Adaptivity & Control of Resources in Embedded Systems

D3a

COOPERATION

State Abstractions

Responsible: **TU Kaiserslautern** (TUKL)

Karl-Erik Årzén (ULUND), **Enrico Bini** (SSSA), **Johan Eker** (EAB),
Gerhard Fohler (TUKL), **Alexander Neundorf** (TUKL)

Project Acronym: **ACTORS**
Project full title: **Adaptivity and Control of Resources in Embedded Systems**
Proposal/Contract no: **ICT-216586**
Project Document Number: **D3a 1.1 (Intermediate release)**
Project Document Date: **2009-01-31**
Workpackage Contributing to the Project Document: **WP3**
Deliverable Type and Security: **R-PU**

Contents

Chapter 1

Introduction

The ACTORS project is focused on resource-constrained devices, where multiple data flow applications with real-time demands run concurrently and compete for the available resources. Data Flow is a computational model that is specially suited for multimedia applications such as streamed video or audio applications, e.g., MPEG decoders and encoders. However, it is also applicable to, e.g., signal processing, feedback control, and computer vision.

The data flow parts of these applications will be written in the CAL language. CAL is a data flow programming language, where so called actors work on a stream of data tokens which flow between them [1]. Through this approach the data dependencies are explicitly codified in the software. This makes it possible to automatically partition potentially parallel parts of the software. This is important since the trend in CPU development is towards multicore systems. In order to fully exploit the available processing power on such systems it is necessary to parallelize the software. Writing parallel software is hard, and even harder for many-core systems with a number of cores » 2. Using a data flow language as CAL takes this burden away from the developer and the parallelization is done by the compiler and the tool chain.

Without special means threads or processes which run in parallel on one or more cores will compete for the available resources, such as CPU time. In order to protect threads or processes from influencing and disturbing each other, which can potentially decrease the overall quality of service of the system, in ACTORS there will be resource reservations provided by the operating system. These reservations will reserve CPU time exclusively for use by the designated threads, so these threads are to some degree isolated from each other.

In order to achieve a maximum overall quality of the system there will be a system-global resource manager. This resource manager will have knowledge about the resource requirements of the participating applications and distribute the resources among them. This means it will decide about the reservation sizes for the applications, notify the applications about this and adjust the reservations provided by the operating system accordingly. The resource manager will not react to minor temporary fluctuations, but only to major structural changes. There will be two types of applications: applications which offer discrete quality levels which they can provide, and applications which offer a continuous mapping between available resources and achieved quality. Both types will be supported in ACTORS. How their properties are represented is discussed in this deliverable.

The ACTORS runtime architecture uses Linux as the operating system, optionally extended by patches which enable resource reservation.

Deliverable context

This deliverable is the result of task 3.1 “System State Abstractions for Control”, which is part of workpackage 3 “Adaptive Resource Management”. This is an intermediate version of the deliverable, the final version will be released towards the end of the ACTORS projects.

This deliverable serves two purposes, first it gives an overview of the ACTORS runtime architecture, and second, it introduces the ACTORS resource manager. The current version (1.1) contains an updated description of the overall ACTORS architecture including the run-time architecture, compared to Version 1.0. It also defines the terminology used in ACTORS. The document should be considered as a working document that will be continuously updated as the ACTORS project proceeds. However, also in its current version it is still the deliverable that gives the best overview of the ACTORS run-time architecture. As such, this deliverable is the basis for understanding how the other components in the ACTORS project fit together and how they will work together.

Together with deliverables D1f “Interface Specification” and D4b “RBS Specification” this deliverable documents the resource manager, how application properties related to their real-time demands will be represented and how the resource reservations will be realized by the operating system. The recommended order of reading is D3a, D1f, and D4b. The results of this deliverable are necessary for all other tasks of workpackage 3, since they all are closely related to the resource manager. Beside those, also tasks 1.5 “Requirements/QoS Interface” and 4.2 “Matching Reservation Schemes with System States”, which deal with the interfaces to the resource manager, depend on this deliverable. In workpackage 2 “Actor Code Generation” mainly task 2.2 is related to this deliverable. In task 2.2 “OpenDF Extensions” the runtime for executing CAL applications will be developed. In this runtime system actors which communicate with the ACTORS resource manager have to be implemented.

