receives the remaining free available bandwidth on the system. If this is greater than B_{jmin} (the assignable bandwidth at the next lower service level), then the application keeps its assigned service level, otherwise the service level is decreased.

As can be seen from the previous policies, the compression of the

bandwidth either in the currently executing applications or in the new application can trigger a change in the assigned service level.

Bandwidth Decompression Each time an application unregisters, the available free bandwidth is distributed among the other applications which bandwidth was compressed, this is known as the bandwidth decompression. By decompressing the bandwidth of the applications which where affected by the compression algorithm previously described, the performance of these applications can be increased.

The algorithm can be described as follows:

- Two sets of applications can be considered, the set of applications that have been compressed, that is Γ_c and which current bandwidth is smaller than the nominal one, that is $B_j < B_{jn}$, and the set of applications that have not been compressed, that is Γ_f and which current bandwidth is greater or equal than the nominal one, that is $B_j \geq B_{jn}$
- Considering that an application that belongs to the set Γ_f decreases its bandwidth consumption then two cases can be observed.

  1. Decompress under the assumption that:

$$B_{cn} + B_f \leq B_M \tag{6.5}$$
$$B_{cn} = \sum_{\tau_j \in \Gamma_c} B_{jn}$$
$$B_f = \sum_{\tau_j \in \Gamma_f} B_{jn}$$

     where B_{cn} is the total sum of the original nominal bandwidth of the applications that have been subject to bandwidth compression, and B_f is the total sum of the assigned bandwidth of the applications that have not been compressed. In case the sum of these bandwidths is smaller or equal than the total assignable bandwidth on the system, that is B_M, then the bandwidth of all the compressed applications can be restored to its nominal value.

2. Compress again under the assumption that:

$$B_{cn} + B_f > B_M \tag{6.6}$$

In case the sum of these bandwidths is greater than B_M, then the set is not schedulable and therefore their bandwidth must be compressed again using the compression algorithm. Notice that this time the constraint over the bandwidth that can be distributed among the compressed applications, that is B_d, will be less restrictive than the first time the compression was carried out.

6.3 Reservation Parameters Assignment

After finding an schedulable solution to the bandwidth distribution problem, either with the balanced or the packed distribution method, the resource manager must set the values of the reservation parameters for each of the virtual processors of the new application. The reservation parameters of a virtual processor are defined by the assigned budget Q and the assigned period P. The reservation parameters can be directly calculated from the BW and granularity values provided in the service level table of each application as follows:

$$\begin{aligned} P_j &= \text{Granularity}_i \\ Q_j &= \frac{\text{BWD}_j \; P_j}{100} \end{aligned} \tag{6.7}$$

where j is the index for the number of the virtual processor of the new application.

6.4 Example

A simple example containing several applications with different structures will be described. This will allow us to compare the performance of the different methods previously described to solve the bandwidth distribution problem.

The scenario contains three applications named A1, A2, and A3 with 3, 4 and 2 service levels respectively. Table 6.1 shows the service level information provided by the three applications to the resource manager. For completeness, the importance value I is also included in Table 6.1.

Table 6.1: Service level table of application A1, A2 and A3

Application name	I	SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
A1	10	0	100	160	40	[40, 40, 40, 40]
		1	80	120	50	[30, 30, 30, 30]
		2	50	80	100	[20, 20, 20, 20]
		x	1	4	100	[1, 1, 1, 1]
A2	1	0	100	200	20	[50, 50, 50, 50]
		1	90	160	40	[40, 40, 40, 40]
		2	70	120	70	[30, 30, 30, 30]
		3	40	80	150	[20, 20, 20, 20]
		x	1	4	100	[1, 1, 1, 1]
A3	100	0	100	80	20	[20, 15, 45]
		1	70	60	100	[20, 10, 30]
		x	1	3	100	[1, 1, 1]

In addition to this the resource manager also knows the number of VPs that each application contains, that is 4 for A1 and A2 and 3 for A3, and the importance of each of the applications, in this case 10, 1 and 100 for A1, A2 and A3 respectively. The number of VPs can also be directly obtained from the number of partitions in the BWD value.

Implementation Considerations

For this example it was assumed that only 90% of the CPU of each of the four cores could be allocated at any time, this implies a total available bandwidth of 360% for the system.

The methods implemented by the balanced and the packed distribution were directly coded in `C++`. This also includes the BIP formulation

described by Equation 6.1. In this case the GLPK toolkit was not used.

The repetitive service level assignment used for both distributions implements the constraint defined by Equation 6.2. Hence the new cost value produced by Equation 5.2 is upper bounded by the old cost value which led to a non schedulable solution.

Balanced Distribution

At the beginning A1 and A2 register with the resource manager at time t_0 and t_1 respectively. The resource manager assigns service level 0 to both applications according to Equation 5.2. The balanced distribution methodology distributes the load of the virtual processors evenly among the system processors. This is done at time t_0 and t_1 for A1 and A2 respectively.

Figure 6.5 shows the assigned service level (SL), the total bandwidth (BW), and the bandwidth distribution (BWD) values of both applications, as well as a graphic representation of the bandwidth distribution of the two applications on the four core platform.

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	0	200	[50, 50, 50, 50]

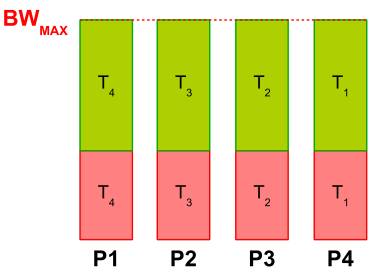


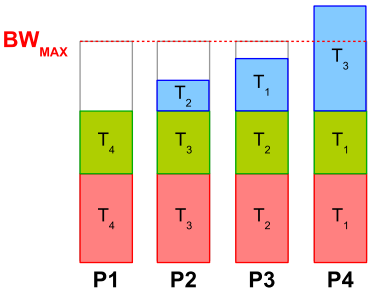
Figure 6.5: Normal registration of applications A1 and A2.

After some time, A3 with higher importance than A1 and A2 registers with the resource manager. Following Equation 5.2 the resource manager assigns service level 0, 2, and 0 to A1, A2, and A3 respectively. According to the balanced distribution formulation in Equation 6.1, the solution to the bandwidth distribution problem is not schedulable. This can be seen in Figure 6.6. In order to handle the infeasible solution, the repetitive service level assignment method, as well as the compression and decompression algorithm are used.

Repetitive Service Level Assignment The non schedulable solution triggers the repetitive service level assignment method. The

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	2	120	[30, 30, 30, 30]
A3	0	80	[20, 15, 45]

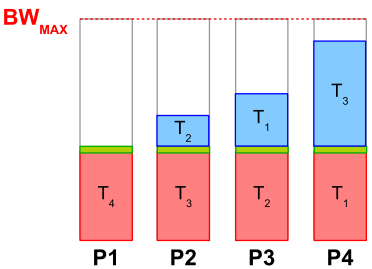
Figure 6.6: Registration of application A3 that leads to an infeasible solution.



method carries out two more service level assignments until it finds a feasible solution that can be mapped onto the cores. Figure 6.7 shows the new service level assignment and the bandwidth distribution for the three applications.

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	x	4	[1, 1, 1, 1]
A3	0	80	[20, 15, 45]

Figure 6.7: Registration of application A3 after repetitive service level assignments.



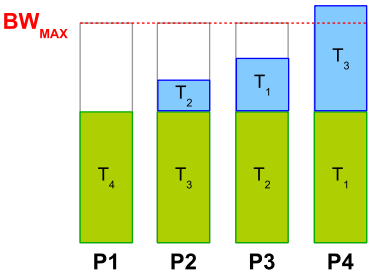
After a while A1 unregisters and the resource manager assigns new service levels to the remaining executing applications A2 and A3. This assignment produces again a non schedulable solution as can be seen in Figure 6.8.

In order to avoid infeasibility, a new service level assignment is done. In this way, A2 is assigned service level 1, while A3 remains at its old service level. Figure 6.9 shows the feasible distribution after the new service level assignment.

Compression and Decompression Algorithm The infeasible solution shown in Figure 6.6 can also be handled by the bandwidth com-

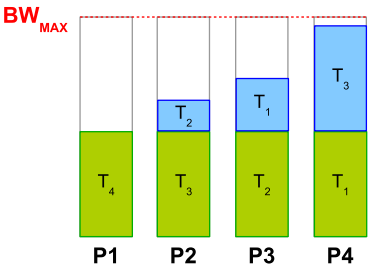
App. name	SL	BW [%]	BWD [%] [%]
A2	0	200	[50, 50, 50, 50]
A3	0	80	[20, 15, 45]

Figure 6.8: New service level assignment of A2 and A3 after unregistration of A1.



App. name	SL	BW [%]	BWD [%]
A2	1	160	[40, 40, 40, 40]
A3	0	80	[20, 15, 45]

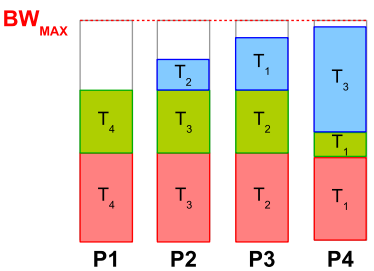
Figure 6.9: New service level assignment of A2 and A3 after repetitive service level assignment.



pression algorithm. According to Equations (6.3) and (6.4), the algorithm reduces the service level of the lowest importance application A2 from 2 to 3 and also reduces the bandwidth values of the virtual processors of A1 and A2 executing in core P4. The final schedulable solution is shown in Figure 6.10.

App. name	SL	BW [%]	BWD [%]
A1	0	157	[40, 40, 40, 37]
A2	3	67	[20, 20, 20, 7]
A3	0	80	[20, 15, 45]

Figure 6.10: Registration of application A3 after bandwidth compression.



Similar to the repetitive service level assignment case, A1 unregisters after finishing its execution. This leads again to the problem shown in Figure 6.8. The bandwidth compression algorithm produces a schedulable solution where the service level of A2 is reduced from 0 to 1, as shown in Figure 6.11.

App. name	SL	BW [%]	BWD [%]
A2	1	160	[40, 40, 40, 40]
A3	0	80	[20, 15, 45]

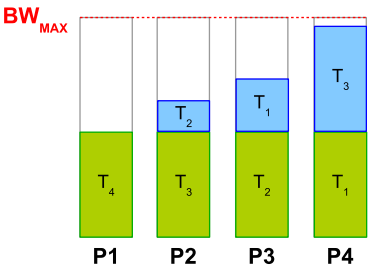


Figure 6.11: New service level assignment of A2 and A3 after bandwidth compression.

As can be seen in the example, each time that an application registers or unregisters with the resource manager, a new service level assignment as well as bandwidth distribution is carried out according to Equations 5.2 and 6.1. This solution might not be schedulable considering all the possible combinations of all the different partitions of each of the applications running on the system. In order to produce a schedulable solution, the bandwidth compression algorithm compresses the bandwidth of the applications in the system according to Equations 6.3 and 6.4, which could again trigger a new service level assignment.

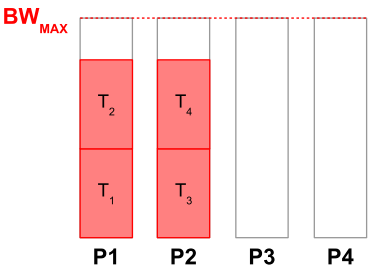
Packed Distribution

For the packed distribution case, only the repetitive service level assignment is considered. In order to see the differences between this distribution and the balanced one the registration of each application will be described. At time t_0 application A1 registers with the resource manager, which assigns service level 0. The packed distribution sets the affinity of the virtual processors such that they fit in as few cores as possible. This can be seen in Figure 6.12 which shows the bandwidth distribution for A1.

At time t_1 application A2 begins the registration with the resource manager. According to Equation 5.2 the resource manager assigns ser-

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]

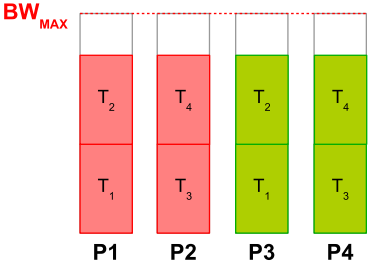
Figure 6.12: Registration of application A1.



vice level 0 to both applications. Of course this leads to an infeasible solution, which the packed distribution handles by recalling the service level assignment method. After one new service level assignment, which reduces the service level of A2 to 1, the packed distribution is able to map the virtual processors into the system cores. Figure 6.13 shows the final result of the distribution.

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	1	160	[40, 40, 40, 40]

Figure 6.13: Registration of application A2 after packed distribution.

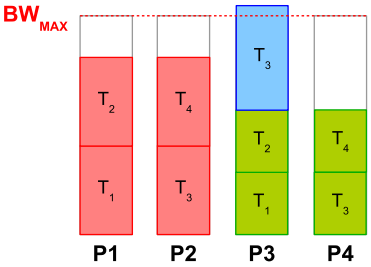


Application A3 registers with the resource manager at time t_3. The resource manager assigns service level 0, 2 and 0 to A1, A2 and A3 respectively. This leads to an infeasible solution when the packed distribution tries to assign the largest VP in the emptiest core. Figure 6.14 shows the infeasible distribution.

The infeasibility is handled by the repetitive service level assignment which assigns service level 3 to A2. Additionally, new affinity values are assigned to all applications. Figure 6.15 shows the feasible solution. Notice that the VPs of A3 (highest importance), are the first to be assigned to a core. Then the assignment follows with A1 and A2.

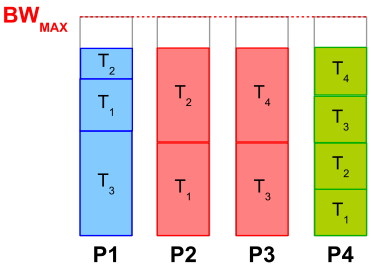
App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	2	120	[30, 30, 30, 30]
A3	0	80	[20, 15, 45]

Figure 6.14: Registration of application A3 which leads to an infeasible solution.



App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	3	80	[20, 20, 20, 20]
A3	0	80	[20, 15, 45]

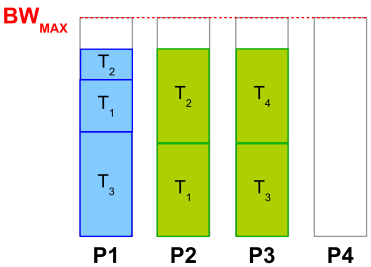
Figure 6.15: Registration of application A3 after repetitive service level assignment.



When A1 unregisters the resource manager assigns service level 0 to A2 and A3, which leads to an infeasible distribution. To solve this the repetitive service level assignment produces a new service level for A2. The final result of the feasible distribution is shown in Figure 6.16.

App. name	SL	BW [%]	BWD [%]
A2	1	160	[40, 40, 40, 40]
A3	0	80	[20, 15, 45]

Figure 6.16: New service level assignment of A2 and A3 after repetitive service level assignment.



7

Bandwidth Adaption

The distribution of CPU resources is performed by the resource manager in two different ways. In the first case the resource manager adapts the applications to changes in the resource availability. This is done by changing the service level of the applications. This adaption takes place whenever applications register or unregister with the resource manager or when the amount of available resources changes. It is event based and includes not only the assignment of the service level, but also the distribution of the bandwidth at the assigned service level.

In the second case the resource manager adapts the resource distribution to changing application requirements. This takes place online during the execution of applications. At this moment the resource manager has provided CPU resources to the application according to the information provided in its service level table. However, this information serves just as an initial prediction of the real amount of resources needed by the application at a certain service level.

It is the task of the resource manager to find out this amount of resources such that the resources are optimally used and not wasted. To do so the resource manager uses the dynamic information provided by the application, that is, the happiness value, and the information obtained from the scheduler such as the application resource utilization values. Based on this information the algorithms implemented by the resource manager will adapt the bandwidth provided to each application.

7.1 Resource Utilization Feedback

To guarantee optimal use of the resources provided to the application, the resource manager periodically measures the application resource utilization. Based on this, as well as the control strategy implemented the resource manager adapts the distributed bandwidth of each virtual processor in each of the cores.

Controller Inputs

The resource utilization values include the cumulative used budget and exhaustion percentage which are periodically fed back from the scheduler layer to the resource manager. Figure 7.1 shows the resource

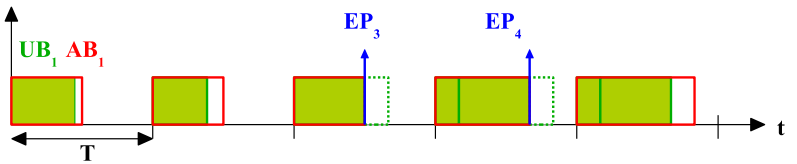


Figure 7.1: Resource utilization measurements per server period for application A1.

utilization measurements inside one of the virtual processors of an application. For explanatory reasons the figure considers the resource utilization measurements per server period and not the cumulative ones. In the figure UB, EP, T, and AB stand for used budget, exhaustion percentage, period, and assigned budget respectively. The assigned budget and period correspond to the reservation parameters assigned by the resource manager to each virtual processor of the application. The used budget as well as the exhaustion percentage values reflect the resources consumed by the tasks running inside the virtual processor.

In the figure one can see that during the first two periods the assigned budget is almost completely consumed. In the third period, the task wants to consume more resources than the ones provided, this is represented by the dashed block. Due to the hard CBS nature of the reservation, the task is not able to consume more than the current AB_3. This triggers an event which is represented by the exhaustion percentage value EP_3. This event as well as the used budget in the

current period will indicate to the resource manager to increase the assigned budget for the next period.

The process of adapting the assigned budget is carried out during the lifetime of the application. It begins after the application has successfully registered with the resource manager and ends when the application unregisters.

For illustration reasons in Figure 7.1 the adaption of the bandwidth is done at each period. In reality this is done at time intervals, which correspond to the sampling time of the controller. This sampling time will be a multiple of the period assigned to the virtual processors of the application. The logic explanation behind this is that the resource manager should execute the feedback algorithms only in response to major changes in the resource utilization. This is something that only can be noticed after the task running inside the virtual processor has executed for some time.

The cumulative values of the used budget and exhaustion percentage are further processed by the resource manager. This results in average values of the used budget and exhaustion percentage within each sampling interval. These values together with the assigned budget of the last sampling interval provide the inputs to the controller.

Controller Strategy

The average used budget and the exhaustion percentage represent the process variables of the control strategy. The correlation between these two variables depends on many factors. One factor is the relationship between the used budget and the assigned budget for a particular instance of time, for instance when the used budget equals or exceeds the assigned budget. Another is the synchronization between the activation time of the task within the reservation and the replenishing time of the reservation budget.

Figure 7.2 shows the dependencies between the used budget and exhaustion percentage variables within a single period. In the case of Figure 7.2a the activation and replenishing times are synchronized. Since the task requires more budget than the assigned one, the exhaustion percentage event is triggered. In the case of Figure 7.2b the exhaustion percentage event is also triggered. However, this happens not due to lack of budget but due to the lack of synchronization between the activation and replenishing time.

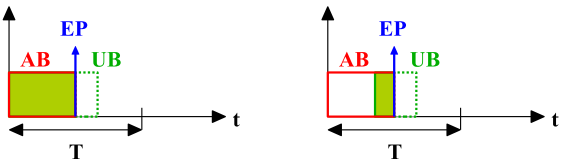


Figure 7.2: Used budget and exhaustion percentage dependencies.

Considering each of the factors that could affect the correlation between the process variables would result in the implementation of a complex controller algorithm. This controller would also require high consumption of CPU resources which are mainly designated for the applications. The trade off between the complexity of the algorithm and the resources needed by the controller is represented by the bandwidth controller shown in Figure 7.3.

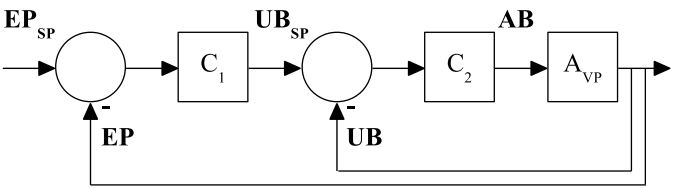


Figure 7.3: Bandwidth controller structure.

The figure shows the cascade structure of the bandwidth controller. The resource manager assigns one bandwidth controller to each virtual processor of the applications. In the figure the average used budget UB and exhaustion percentage EP correspond to the process variables of the inner and outer loop respectively. The task of the outer controller C_1 is to define the set point UB_{SP} of the inner controller C_2 based on the values of EP_{SP} and EP. The inner controller C_2 defines the assigned budget AB for each of the virtual processors of every application. The AB is defined such that the UB does not deviate from the UB_{SP} defined by the outer controller. Each of the set point values EP_{SP} and UB_{SP}

do not correspond to scalar values but to bounded intervals.

The idea behind the bandwidth controller is to be able to keep the average used budget and exhaustion percentage within the bounds defined by the used budget and exhaustion percentage set points respectively. This can be achieved by adjusting the assigned budget of the virtual processors.

Outer Controller The inputs of the outer controller C_1 are EP and EP_{SP}. The average exhaustion percentage EP represents the percentage of server periods when the used budget exceeds the assigned budget. It may also represent the percentage of server periods within a sampling interval where the activation and replenishing time where not synchronized.

The EP value, can have a very noisy nature. This is the effect of the different factors that affect the dependencies with the UB. Consider that UB reflects the amount of resources used by a task inside a reservation, and that those resources may change abruptly over time. For instance in the case of a MPEG 4 video decoder application, decoding a full color image may need more resources than the ones needed for a black and white image.

This poses some constraints to the selection of the EP_{SP}. Thus, it is defined by the interval $[EP_{SL}, EP_{SU}]$, which defines the lower and upper limit of the exhaustion percentage set point. Figure 7.4 shows

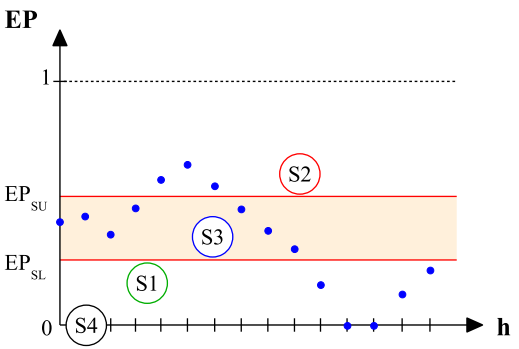


Figure 7.4: Exhaustion percentage set point defined by the limits EP_{SL} and EP_{SU}

the average EP value. The EP_{SP} defines three areas in the figure. Different decisions will be taken by the outer controller depending on which of these areas the EP is in. These decisions are represented by the states $S1$ to $S4$ in the outer controller.

Each state defines actions taken by the outer controller. The controller output produced in each of these states will affect the UB_{SP} of the inner controller. Just like in the case of the EP_{SP}, the UB_{SP} is also specified by the interval $[UB_{SL}, UB_{SU}]$.

Figure 7.5 shows the different states of the outer controller. The state $S0$ represents the initial state. State $S1$ affects the used budget bounds such that AB is decreased. Similarly, state $S2$ affects the bounds such that AB is increased. State $S3$ affects both bounds causing AB to be kept constant, this defines a stability region for the controller. State $S4$ smooths the action of the state $S1$. The variable D corresponds to the sample standard deviation of UB that will later on be explained in more detail.