Chapter two introduces the overall architecture and components of an ACTORS system. This is necessary for the understanding of the whole project. Chapter three documents the application interface of the resource manager and how it will be used to represent requirements of real-time applications. Chapter four shows how we envision how this interface can be used by different types of applications.

Chapter 2

Runtime Architecture of ACTORS

2.1 Architecture Overview

The ACTORS run-time architecture consists of three major components, which will be discussed in the following sections:

- CAL Applications
- Resource Manager
- Operating System

CAL Applications

The embedded devices considered in ACTORS will typically contain a mixture of different application types. Some applications will have very soft, or no, real-time requirements whereas others will have hard real-time requirements. Some applications will be implemented in CAL whereas others will be implemented using conventional techniques. Some applications will be aware of the ACTORS resource reservation framework and may interface to it, whereas other applications are completely ignorant of this. The long-term ambition of ACTORS is to be able to host all of these applications, i.e., it should be possible also for non-CAL applications to interface to the resource reservation framework. However, to begin with the focus will be ACTORS-aware CAL-applications. This is what will be described in this report.

A CAL application is an application that is written in CAL. The basic abstraction in CAL is an *actor*. Actors communicate asynchronously with other actors by consuming tokens from input ports and producing tokens at output ports. The output port of one actor may be connected to the input port of another actor via a FIFO buffer, forming a network of actors, i.e., a *CAL network*. The computations within an actor are performed through a sequence of firings. In each firing the actor may consume tokens from input ports, may modify its internal state, and may produce tokens at output ports. The computations performed within a firing are defined by an *action*. An actor, hence, internally consists of one or several actions. In general, the order in which the actions should be fired can only be determined dynamically. The reason for this can, e.g., be data-dependent actions.

The level of expressiveness of CAL is high in the sense that it is possible to create CAL networks that correspond to a variety of different data flow model classes e.g., synchronous data flow (SDF) models [2] and dynamic data flow (DDF) models [3]. Of particular interest here is the SDF model. In a SDF network the number of tokens consumed and produced during each firing is constant. This means that

it is possible to determine the firing order statically. In this case actors in the network can be merged into a single actor and the intermediate FIFO buffers can be removed. Also if an entire network is not SDF it is common that it contains regions that are SDF, so called *statically schedulable regions*. In that case the actors within a region can be merged. It is also possible that a statically schedulable region only contains parts of one or several actors, i.e., some of the actions and their associated ports. Alternatively, it may be desirable to split an actor into several actors to better express fine-grained parallelism when the target platform is an FPGA. The CAL network after these transformations will be referred to as the *data flow graph*.

Identifying a statically schedulable region and merging actors or actions makes it possible to create sequential binary code that due to the removal of the intermediate buffers can execute faster on a single processor than to schedule the execution of the corresponding actors dynamically on the same processor. However, on a multi-core platform this may not necessarily be the case. Consider the example in Fig. 2.1. The example consists of four actors in series. The number associated with

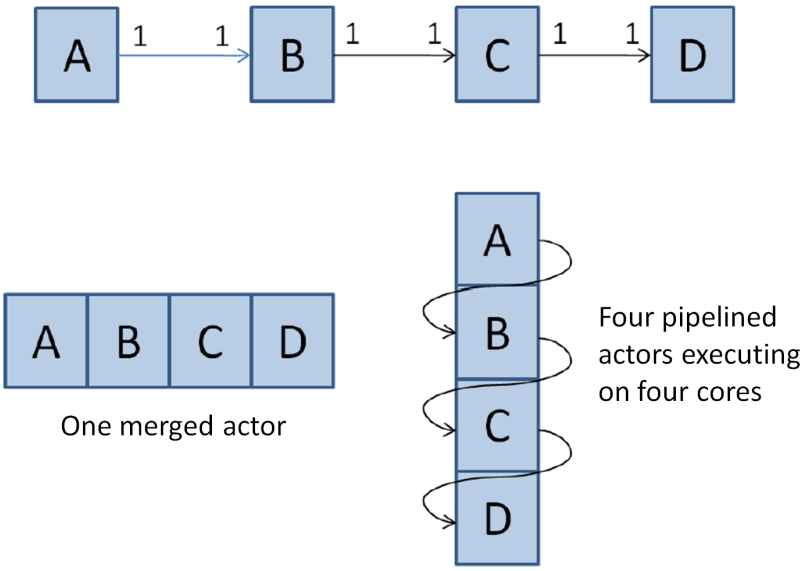

Figure 2.1: A CAL example consisting of four actors in series. The CAL network is a statically schedulable region that can be merged into a single actor that is executed on a single core. The actors can, however, also be kept separate and pipeline their execution on different cores.