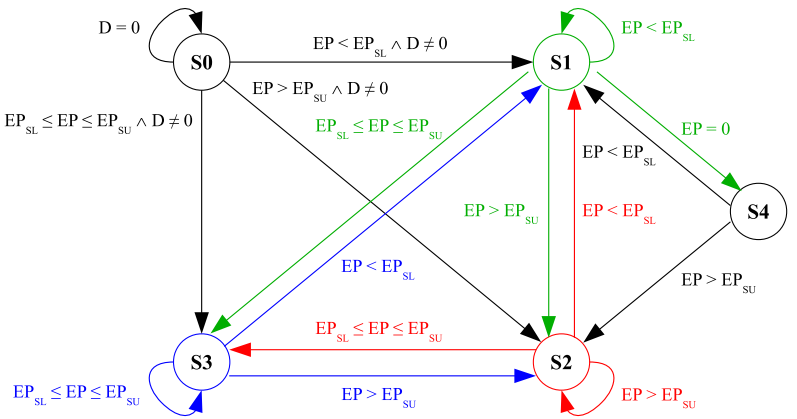


Figure 7.5: Outer controller state machine

When an application has registered, the resource manager initializes bandwidth controllers for each of the virtual processors of the application. Thus, the outer controller state is set to $S0$, the exhaustion percentage set point limits EP_{SL} and EP_{SU} are set to values that guarantee a good performance of the registered application, while the

used budget set point limits UB_{SL} and UB_{SU} are set to initial default values that later on will be modified or changed by the outer controller.

In order to be able to change the UB_{SP} values, the outer controller needs to know the initial values of UB_{SP}, as well as the trend of the average used budget UB in the last sampling intervals. The trend of UB during the last sampling intervals is obtained through statistical measurements. These measurements include the sample mean and the sample standard deviation of UB.

The size of the time window where the statistical measurements are calculated must be defined considering different aspects. It must be able to catch the abrupt changes that UB may experience from one sampling time to another. At the same time it must filter the UB signal that by nature can be very noisy.

The statistical measurements are defined by Equation 7.1, where $\overline{UB}$ and D correspond to the used budget sample mean, and the used budget sample standard deviation respectively. The total number of observations N is a multiple of the sampling interval. Its selection is a trade off between having not enough information and using too old information to obtain the trend of UB.

$$\overline{UB} = \frac{1}{N}\sum_{i=1}^{N} UB_i \tag{7.1}$$

$$D = \sqrt{\frac{\sum_{i=1}^{N}(UB_i - \overline{UB})}{N-1}}$$

Recalling the state machine of the outer controller shown in Figure 7.5, one can see that in order to evolve from the initial state $S0$ to $S1$, $S2$, or $S3$, the outer controller uses the exhaustion percentage value EP, and additionally the sample standard deviation D. In $S0$ no changes are done to UB_{SP}, mainly it provides the time needed to generate the statistical measurements required by the other states.

Figure 7.6 shows the different transitions among the states of the outer controller. It also shows how the output of the outer controller changes the lower and upper limit of the average used budget that is, UB_{SL} and UB_{SU}.

The transition to the state $S1$ from any of the other states is done

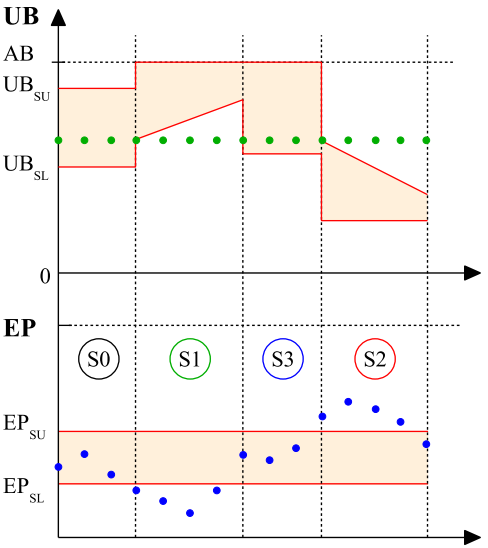


Figure 7.6: States transitions of the outer controller and changes of the UB_{SP} in each state.

whenever EP is below EP_{SL}. The limits defined by UB_{SL} and UB_{SU} are changed according to Equation 7.2.

$$
\begin{aligned}
e_{EP} &= EP_{SL} - EP, \qquad e_{EP} \in [e_m, e_M] \\
O_L &= -\frac{(b-a)}{e_M} \, D e_{EP} + bD \\
UB_{SP} &= \begin{cases} UB_{SU} & = AB \\ UB_{SL} & = UB + O_L \end{cases}
\end{aligned}
\tag{7.2}
$$

Here e_{EP} corresponds to the exhaustion percentage error. This value is bounded by $[e_m, e_M]$ which are the minimum and maximum e_{EP}. For state $S1$, e_m and e_M correspond to 0 and EP_{SL} respectively.

An exhaustion percentage EP smaller than EP_{SL} implies an overestimation of the assigned budget AB. This problem can be solved by shifting UB_{SL} by a factor which is a function of the sample standard

deviation D and the exhaustion percentage error e_{EP}. This is indicated by O_L. The lower offset or O_L correspond to a line equation with negative slope, where a and b are small positive constants. These constants determine the aggressivity of the controller.

The transition to state $S2$ is done whenever EP is greater than EP_{SU}. The limits defined by UB_{SL} and UB_{SU} are changed according to Equation 7.3.

$$\begin{aligned} e_{EP} &= EP_{SU} - EP, \qquad e_{EP} \in [e_m, e_M] \\ O_U &= -\frac{(b-a)}{e_M} De_{EP} + aD \\ UB_{SP} &= \begin{cases} UB_{SU} & = UB - O_U \\ UB_{SL} & = UB_{SU} - cD \end{cases} \end{aligned} \tag{7.3}$$

Similar to the previous case EP is bounded by $[e_m, e_M]$, which corresponds to the interval $[0, 1 - EP_{SU}]$.

An exhaustion percentage EP greater than EP_{SU} implies an underestimation of the the assigned budget AB. This is handled by adjusting UB_{SU}. In this case the bound is shifted such that it lays below UB. The shifting factor also know as the upper offset O_U corresponds to a line equation with a positive slope. The constants a, b, and c have positive values.

The transition to state $S3$ is done whenever EP is within the interval $[EP_{SL}, EP_{SU}]$. In this case the upper and lower limits of UB_{SP} are defined by Equation 7.4 where c is a positive constant.

$$UB_{SP} = \begin{cases} UB_{SU} & = AB \\ UB_{SL} & = UB - cD \end{cases} \tag{7.4}$$

The outputs produced by state $S3$ set UB_{SU} and UB_{SL} such that UB lays within the bounds. Thus, a stability region is reached.

The last state $S4$ can only be reached from the state $S1$ whenever EP is equal to 0. The output is the same as the one produced by the state $S3$ (see Equation 7.4). This state smooths the output produced in the state $S1$. Whenever EP is smaller than EP_{SL} the state machine will be oscillating between the states $S1$ and $S4$.

The constants a, b, and c previously described have different values for each state.

Inner Controller The function of the inner controller C_2 is to change the assigned budget AB provided to the virtual processor. This is done based on the deviation between UB and UB_{SP}. When the bandwidth controller is executed the first time, the UB_{SP} has initial default values for UB_{SU} and UB_{SL}. These values are updated if necessary by the outer controller C_1.

The inner controller C_2 is also modeled as a state machine. The state machine of the inner controller consisting of four states is shown in Figure 7.7. In the figure $S0$ corresponds to the initial state. The states $S1$ and $S2$ are the ones that will change AB according to deviation between the UB and the UB_{SP}. The state $S3$ keeps the value of the AB generated in any of the other states.

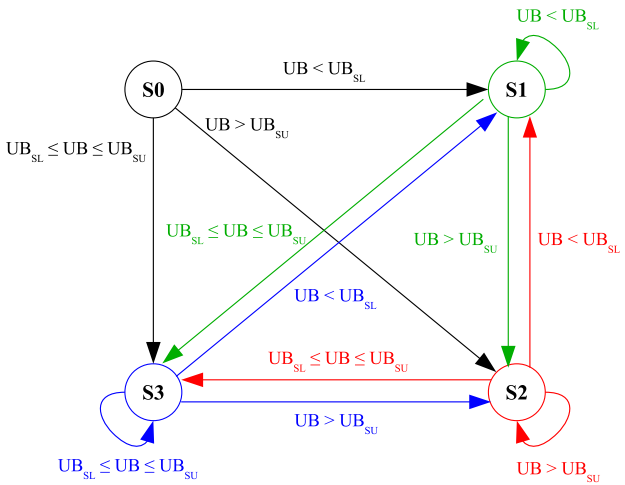


Figure 7.7: Inner controller state machine

The initial state $S0$ is able to reach the other states once the bandwidth controller begins to execute. In this state no changes are done to AB, that is, it keeps the value assigned during registration.

The changes produced in AB by the states $S1$ to $S3$ are shown in Figure 7.8. In the figure AB_M is the maximum allowed assigned budget. It corresponds to the budget assigned during registration, and is also known as the initial budget. In Chapter 8 it will be shown that

this value can be tuned for each service level through the bandwidth controller.

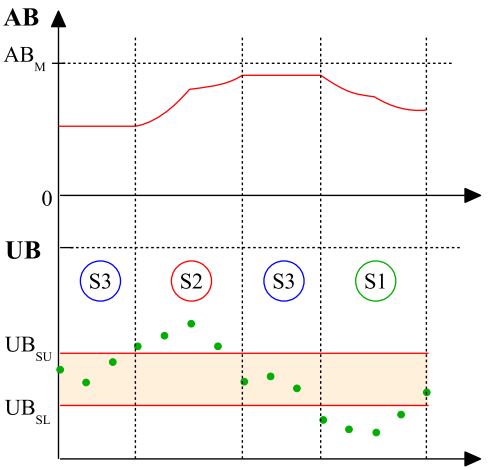


Figure 7.8: States transitions of the inner controller and changes in the AB in each state.

The state $S1$ can be reached from any of the states whenever UB is smaller than UB_{SL}. This suggests a non-optimal use of AB or waste of resources. Thus, the assigned budget must be reduced according to Equation 7.5. In the equation e_L is the controller error with respect to the lower bound UB_{SL}. Similar to the case of the outer controller this error is also bounded by e_m and e_M, which correspond to 0 and UB_{SL} respectively. An exponential controller is used to change the value of AB. How fast or slow it changes will depend on the value of K_L and a. The factor K_L changes dynamically according to e_L and is bounded by the interval $[1, 10]$. The constant a is a positive small number derived through tuning.

$$\begin{aligned} e_L &= UB_{SL} - UB, \qquad e_L \in [e_m, e_M] \\ K_L &= \frac{9}{UB_{SL}}\, e_L + 1 \\ AB &= e^{-aK_L}\, AB \end{aligned} \tag{7.5}$$

The state $S2$ can be reached if UB is greater than UB_{SU}. In this case the assigned budget is too low to satisfy the performance criteria of the outer and the inner controller. Hence the assigned budget is increased according to Equation 7.6. The controller error e_U is upper and lower bounded by e_m and e_M, which for the state $S2$ correspond to 0 and $AB - UB_{SU}$ respectively. The controller employed is also an exponential controller. The factor K_U changes dynamically in the interval $[1, 10]$, the small positive constant a is derived through tuning.

$$\begin{aligned} e_U &= UB - UB_{SU}, \qquad e_U \in [e_m, e_M] \\ K_U &= \frac{9}{AB - UB_{SU}}\ e_U + 1 \\ AB &= e^{\,aK_U}\ AB \end{aligned} \tag{7.6}$$

The state $S3$ is reached whenever UB lays within the bounds defined by UB_{SL} and UB_{SU}. This means that the performance criteria of both controllers C_1 and C_2 is satisfied. Under this conditions the assigned budget keeps its current value.

7.2 Achieved QoS Feedback

The resource manager also adapts the resources distributed to the registered applications based on their achieved QoS, which is indicated by the happiness value. This approach considers the achieved QoS as a function of the assigned service level and the CPU resources provided at this service level only. It does not consider other factors that may affect the obtained QoS. For instance in the case of a video conference application, it would not consider degradation of the application performance due to package losses or time delays on the communication which are not CPU bandwidth dependent.

Controller Input

The happiness value is an indicator of the quality obtained with the allocated resources at the assigned service level. It takes one of two values 0 or 1, with 1 meaning that the application is happy and 0 otherwise. For those applications that cannot provide the happiness

value the resource manager assumes that the application is always happy.

Controller Strategy

For the achieved QoS feedback the bandwidth controller corresponds to a simple proportional controller. The controller is activated if the application is unhappy. Figure 7.9 shows how the inner and outer state machines of the logic explained in Section 7.1 evolve to the unhappy state S.

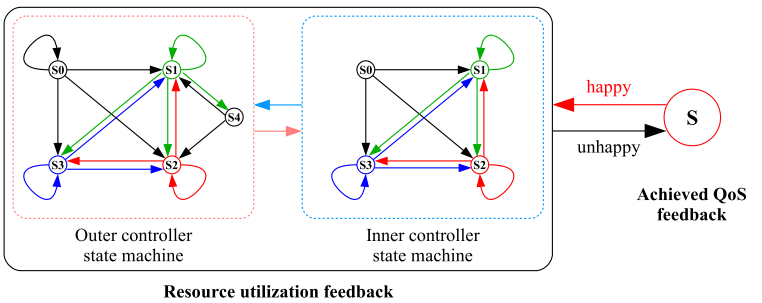


Figure 7.9: Complete state machine of the bandwidth controller.

The state S is activated when the application of the virtual processor is unhappy. In such a case the controller simply increases the assigned budget AB linearly according to Equation 7.7, where K is the proportional constant of the controller.

$$AB = K\ AB \tag{7.7}$$

The assigned budget AB is increased until the application becomes happy again or the assigned budget becomes equal to the initial budget AB_M of the virtual processor.

The happiness value sent by the application is event based in nature. However, the bandwidth controller always consider the most recent happiness value and uses time triggered control.

7.3 Example

Two different scenarios are shown in this section. In the first scenario a CAL MPEG 4 SP decoder application is used. This scenario shows how the bandwidth controller adapts the assigned bandwidth and the effects of different sampling periods and exhaustion percentage set points in the performance of the application.

In the second scenario the MPEG 4 SP decoder is used together with a CAL periodic pipeline application. This allow us to evaluate the bandwidth controller of the CAL MPEG 4 SP decoder at different service levels.

The information related to the decoder and pipeline applications is shown in Table 7.1. The importance values of the decoder and the

Table 7.1: Service level table of the decoder and pipeline applications

Application name	I	SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
Decoder	1	0	100	120	100	[60, 60]
		1	80	100	330	[50, 50]
		2	60	40	400	[20, 20]
Pipeline	10	0	100	80	20	[40, 40]
		1	90	54	40	[27, 27]
		2	70	32	70	[16, 16]

pipeline application are 1 and 10 respectively, which implies that the pipeline application is more important than the decoder application.

In this section the terms used bandwidth and assigned bandwidth will be used instead of used budget and assigned budget respectively.

Implementation Considerations

The decoder is connected to an Axis network camera that streams MPEG 4 SP frames. The decoder has two partitions, three service levels, and can report its happiness value to the resource manager. When the decoder is required to switch to a lower service level it configures the camera to reduce the frames per second (fps) and resolution in

order to reduce the resources required to decode the video frames. The happiness is a boolean value which indicates if the resulting frame rate of the displayed video corresponds to what can be expected at the current service level. Figure 7.10 shows the internal structure of the MPEG 4 SP application.

Figure 7.10: MPEG 4 SP application.

The periodic pipeline application has two partitions and three service levels. This application is intended to model a typical rate-based streaming application. The structure of the application is shown in Figure 7.11. The application has two partitions and consists of four parallel pipelines, where each pipeline consists of four actors: one producer actor, two forward actors, and one consumer actors. The producer is triggered by a clock token from the clock system actor. When triggered it generates a token that enters a feedback loop where the number of loops taken depends on a parameter value. Through this value it is possible to model that the computations performed by the producer

takes a certain amount of time. After the correct number of loops the token is forwarded to the first forward actor. This also feeds back the token for a user-dependent number of loops before it is forwarded to the next forward actor. The final consumer actor instead consumes the token once the feedback loops are finished.

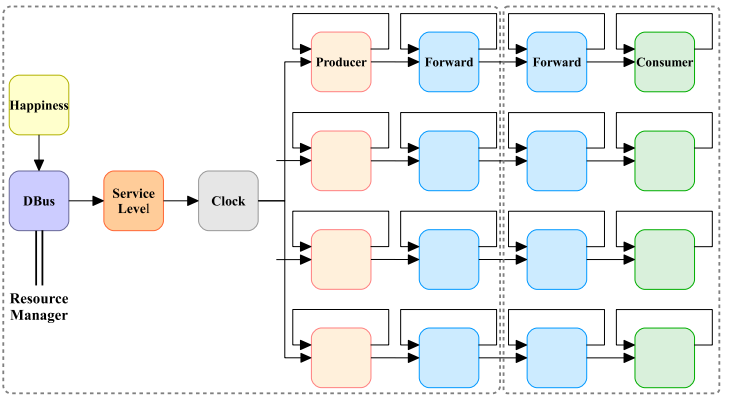


Figure 7.11: Periodic pipeline application. The dashed rectangles represent the different partitions.

The DBus actor constitutes the interface to the resource manager. When the resource manager changes the service level, it is being translated into a corresponding sampling period for the clock actor. Finally, the happiness actor implements a keyboard interface through which the user interactively can change the happiness of the application, in which case the value is forwarded to the resource manager over the D-Bus.

The periodic pipeline application has three service levels where the service levels correspond to different sampling periods. Although the amount of computations performed per sampling period is the same independently of the sampling period, that is, the required budget is the same, the required bandwidth gets smaller as the service level value increase. The delay of the application in the different service levels are equal to the sampling periods.

The values of the different constants of the bandwidth controllers are shown in Table 7.2. In the achieved QoS feedback, the value of the

constant K was set to 1.1. This constant as well as the ones shown in Table 7.2 were used for both applications.

Table 7.2: Tuning constants of the bandwidth controller based on resource utilization feedback.

	Outer controller			Inner controller	
	$S1$	$S2$	$S3$	$S1$	$S2$
a	0.0625	0.5	-	0.01	0.025
b	0.125	0.75	-	-	-
c	-	2	2	-	-

The experiment is carried out in a dual core system with resource availability of each core set to 90%. The sampling period for the two scenarios is $10P_i$, where P_i is the server period of the application i that is being controlled. Notice that the server period varies with the service level for the periodic pipeline application. An exception occurs in two of the experiments in scenario 1, where the sampling interval corresponds to $5P_i$. The size of the time windows to do the statistical measurements, that is N, was set to $5h_i$, where h_i is the sampling period of the application i. In order to evaluate the different input and output signals of the bandwidth controllers, the UB, AB and EP signals are normalized to values between 0 and 1.

Scenario 1: MPEG 4 SP Decoder Application

For the first scenario three different experiments are carried out. In the first experiment EP_{SP} is set to $[0.1, 0.18]$. Figure 7.12 shows the bandwidth adaption for virtual processor 0 ($VP0$) of the decoder application. The UB (green), AB (red) and EP (blue) are shown in the first two plots. The transitions between the different states of the outer controller are shown in the last plot.

At time $t = 0$ the decoder application registers with the resource manager, which assigns service level 0 to the application. This produces an initial bandwidth distribution of 0.6 to both virtual processors. At the beginning the state machine of the outer controller is in $S0$ which is shown as value 0 in the lower plot. At this point the resource manager collects information about the trend of the UB. After sometime it

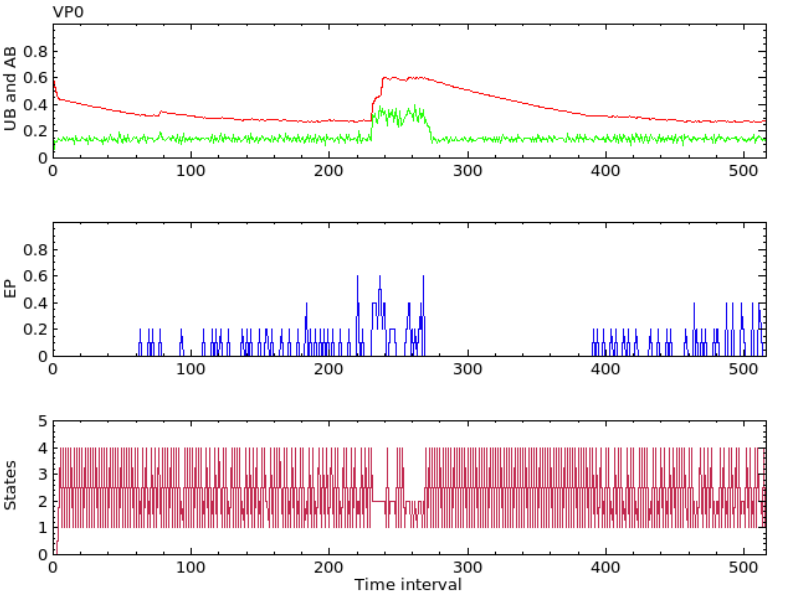


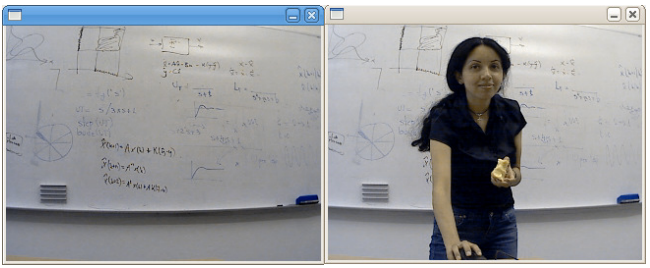
Figure 7.12: Resources adaption for VP0 of the CAL SP decoder application.

begins to generate the statistical measurements required by the outer controller. Thus, the outer controller begins to switch among states $S1$ to $S4$. One can observe that when decreasing AB there is a back and forth transition between states $S1$ and $S4$, which provides a smoother decrease of AB. Around time $t = 220$ a disturbance occurs which increases the resource consumption and gives rise to a deviation of EP from the set point EP_{SP}. This is corrected by increasing AB. This is a combination of the actions of the states $S2$ of the outer and the inner controller.

It is important to remark that each value in the *Time interval* axis in Figure 7.12 corresponds to one sampling interval, which in this case is equal to one second.

The disturbance consists of introducing a moving person in the image. This increases the complexity of the frames that must be decoded. At the same time it increases the amount of resources needed by the

decoder application to produce an image that satisfies the QoS requirements. The images of the video generated can be seen in Figure 7.13.



(a) Decoding without disturbance (b) Decoding with disturbance

Figure 7.13: Images of the video generated by the CAL MPEG 4 SP decoder application.

In the second experiment the sampling time is reduced and EP_{SP} is set to $[0.1, 0.15]$. Figure 7.14 shows the bandwidth adaption for both virtual processors of the application. A disturbance was also introduced between time $t = 200$ and $t = 220$. For this experiment each measurement point in the *Time interval* axis corresponds to a measurement done each 0.5 seconds. One can notice that the adaption in this case is much faster than in the previous experiment.