the ports indicate the number of tokens consumed or produced. Since the network is SDF it can be merged into a single actor in which the actors A, B, C and D are connected in series. However, with four cores all the actors can be pipelined giving a throughput that theoretically is four times as high as in the single core case. Hence, whether to merge statically schedulable regions into single actors or not depends both on the application demands and on the execution platform characteristics. In ACTORS this decision is left to the model compiler developed in WP1, which, through the help of interaction from the human developer, will decide this and other related issues.

The *compiled actors* are the binary representations of of the actors in the data flow graph. A compiled actor is a set of instructions to be executed sequentially on a single processor. No parallelism is possible within a compiled actor. A *task* is an instance of a compiled actor with a unique set of ports and state.

In ACTORS a distinction is made between two types of CAL applications:

- dynamic CAL applications, and
- static CAL applications.

The dynamic case is the general case corresponding to, e.g, most multimedia streaming applications. Here the execution is highly data-dependent. Although statically schedulable regions may exist, it is not possible to schedule the entire CAL network statically. In, e.g., feedback control applications, however, the CAL networks are, in general, statically schedulable. In this case the data flow graph can be translated into a static precedence relation described by a Directed Acyclic Graph (DAG). This will be referred to as an *job precedence graph (JP)*. In general the data flow graph and the corresponding job precedence graph are not identical. When the job precedence graph is created actors in the data flow graph typically need to be duplicated. Also, the transformation from data flow graph to job precedence is not unique. Consider the example in Fig. 2.2. Here each execution of the *A* actor produces one token.

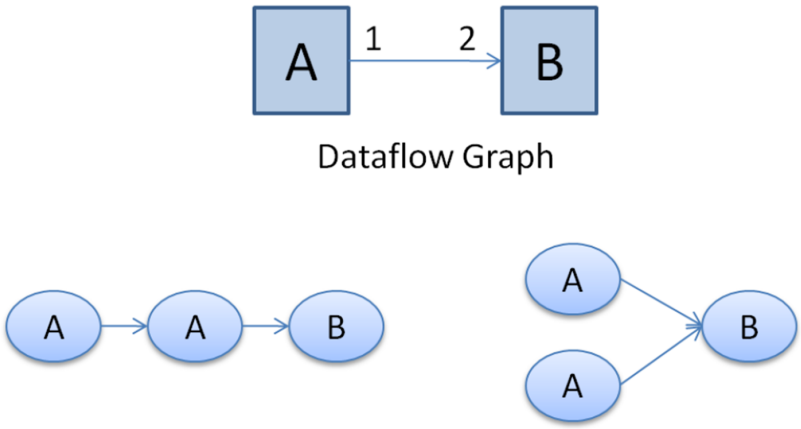

Figure 2.2: A CAL data flow graph for a static application. Depending on whether the *A* actor has internal state or not two different job precedence graphs are possible.

The *B* actor, however, needs two input tokens in order to execute. Hence, each execution of *B* must be preceded by two executions of *A*. If the *A* actor has internal state that is modified then the two *A* executions must be performed in series (the left case). If, not they may be performed in parallel (the right case). An invocation of a task is referred to as a *job*, hence the name job precedence graph.

The execution of a CAL application is governed by the associated *CAL runtime* system. The runtime system, or rather the application-dependent part of it, is generated by the model compiler. The execution model is different for the two main classes of CAL applications.

For static CAL applications with periodic or sporadic execution and where realistic WCET estimates are available it is possible to apply schedulability theory to check whether the applications will meet their deadlines or not. For these applications, the JP is known a priori and can be analyzed off-line as a part of the model compilation. The analysis produces the minimum number of serialized (totally ordered) sets of jobs, or *flows*, that can be accommodated on sequential machines,

with given computational demand. Each flow is allocated to a *virtual processor*, 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. The CAL runtime environment generates eligible jobs according to available tokens and allocate them to the proper virtual processor according to the corresponding flow (following a precomputed table).