In the third experiment the sampling time is the same as in the previous experiment. The exhaustion percentage set point EP_{SP} is set to $[0.05, 0.1]$. Similar to the previous experiment the disturbance is present between time $t = 220$ and $t = 250$. In this case EP_{SP} is closer to an ideal situation of having a used bandwidth UB smaller than the assigned bandwidth AB during *all* the sampling intervals. Figure 7.15 shows the final results of the resources adaption for the virtual processors of the decoder application.

The outliers observed in the EP are caused by lack of synchronization between the activation time of the tasks within the virtual processors and the replenishing time of the reservation assigned bandwidth AB.

One can notice in the figure that the bandwidth controllers are able to keep the EP close to 0 most the time without wasting the

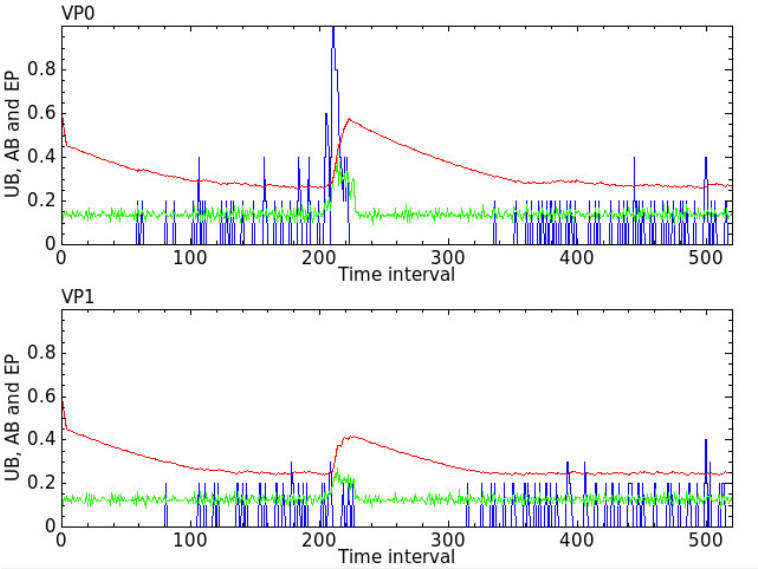


Figure 7.14: Resources adaption of the CAL SP decoder application for VP0 and VP1.

bandwidth resources. This means that the application does not need 120% of bandwidth in order to have a good performance.

Scenario 2: MPEG 4 SP Decoder and Pipeline Applications

For the second scenario the upper and lower bounds of the exhaustion percentage set point, that is, EP_{SL} and EP_{SU} were set to 0.1 and 0.15 respectively. These bounds were used for both applications. Figure 7.16 shows the used bandwidth UB, the assigned bandwidth AB and the exhaustion percentage EP signals of the two virtual processors $VP0$ and $VP1$ of the decoder application.

At time $t = 0$ the decoder application registers with the resource manager. Since there is no other application executing on the system, the resource manager assigns the highest service level 0 to the application, which corresponds to an initial assigned bandwidth AB equal to 0.6. After registration the bandwidth controllers adapt the assigned bandwidth AB in each of the VPs trying to keep the EP within EP_{SP}.

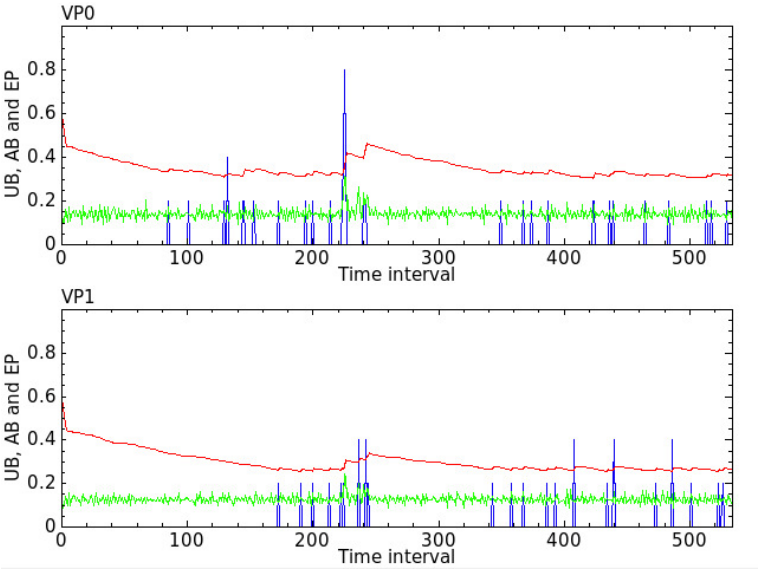


Figure 7.15: Resources adaption of the CAL SP decoder application.

If the EP is greater than 0.15 the bandwidth controllers increment the AB. The decoder application becomes unhappy at time $t = 10$ and $t = 210$ which causes the bandwidth controllers to increment the allocated bandwidth until the application is happy again. The periodic pipeline application registers with the resource manager at time $t = 240$. Since this application has higher importance than the decoder, the resource manager assigns service level 0 to the pipeline application and reduces the service level of the decoder application from 0 to 1. The initial assigned bandwidth of the decoder application at the new service level equals 0.5, which later on is decreased by the bandwidth controllers. Around time $t = 410$, the pipeline application unregisters, this increases the amount of free CPU resources, and triggers a new service level assignment for the decoder application, which in this case increases from service level 1 to service level 0.

It is important to remark that the time scale in the figure changes when the service level changes.

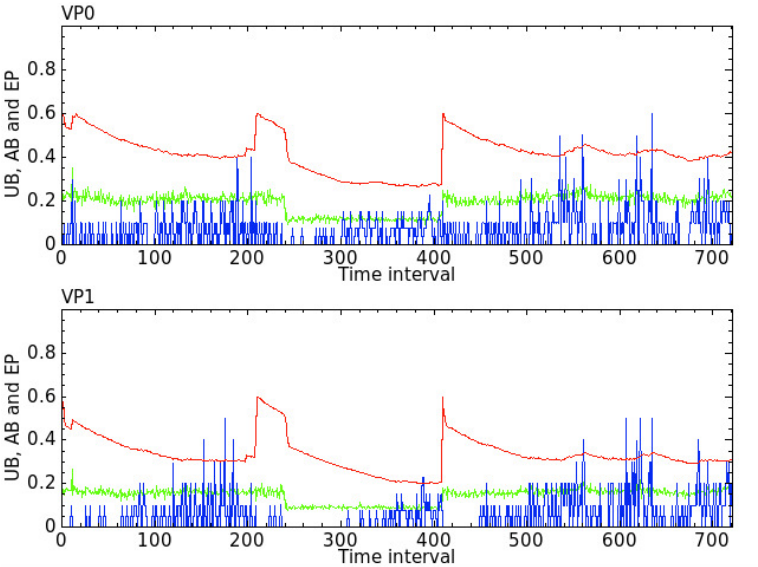


Figure 7.16: Resources adaption of the CAL SP decoder application.

8

Adaption and Learning

The temporal behavior of the registered applications is initially unknown to the resource manager. The only *available* information at this point for the resource manager is what is provided by the service level table of each application. However, the service level table must be considered just as an initial model of the application which is not completely accurate.

For the resource manager, the implemented feedback techniques provide in first place the means to adapt at runtime the resources provided to the registered applications. This guarantees that the performance criteria based on resource utilization and/or achieved QoS is always satisfied. In second place they also provide knowledge about the real amount of resources needed by the applications, which may differ from the initial information provided by the service level table.

8.1 Service Level Table Inaccuracy

The application developer specifies offline each of the values in the service level table. The information used by the developer to define these values includes the internal structure of the application, the level of interconnection and communication of the different components of this structure, the hardware platform, and the nature of the data to be handled.

Despite having a great deal of information about the internal topology and networking of the application, the information about the data

is something that can be certainly known only at runtime. Consider for instance the CAL MPEG 4 SP decoder application from Chapter 7. Depending on the nature of the decoded frames, the amount of resources needed may vary substantially for the same service level.

Therefore, the resource manager must be able to handle uncertainties in the initial model and to tune the values specified in it.

8.2 Resource Allocation Beyond Service Level Specifications

The lack of accuracy of the values defined in the service level table can produce two different scenarios. In the first scenario the application may require less resources than the ones initially specified. In such a case the bandwidth controllers can adapt the allocated resources to the real needs of the application, and reallocate the unused resources to other applications if needed.

In the second scenario the application may require more resources than the ones initially specified. This is the worst case scenario due to the performance criteria of the bandwidth controllers may not be satisfied. This means that although the bandwidth controllers provide the maximum assigned bandwidth, the application will not be able to have a good performance. This of course could reduce the provided QoS.

In order to avoid this problem, the resource manager must be able to allocate more resources than what is initially specified. This procedure must be done in a systematic way that considers the resource limitations of the system, and specifies the rules or policies under which more resources can be provided.

The maximum assignable bandwidth of each core of the system must be considered when increasing the assigned bandwidth. In order to satisfy the schedulability condition. The policies to increase the assigned bandwidth specify that:

- A virtual processor of an application can be assigned more resources if there is available free bandwidth.
- A virtual processor of an application can take bandwidth from other virtual processors that are less important, and are exe-

cuting with a bandwidth that is larger than what was initially assigned during registration.

To understand these policies a simple example is included. Figure 8.1 shows the additional bandwidth allocated for a virtual processor of application 1. In the figure BW_M is the maximum assignable bandwidth of the core, that is 90%, where the virtual processor is executing. The maximum assigned bandwidth AB_{M1} (dotted red line), corresponds to the initial bandwidth distribution value assigned during registration. At time t_0 the assigned bandwidth AB_1 corresponds

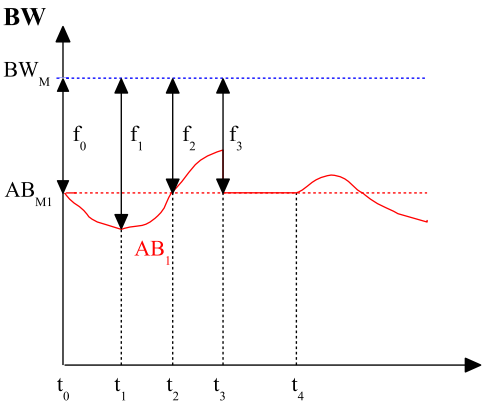


Figure 8.1: Bandwidth assignment beyond service level specifications.

to AB_{M1}, f_0 is the free available bandwidth which can be assigned if needed. The bandwidth controller continuously adapts AB_1, and at time t_1 the free available bandwidth increases. This free resource could be allocated to other virtual processor executing on the same core. The resource manager begins to allocate more resources to the application at time t_2. The virtual processor of a new registered application is assigned to the same core at time t_3, this forces AB_1 to return to AB_{M1}. After sometime at time t_4 the bandwidth controller of the new virtual processor release unused resources which are taken and allocated to the first application.

8.3 Service Level Table Update

In either of the two scenarios, the bandwidth controllers obtain runtime information about the real resource consumption of the applications. Thus, the resource manager is able to determine the adequate amount of resources needed by the application at an specific service level.

The next natural step would be to update some of the values in the service level table. These values include the bandwidth and the bandwidth distribution at each service level. A complete update of these values is only possible if the application has been assigned each service level at some point during its execution time. Otherwise only a partial update can be carried out.

The policies to update the bandwidth and bandwidth distribution values at a particular service level specify that:

- The application has been assigned that particular service level at least once during its execution time.
- The update is carried out after the application is assigned a new service level. The values to be updated correspond to the previous service level. These values are equal to the assigned bandwidth and bandwidth distribution prior to the new service level assignment.
- In case the same service level has been active more than once, the updated values correspond to the largest ones among the last three updates carried out for the same service level.

The last policy gives the possibility to discard old data that does not provide new information to the update process.

8.4 Example

This section will show the functionality of the service level table update for the CAL MPEG 4 SP decoder application. To force the service level change of the decoder application, the CAL periodic pipeline application is used.

The original service level table for both of the applications is shown in Table 8.1.

Table 8.1: Original service level table of the decoder and pipeline applications

Application name	I	SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
Decoder	1	0	100	120	100	[60, 60]
		1	80	100	330	[50, 50]
		2	60	40	400	[20, 20]
Pipeline	10	0	100	80	20	[40, 40]
		1	90	54	40	[27, 27]
		2	70	32	70	[16, 16]

Implementation Considerations

The experiment is carried out in a dual core system where the resource availability of each of the cores is set to 90%. The sampling time is set to $5P_i$, where P_i is the period of the application i that is being controlled. The exhaustion percentage set point EP_{SP} is set to $[0.05, 0.1]$. The tuning parameters of the bandwidth controllers are the same as in Section 7.3 (see Table 7.2). For a better visualization of the results the input and output signals of the bandwidth controller are normalized to values between 0 and 1.

Service Level Table Update

The resource adaption and bandwidth update of the decoder application are shown in Figure 8.2. In the figure the used bandwidth UB, the assigned bandwidth AB, and the exhaustion percentage EP are represented by the green, red, and blue colors respectively.

The decoder application registers with the resource manager at time $t = 0$, and gets service level 0. According to the service level table this means that each virtual processor gets an initial assigned bandwidth of 0.6. At time $t = 100$ a disturbance occurs which is counteracted by the bandwidth controllers by increasing AB. The application

becomes unhappy at time $t = 200$ this produces a new increase of AB until the application becomes happy again. Around time $t = 250$ the periodic pipeline application registers with the resource manager. Before the new service level assignment is performed, the resource manager updates the bandwidth distribution values and the total bandwidth for the decoder application at service level 0. The updated bandwidth distribution values of $VP0$ and $VP1$ correspond to 0.47 and 0.33 respectively.

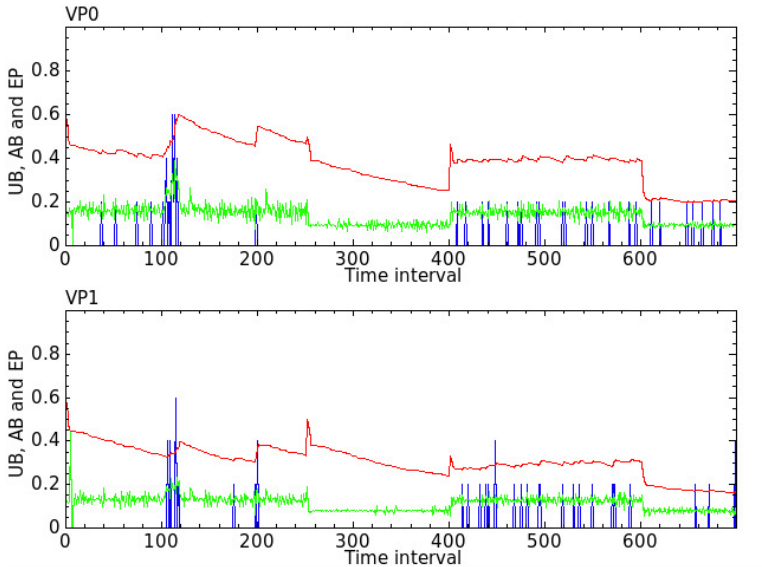


Figure 8.2: Resources adaption and bandwidth update of the CAL SP decoder application.

After registration of the pipeline application the resource manager assigns service level 0 to the pipeline and the service level of the decoder application to 1. The initial assigned bandwidth of the decoder at this service level equals 0.5 which later on is decreased by the bandwidth controllers. At time $t = 400$ the pipeline application unregisters. Before the new service level assignment is performed the resource manager updates the bandwidth distribution and the total bandwidth values for the decoder application at service level 1. In this case the

updated bandwidth distribution values of *VP*0 and *VP*1 correspond to 0.25 and 0.24 respectively. After the updating process the decoder is assigned service level 0. This time the initial assigned bandwidth of *VP*0 and *VP*1 correspond to 0.47 and 0.33 respectively.

For illustration reasons a new service level assignment is forced with the registration again of the pipeline application. This reduces the service level of the decoder from 0 to 1. In this case the initial assigned bandwidth of *VP*0 and *VP*1 correspond to the updated 0.25 and 0.24 and not to the original 0.5.

The update of the bandwidth distribution values as well as the total bandwidth for service level 0 and 1 are shown in Table 8.2.

Table 8.2: Updated service level table of the decoder application

Application name	I	SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
Decoder	1	0	100	80	100	[47, 33]
		1	80	49	33	[25, 24]
		2	60	40	100	[20, 20]

Once can notice in Figure 8.2 that the output of the bandwidth controllers after the update is more steady and almost constant. This is the result of having a model of the application that is tuned at runtime.

9

Adaption towards changes in resource availability

The resource manager is able to adapt how resources are distributed when the application requirements change and to adapt the applications to changes in resource availability. In this last case, it has so far been assumed that the available amount of system resources is constant. However, this is also subject to changes over time, specially if the system supports power management and/or thermal control.

Not only the available system resources may change dynamically, but also the significance that each application may have for the user at different points in time. This implies that the importance of an application with respect to others may change dynamically.

9.1 Changing Resource Availability

The system power consumption can become very significant when the total computational load generated by the registered applications is too high. This will increase the temperature of the system chips. One way to prevent failures due to overheating is to limit the computational load or utilization of the system.

The approach described in [52] which uses a single core platform, combines a PI controller for thermal control of the chip and the resource manager described in the previous chapters. Figure 9.1 shows the proposed system model. The thermal controller keeps the tempera-

ture at an acceptable temperature for the processor. The resource manager dynamically allocates resources to each application on the system. In the figure T and T_R are the current temperature of the system, and the reference temperature respectively. This last value is defined by the system designer. The values U, U_{min}, U_{max} and U_L correspond to the utilization of the system, the lower and upper utilization bounds and the utilization limit defined by the thermal controller respectively.

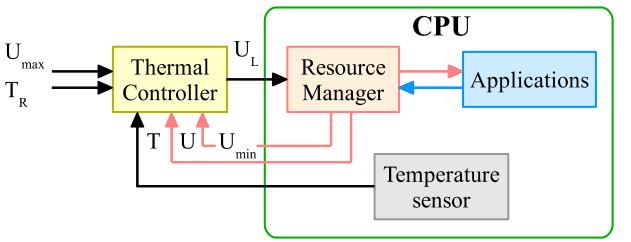


Figure 9.1: System model for thermal control in a single core.

The input U_{min} represents the minimum utilization required by the applications to provide the lowest permissible QoS. The input U_{max} is the maximal available utilization defined by the employed scheduling policy. The output of the thermal controller U_L decides the maximum amount of bandwidth available to the resource manager for allocation to applications. A change in U_L could trigger new service level assignments for the registered applications.

The extension of this approach to multicore systems would require the implementation of thermal controllers for each of the cores. The resource manager would require to dynamically pack the virtual processors onto as few physical processors as possible. This would make it possible to turn off cores. The functionality for this, that is, to be able to dynamically migrate virtual processors and their tasks is already available and was explained in Chapter 6.

The service level assignment, the bandwidth distribution as well as the bandwidth adaption functionalities would still be used on this extension. The only difference is that they would be subject to the constraint defined by U_L.

9.2 Changing Application Importance Values

The significance that the user may give to the running applications may also change over time. This means that the user may want to change the importance values of the registered applications at runtime. In this case the resource manager may have to redistribute the resources. This will possibly require new service level assignments for the registered applications, and change how the bandwidth is distributed among the cores.

9.3 Example

This section will show the results obtained when the resource availability as well as the application importance values are subject to changes. The chosen scenario contains three applications A1, A2 and A3. Table 9.1 shows the service level information provided by the three

Table 9.1: Service level table of application A1, A2 and A3

Application name	I	SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
A1	10	0	100	160	40	[40, 40, 40, 40]
		1	80	120	50	[30, 30, 30, 30]
A2	1	0	100	110	20	[20, 30, 30, 30]
		1	90	55	40	[10, 15, 15, 15]
		2	70	35	70	[5, 10, 10, 10]
A3	100	0	100	75	20	[20, 15, 40]
		1	70	60	100	[10, 10, 30]

applications and their importance values in column I. Originally the resource availability of each of the cores in the four core system is set to 90%.

Figure 9.2 shows the assigned service level and the bandwidth distribution of each of the applications after their registration with the

resource manager. The bandwidth distribution policy is packed distri-

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	1	55	[10, 15, 15, 15]
A3	0	75	[20, 15, 40]

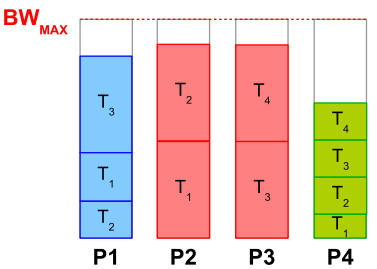


Figure 9.2: Registration of applications A1, A2 and A3.

bution.

Changing Resource Availability

For the first experiment the resource availability of the processor 4 is set to 0. This change could be generated by a power management controller. Figure 9.3 shows how application A2 receives a new ser-

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	2	35	[5, 10, 10, 10]
A3	0	75	[20, 15, 40]

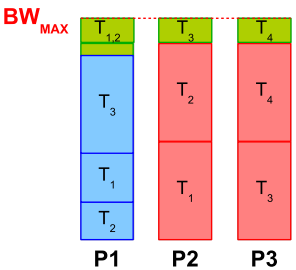


Figure 9.3: New distribution and service level assignments of the applications after changes in resource availability.

vice level. This is not surprising since this application is the one that contributes the least to the overall QoS. After the new service level assignment migration of the virtual processors of A2 to the other processors is possible.

Changing Application Importance Values

For the second experiment the importance values of A1, A2 and A3 are changed to 100, 10 and 1 respectively. This means that the importance of application A2 is increased. This produces a new service level assignment for A2 which increases to 0, and for A3 which decreases to 1. The new bandwidth distribution is shown in Figure 9.4.

App. name	SL	BW [%]	BWD [%]
A1	0	160	[40, 40, 40, 40]
A2	0	110	[20, 30, 30, 30]
A3	1	40	[10, 10, 30]

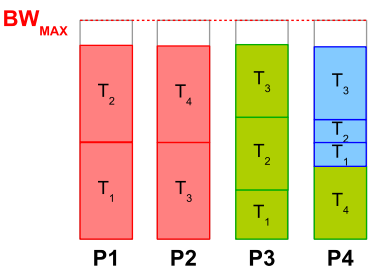


Figure 9.4: New distribution and service level assignment of the applications after changes in the importance values.

10

Application Examples

This chapter briefly describes different applications developed based on the resource management framework described in Chapters 3-9. The applications were all developed as demonstrators in the ACTORS project. Most of the applications were implemented in CAL.

The main objective for implementation of the demonstrators was to evaluate the capacity and the performance of the resource manager. Of course this also includes evaluation of the performance achieved by the applications under the control of the resource manager.

10.1 Video Decoder Demonstrator

The camera decoder application consists of an MPEG 4 SP decoder written in CAL that is connected to an Axis M1011 network camera capable of generating SP encoded video streams and where the frame rate and the resolution can be changed dynamically. This application is the same that has been used in Chapters 7, and 8.

The video frames are received over the network using a special system actor that extracts the SP frames from the Real-Time Transport Protocol (RTP) transport format generated by the camera and which also issues commands to the camera to change the frame rate and resolution. High frame rate and high resolution both implies a higher resource demand for the decoding.

The camera decoder application is considered to be a low importance process, its importance value is set to 100.