In the case of dynamic CAL applications the run-time system consists of two parts, the *actor activator* and the run-time system dispatcher that selects which active actor to execute on which virtual processor. The actor activator activates actors as input data becomes available by marking them as ready for execution. The dispatcher repeatedly selects an active actor according to some policy and then executes it until completion on some virtual processor. For dynamic applications it is not useful to have more virtual processors than the number of cores available. One way of implementing the dispatcher is by having as many identical dispatcher threads as there are cores. Each of these threads executes within a dedicated virtual processor that is statically allocated to a particular core. The dispatcher threads executes in an infinite loop where they repeatedly select an active actor from the data flow graph and execute it until completion. During execution of the actor new tokens will be generated at output ports which will in turn activate new actors.

The different execution models for static and dynamic CAL applications have certain implications. For example, for static applications different invocations (jobs) of the same actor will execute in the same virtual processor, and, hence, on the same core. For dynamic applications it is the scheduling policy of the dispatcher that decides which virtual processor will be used. One option is to use the first available virtual processor, i.e., different invocations of the same actor are not bound to a particular virtual processor.

In addition to being responsible for the execution of CAL actors the runtime should also be able to execute external actors written in other languages, that either are linked in from separate libraries or loaded at run-time. One example of this type of actors are the *system actors*, whose purpose is to provide a means for communication between the CAL application and the resource manager and the operating system. Using these actors the applications can, e.g., announce their resource requirements to the resource manager and receive the information about the allocated reservations for them.

Application Adaptation

Applications executing on the ACTORS platform, whether they are CAL applications or general applications, can provide different amount of support for resource and quality adaptation. In the normal case we will assume that an application supports several quality levels consuming different amount of resources at the different levels. This type of application will be referred to as an *adaptive application.* An adaptive application informs the resource manager about its possible quality levels and the resource manager in return informs the application about at which level it should execute. Different service levels are achieved by, e.g., changing operating mode, assigning new values to some application-specific execution-related parameters, or changing the application's execution rate. During execution the application estimates its perceived quality and informs the resource manager if the allotted resources are sufficient or not. It is also possible for an adaptive application to contain an internal feedback loop that controls the amount of resources used and, hence, the quality obtained. The service level decided by the resource manager can in this case be viewed as a setpoint signal for the internal controller.

A *non-adaptive application* on the contrary has no means that it can use to execute at different quality levels and thereby consuming different amounts of resources. A non-adaptive application could either be ACTORS-aware or ACTORS-unaware. An ACTORS-aware application will inform the resource manager that it only supports a single quality/resource level. An ACTORS-unaware application will not provide any quality/resource information to the resource manager. In this case the resource manager will simply have to provide some initial guessed amount of resources to the application and then rely on the feedback for adjusting this to a suitable level.

The Resource Manager

The resource manager is the central component which has comprehensive knowledge about the system and which communicates with all other components. This means the resource manager

- knows which applications that should execute.
- knows which quality levels the applications provide and their resource demands.
- knows at which quality level the applications are currently running.
- knows how important the applications for the overall quality of the system are.
- knows the amount of resources (e.g. processing capacity) the current system provides.
- will have a way to "translate" the abstract, i.e. portable resource demands of the applications to actual resources of the current system.
- knows the current resource usage of the individual applications, reported by the operating system.

All applications running in the system have to register at the resource manager, so that the resource manager knows about them. The resource manager queries the applications for their resource requirements and it monitors their actual resource consumption. Based on this information it decides using some optimization/control logic how to distribute the reservations so that an optimal overall system performance is achieved.

In order not to overload the resource manager and to provide strong encapsulation and modularity, the interface between the applications and the resource manager will be a "slow" interface, only major changes will be communicated there. The resource manager will only react to qualitative changes, small quantitative changes will be handled locally in the applications.