10.2 Video Quality Adaption Demonstrator

The video quality adaption demonstrator consists of a video player client executing under the control of the resource manager. The video player can either be implemented in CAL or be a legacy media player. The video stream is received over the network from a video server. When the available resources for the decoding decrease and it needs to lower its service level it issues a command to the video server to adapt the video stream by skipping frames, in the case of MPEG 2 streams [53], or by skipping macro block coefficients in the case of MPEG 4 streams.

10.3 Feedback Control Demonstrator

The feedback control demonstrator [54] consists of the following applications which all are executing under the control of the resource manager:

- A ball and beam controller implemented in CAL. Two instances of this application are used.
- An inverted pendulum balancing and a swing-up controller implemented in CAL. The actuator for the pendulum is an ABB industrial robot.
- A CAL MPEG 4 SP video decoder in combination with an Axis network camera which has already been described in Chapter 7.
- A GUI for the resource manager implemented in `C++`.
- An external load generator implemented in `C++`. The load generator is a compute-bound application that consumes all CPU cycles given to it. It is used to generate disturbing computing load on the system.
- A CAL pipeline application described in Chapter 7. This application is also used to generate disturbing computing load on the system.

Figure 10.1 shows a schematic overview of the demonstrator.

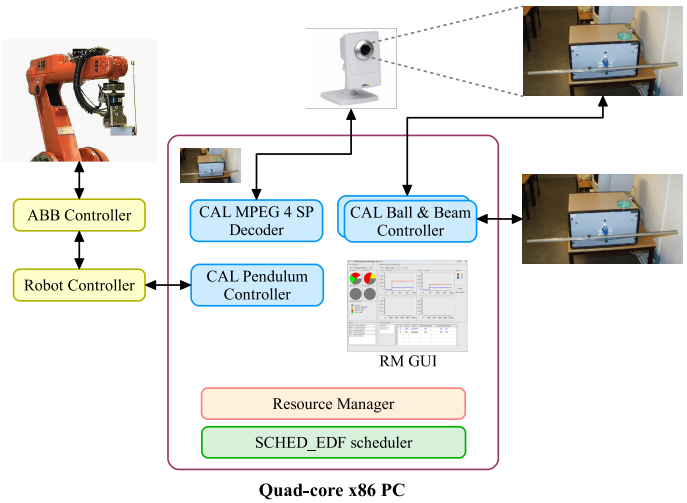


Figure 10.1: Overview of the control demonstrator.

The Ball and Beam Controller

The ball and beam process [55] consists of a horizontal beam and a motor that controls the beam angle. The measured signals from the process are the beam angle relative to the horizontal plane and the position of the ball. Figure 10.2 shows the process.

Figure 10.2: The ball and beam process.

The dynamic model from the motor to the ball position consists of two transfer function blocks connected in series, in which the beam angle appears as an intermediate output signal (see Figure 10.3).

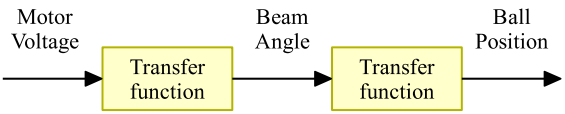


Figure 10.3: Ball and Beam Model Structure.

The aim of the control system is to control the position of the ball on the beam. Due to the dynamic of the process a cascade controller is used.

The CAL implementation of the controller includes different actors:

- The D-Bus actor acts as an interface to the resource manager.
- The Service Level actor translates the service level into a suitable sampling period, that is, in this application different service levels correspond to different sampling periods. A high service level implies a short sampling period which in turn results in a high bandwidth and high QoS obtained.
- The Exit actor implements functionality for terminating the application using a keyboard command.
- The Merge actor merges together the sampling period from the Service Level actor and a token from the Exit actor and forwards the tokens to the Clock actor.
- The reference signal for the outer controller is a low frequency square-wave signal. This is generated by a separate clock system actor (RefGen Clock) and a reference signal generator (RefGen).
- The clock, position, outer controller, angle, inner controller, and output actors constitute the cascade controller of the process.

The complete CAL implementation of the ball and beam controller is shown in Figure 10.4.

The service level table of the ball and beam application is shown in Table 10.1. The task to be performed is the same for all service levels.

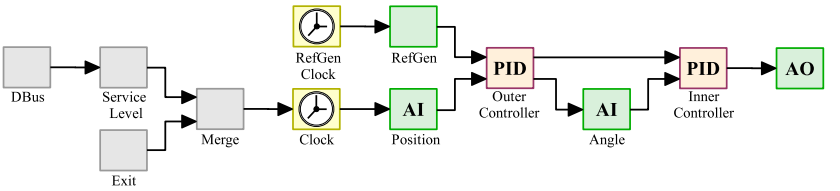


Figure 10.4: Ball and Beam CAL Model.

A service level change produces a change in the controller parameters, such that the controller is designed with respect to the new period. The change in the parameters is necessary to have a stable closed loop system. Due to the structure of the CAL network no parallel computations are needed, and therefore only one core is used. The ball and beam application is considered to be a medium importance process with an importance value of 20.

Table 10.1: Service level table for the ball and beam controller application

SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
0	100	30	20	30
1	90	20	30	20
2	70	12	50	12
3	40	9	70	9

The controller does not report any explicit happiness value to the resource manager. This implies that the application is always happy as long as it is allowed to execute at one of the specified service levels. It is, however, straightforward to extend the implementation with functionality for calculating the quality of control achieved. One possibility is to use an actor that takes the control signal and the measured ball position as inputs and calculates a quadratic cost function. The value of this cost function is then sent to a Happiness actor that translates the value of the cost function into a happiness value taking the current service level into account. The happiness value would then be sent as

an input to the D-Bus actor that would send a message to the resource manager.

The Inverted Pendulum Controller

The inverted pendulum controller [55] consists of a free swinging pendulum that is attached to an ABB IRB 2400 industrial robot. The inverted pendulum actuated by the industrial robot is shown in Figure 10.5

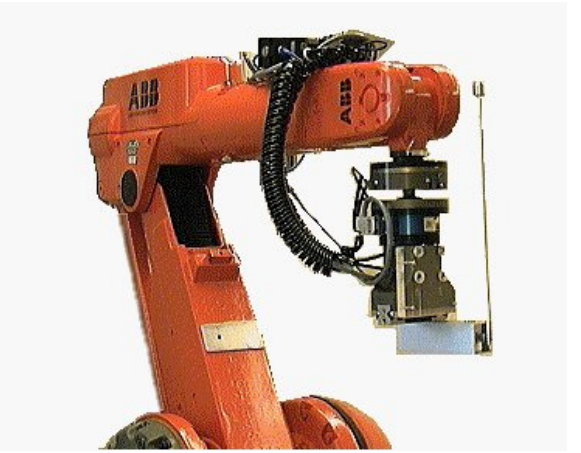


Figure 10.5: Inverted pendulum actuated by an industrial robot.

The objective of the CAL controller is to automatically swing-up the pendulum and then balance the pendulum in its upward position. The pendulum controller consists of four main parts:

- Signal processing logic for calculating the angular velocity of the pendulum from the angle measurement.
- The balancing controller that balances the pendulum in the upward position. This controller is a state feedback controller using four states: the cart position, the cart velocity, the pendulum angle, and the pendulum angular velocity.

- The swing-up controller. This controller automatically swings up the pendulum from the downward position to the upward position by gradually pumping in more and more energy into the pendulum.
- Mode selection logic for deciding which one of the balancing controller and the swing-up controller that should be connected to the cart.

The pendulum controller is shown in Figure 10.6. The Sampling

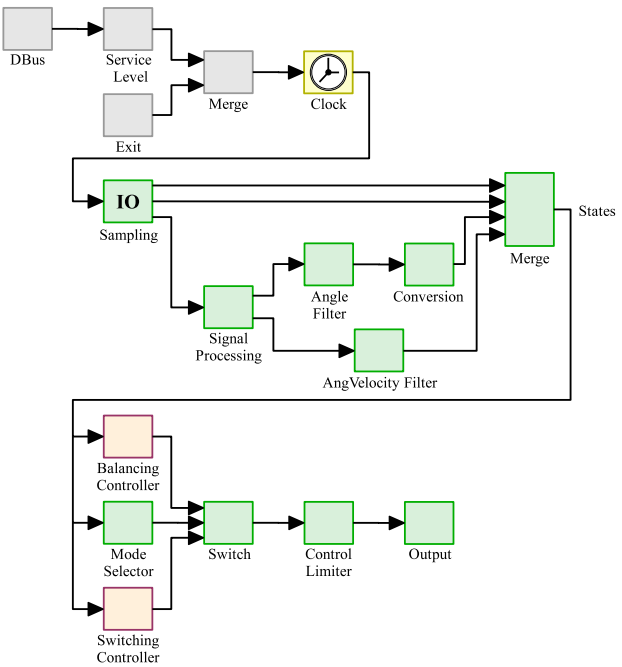


Figure 10.6: Inverted Pendulum CAL Model.

actor takes a sample of the robot arm and velocity, and of the pendulum angle. From the angle the angular velocity is calculated through a simple difference approximation. The resulting four state variables are merged together and sent the the balancing controller, the mode

selector, and the swing-up controllers which are executed in parallel. The Switch actor selects the output of one of the two controllers based on the output of the Mode Selector actor. The limited control signal is then sent out to the physical pendulum process. Also, in this example some of the connections have been omitted for the sake of clarity.

The inverted pendulum controller is considered as the most important process, its importance value is set to 100. The service level table is shown in Table 10.2.

Table 10.2: Service level table for the inverted pendulum controller application

SL	QoS [%]	BW [%]	Granularity [ms]	BWD [%]
0	100	40	10	[8, 16, 16]
1	90	20	20	[4, 8, 8]
2	70	10	40	[2, 4, 4]
3	40	5	80	[1, 2, 2]

The calculations in the CAL network are done in parallel. The clock actor that triggers the controller, together with the actors that handle the D-Bus communication execute in the first virtual processor. The kinematics to obtain the pivot position are calculated in the second virtual processor, while the signal processing of the angle measurement is done in the third virtual processor. When all the states are obtained they are sent to the swing-up and balancing controller (see Figure 10.6) which execute in parallel in the second and third virtual processors.

The inverted pendulum controller does not report any happiness to the resource manager, that is, it is assumed to always be happy.

The Resource Manager GUI

The resource manager GUI is implemented in `C++`, and is used to visualize the internal actions of the resource manager. It runs itself under the control of the resource manager using a single service level and a single virtual processor with a default bandwidth of 15%, a granularity of 10000 ms and an importance value of 10.

11

Conclusions

11.1 Summary

The central theme of this thesis is adaptive CPU bandwidth resource management for applications executing on multicore platforms. The work focuses on the development and implementation of different algorithms for the resource manager part of the ACTORS framework. The framework uses the fairly abstract concepts of service levels and happiness to interface the applications with the resource manager. The interface between the resource manager and the operating system is based on reservation parameters and resource utilization measurements.

The implemented algorithms combine feedforward and feedback techniques. As a result the resource manager is able to adapt the applications to changes in resource availability, and to adapt how the resources are distributed when the application requirements change. Some remarks about the outcomes of this thesis are given below.

Service Level Assignment and Bandwidth Distribution

An algorithm that uses feedforward techniques is presented. The algorithm proposes a BIP formulation to assign service levels to the applications. The assignment is done according to their bandwidth requirements and QoS provided at each service level, as well as their importance values. The formulation is very simple and uses little information to produce a solution. The lack of more detailed information

may lead to solutions that are not schedulable, this is specially noted during the bandwidth distribution process.

Different distribution policies are proposed and implemented to perform the bandwidth distribution of the registered applications. Each policy produces a particular mapping onto the physical cores of the virtual processors of an application. This is always possible when a schedulable solution is produced during the service level assignment.

Different algorithms are presented to handle unschedulable solutions. The algorithms include a repetitive service level assignment method and a bandwidth compression and decompression algorithm. The first one solves the problem in a very simple but non optimal way. The second one, more complex in nature, provides a better solution.

Bandwidth Adaption and Learning Process

An algorithm that implements bandwidth controllers based on feedback techniques is presented. The resource manager assigns one bandwidth controller per virtual processor of every application. Each bandwidth controller dynamically adapts the allocated CPU resources. The bandwidth controllers are periodically activated. The adaption is performed based on resource utilization and/or achieved QoS feedback.

For the resource utilization feedback a bandwidth cascade controller structure is employed. The output of the controller is generated based on the cumulative measurements of the used budget and exhaustion percentage, as well as statistical measurements. For the achieved QoS feedback a simple proportional controller is used. This controller produces an output based on the happiness measurements directly provided by the application.

The bandwidth controllers guarantee that the allocated resources are optimally used and not wasted. Additionally, the bandwidth controllers provide knowledge about the real amount of resources needed by the applications, which may highly contrast with the initial information provided by the service level table. Thus, they are able to produce a model of the application that is tuned at runtime.

Adaption Towards Changes in Resource Availability

The different implemented algorithms are able to perform adaption towards changes in resource availability. This is very relevant specially

for systems that provide support for power management and/or thermal control. The algorithms are also able to handle changes related to the significance that each application may have for the user at different points in time.

11.2 Future Work

The work presented in this thesis can be continued in several directions. Some of the more interesting ones are the following:

Support for power management and/or thermal control The current functionality of the resource manager is to a large extent already prepared for this. A possible approach is to use a cascaded structure where an outer power or thermal controller decides how much CPU resources that the resource manager may use to allocate to applications. The thermal controller described in Chapter 9 uses this approach, but only in the single-core case. Accurate multicore thermal control requires sensors that measure the temperature of the individual cores as well as a thermal controller that controls the amount of resources that may be allocated on a per core basis. A possible approach to include power management in the system would be to add terms to the cost function in the service level optimization that allows individual cores to be either active or inactive.

Multi-resource management The current resource manager only manages the CPU time. An interesting extension would be to also allow management of other resources, for example, memory. The service level table format was initially developed to support multiple resources. The idea was to use periodic server abstractions for all resources and to express the bandwidth and granularity requirements on a per resource basis.

Model-free resource adaptation The current resource manager requires the application developer to provide estimates of the resource requirements of the application at each service level and for the particular hardware platform that the application should execute on. This information can be viewed as a model of the application that is used

in the service level optimization and the bandwidth distribution. However, this approach has certain drawbacks. In addition to the practical problems associated with deriving this information it also limits the application portability from one platform to another. An alternative approach would be to instead base the resource adaptation only on feedback from the measurements of the resource consumption and the application happiness. The bandwidth requirement and the QoS information in the service level table could still be used, but should now be interpreted as relative values that the resource manager may use to, for example, decide whether to switch service level of an application, rather than as absolute values. A problem with a purely feedback-based approach is to decide how much bandwidth that an application should receive initially.

12

Bibliography

[1] *ACTORS: Adaptivity and Control of Resources in Embedded Systems*. April 2008. `http://exoplanet.eu/catalog.php`.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, 1973.

[3] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[4] L. Abeni and G. Buttazzo, "Resource reservations in dynamic real-time systems," *Real-Time Systems*, vol. 27, no. 2, pp. 123–165, 2004.

[5] L. Abeni, G. Lipari, and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *6th IEEE International Conference on Multimedia Computing and Systems*, p. 107, June 1999.

[6] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, 1993.

[7] D. Petrou, J. W. Milford, and G. A. Gibson, "Implementing lottery scheduling: matching the specializations in traditional schedulers," in *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, 1999.

[8] L. Abeni, C. Scordino, G. Lipari, and L. Palopoli, "Serving non real-time tasks in a reservation environment," in *Proceedings of the 9th Real-Time Linux Workshop (RTLW)*, November 2007.

[9] C. Scordino, *Dynamic Voltage Scaling for Energy-Constrained Real-Time Systems*. PhD thesis, Computer Science Department, University of Pisa, December 2007.

[10] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *SPIE/ACM Conference on Multimedia Computing and Networking*, pp. 150–164, January 1998.

[11] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith, "Multicore resource management," *IEEE Micro*, vol. 28, pp. 6–16, May 2008.

[12] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing - Real-Time Systems (Euromicro RTS-03)*, vol. 1, April 2005.

[13] C. Lu, J. Stankovic, G. Tao, and S. H. Son", "Design and evaluation of a feedback control edf scheduling algorithm," in *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 56–67, December 1999.

[14] L. Palopoli, L. Abeni, and G. Lipari, "On the application of hybrid control to cpu reservations," in *Proceedings of the Conference on Hybrid Systems Computation and Control (HSCC03)*, pp. 389–404, April 2003.

[15] L. Abeni and G. Buttazzo, "Adaptive bandwidth reservation for multimedia computing," in *6th IEEE Conference on Real-Time Computing Systems and Applications*, pp. 70–77, December 1999.

[16] W. Knight, "Two heads are better than one [dual-core processors]," *IEEE Review*, vol. 51, no. 9, pp. 32–35, 2005.

[17] M. Johson, *Superscalar Microprocessor Design*. New Jersey, USA: Prentice Hall, 1991.

[18] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. New York, NY, USA: McGraw-Hill, 2004.

[19] C. L. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment," *JPL Space Programs Summary 37–60*, vol. 2, pp. 28–31, 1969.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W.H. Freeman & Co., 1979.

[21] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Transactions on Computers*, vol. 44, pp. 1429–1442, December 1995.

[22] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operation Research*, vol. 26, pp. 127–140, January 1978.

[23] S. Lauzac, R. Melhem, and D. Mossé, "An improved rate-monotonic admission control and its applications," *IEEE Transactions on Computers*, vol. 52, pp. 337–350, March 2003.

[24] J. M. López, M. García, J. L. Díaz, and D. F. García, "Utilization bounds for multiprocessor rate-monotonic scheduling," *Real-Time Systems*, vol. 24, pp. 5–28, January 2003.

[25] L. Rizvanovic, D. Isovic, and G. Fohler, "Integrated global and local quality-of-service adaptation in distributed, heterogeneous systems," in *The 2007 IFIP International Conference on Embedded and Ubiquitous Computing*, December 2007.

[26] L. Rizvanovic and G. Fohler, "The MATRIX: A framework for real-time resource management for video streaming in networks of heterogeneous devices," in *The International Conference on Consumer Electronics 2007*, pp. 219–233, January 2007.

[27] T. Cucinotta, L. Palopoli, L. Marzario, and G. Lipari, "AQoSA - adaptive quality of service architecture," *Software - Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2008.

[28] *AQuoSA: Adaptive Quality of Service Architecture*. November 2005. `http://aquosa.sourceforge.net/index.php`.

[29] A. Kassler, A. Schorr, C. Niedermeier, R. Schmid, and A. Schrader, "MASA - a scalable qos framework," in *Proceedings of Internet and Multimedia Systems and Applications (IMSA)*, August 2003.

[30] B. Li and K. Nahrstedt, "A control-based middleware framework for quality-of- service adaptations," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1632–1650, 1999.

[31] B. Li and K. Nahrstedt, "Impact of control theory on QoS adaptation in distributed middleware systems," in *Proceedings of the American Control Conference*, pp. 2987–2991, June 2001.

[32] J. Stankovic, T. Abdelzaher, M. Marleya, G. Tao, and S. Son, "Feedback control scheduling in distributed real-time systems," in *Proceedings of the Real-Time Systems Symposium (RTSS)*, p. 59, December 2001.

[33] S. Craciunas, C. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Programmable temporal isolation through variable-bandwidth servers," in *Proceedings of the Symposium on Industrial Embedded Systems (SIES)*, pp. 171–180, July 2009.

[34] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pp. 11–16, April 2008.

[35] G. C. Buttazzo, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, pp. 289–302, March 2002.

[36] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, 2004.

[37] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. R. Segovia, and C. Scordino, "Resource management on multicore systems: The actors approach," *IEEE Micro*, vol. 31, no. 3, pp. 72–81, 2011.

[38] J. Eker and J. Janneck, "CAL language report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, 2003.

[39] E. A. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.

[40] E. A. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[41] C. W. Mercer, R. Rajkumar, and H. Tokuda, "Applying hard real-time technology to multimedia systems," in *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, November 1993.

[42] G. Lipari and C. Scordino, "Linux and real-time: Current approaches and future opportunities," in *IEEE International Congress ANIPLA*, November 2006.

[43] N. Manica, L. Abeni, L. Palopoli, D. Faggioli, and C. Scordino, "Schedulable device drivers: Implementation and experimental results," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 53–62, July 2010.

[44] *D-Bus*. `http://www.freedesktop.org/wiki/Software/dbus`.

[45] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *Proceedings of the 22nd IEEE Real-Time System Symposium (RTSS)*, pp. 151–160, December 2001.

[46] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, MA, USA: Cambridge University Press, 2004.

[47] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[48] R. E. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Mathematical Society*, vol. 64, pp. 275–278, 1958.

[49] J. F. Shapiro, "Group theoretic algorithms for the integer programming problem 2: Extension to a general algorithm," *Operations Research*, vol. 16, no. 5, pp. 928–947, 1968.

[50] V. R. Segovia and K.-E. Årzén, "Towards adaptive resource management of dataflow applications on multi-core platforms," in *Proceedings Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems, ECRTS*, pp. 13–16, July 2010.

[51] *GLPK: GNU Linear Programming Kit*. http://www.gnu.org/s/glpk/.

[52] V. R. Segovia, M. Kralmark, M. Lindberg, and K.-E. Årzén, "Processor thermal control using adaptive bandwidth resource management," in *Proceedings of the 18th World Congress of the International Federation of Automatic Control, IFAC*, pp. 123–129, August 2011.

[53] A. Kotra and G. Fohler, "Resource aware real-time stream adaptation for MPEG-2 transport streams in constrained bandwidth networks," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, pp. 729–730, July 2010.

[54] M. Kralmark and K.-E. Årzén, *Deliverable D5b: Control Demonstrator*. January 2011. http://www3.control.lth.se/user/karlerik/Actors/M36/d5b-main.pdf.

[55] K.-E. Årzén, M. Kralmark, and J. Eker, *Deliverable D5d: Control Algorithms: Dataflow Models of Control Systems*. January 2011. http://www3.control.lth.se/user/karlerik/Actors/M36/d5d-main.pdf.

Department of Automatic Control
Lund University
Box 118
SE-221 00 Lund Sweden

Document name
LICENTIATE THESIS

Date of issue
September 2011

Document Number
ISRN LUTFD2/TFRT--3252--SE

Author(s)
Vanessa Romero Segovia

Supervisor
Karl-Erik Årzén

Sponsoring organisation
ACTORS (EU FP7)

Title and subtitle
Adaptive CPU Resource Management for Multicore Platforms

Abstract

The topic of this thesis is adaptive CPU resource management for multicore platforms. The work was done as a part of the resource manager component of the adaptive resource management framework implemented in the European ACTORS project. The framework dynamically allocates CPU resources for the applications. The key element of the framework is the resource manager that combines feedforward and feedback algorithms together with reservation techniques. The reservation techniques are supported by a new Linux scheduler through hard constant bandwidth server reservations. The resource requirements of the applications are provided through service level tables. Dynamic bandwidth allocation is performed by the resource manager which adapts applications to changes in resource availability, and adapts the resource allocation to changes in application requirements. The dynamic bandwidth allocation allows to obtain real application models through the tuning and update of the initial service level tables.

Key words
resource management, embedded systems, real-time systems, multimedia, multicore

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
120

Recipient's notes

Security classification