There are two options how the resource manager can be implemented. The resource manager can be a daemon running in user space, talking on one side with the applications and on the other side with the resource reservation implementation in the kernel. This will make development of the resource manager itself and the control and optimization logic easier,and more convenient development tools are available. It will also be possible to support different resource reservation implementations in the kernel. On the other hand there will be some communication overhead.

The other option is to integrate the resource manager directly in the kernel. This will mean less overhead and in some points the communication with the applications will be easier, e.g. they would not have to explicitly register, since the kernel knows about them anyway. But it will make developing the resource manager harder and it will be harder to support multiple implementations in the kernel, as e.g. CFS and partitioned EDF.

The Operating System

A requirement for the operating system for the use in ACTORS is that it provides support for resource reservations, at least for the resource CPU time. In more detail this means:

- resource reservation support for CPU time, i.e. limit CPU usage per processes
- support for defining the time granularity for the reservations
- multicore support
- support for the ARM architecture
- per reservation reporting of resource usage to the userspace

The ACTORS resource manager will use these functions to monitor the resource usage of the applications and distribute the resources among them, so that the overall quality of service of the system is maximized.

The operating system used for ACTORS will be Linux. The Linux kernel is a UNIX-like general purpose operating system kernel, optimized for throughput and average performance. The most current version is 2.6.28, released December 24th, 2008. Since a few years there are multiple projects working on improving the real-time characteristics of Linux. One or more of these approaches will be used in ACTORS. The following will give a short overview over the approaches which are related to CPU resource reservations. None of the approaches right now fulfills all requirements ACTORS has, so there is no clear best candidate. Due to this an interface between the resource manager and the resource reservation implementation is planned, which abstracts the differences away. This way the resource manager can talk to the same interface, no matter which implementation is actually used.

Linux Control Groups

Since version 2.6.25, released April 17th, 2008, the Linux kernel comes with an option to enable the so called Control Groups (cgroups). *"Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour."* (from linux/Documentation/cgroups/cgroups.txt) Different parts of the Linux kernel can then support cgroups to control the usage of a certain resource by these groups of processes [4]. In principle, this enables to control the usage of many kinds of resources by processes. Currently, the cgroups interface is used only to control the usage of processor and memory, but in the very next future it may be used to control also I/O and networking.

Since version 2.6.24 Linux uses the Completely Fair Scheduler (CFS). CFS has been extended to support Control Groups, which means it is able to restrict the CPU usage of such control groups. This is an interesting feature for ACTORS, since it makes it possible to restrict groups of processes to some percentage of the available CPU time. This works quite, but not completely exact, i.e. the given percentage

is met by a few percents more or less. A downside of the current implementation is that only one global period is available, i.e. the percentage of CPU time is for all processes realized over the same period. This is not ideal, since different applications can require different periods. There is basic support for multicore systems, but this is not yet mature right now. An advantage of the control groups is that it is part of the mainline kernel and it is actively being worked on, so it may become even more useful for ACTORS during the project.

Aquosa

AQuoSA (Adaptive Quality of Service Architecture) is an open architecture for supporting adaptive Quality of Service functionality in the Linux kernel. It features a flexible, portable, lightweight and open architecture for supporting QoS related services on the top of Linux. The architecture is well founded on formal scheduling analysis and control theoretical results. A key feature of AQuoSA is the resource reservation layer that is capable of dynamically adapting the CPU allocation for QoS aware applications based on their run-time requirements. In order to provide such functionality, AQuoSA embeds a kernel-level CPU scheduler implementing the CBS resource reservation mechanism for the CPU, which gives the ability to the Linux kernel to realize (partially) temporal isolation among the tasks running within the system [5]. Aquosa also supports adapting the resource reservations to the actual CPU requirement by using an integrated feedback control system, which monitors the current resource consumption and adjusts the reservations accordingly.

AQuoSA is implemented as a patch to the Linux kernel, version 2.6.22 and supports currently the x86 architecture. The ARM architecture and multicore systems are not supported in the latest release from January 2008. AQuoSA is developed as part of the FRESCOR project [6] funded in part by the European Union's Sixth Framework Programme.

Litmus

Litmus (Linux Testbed for Multiprocessor Scheduling in Real-Time Systems) is also an extension to the Linux kernel [7]. It focuses on multicore systems and synchronization. For this it implements several multicore scheduling algorithms, e.g. Global EDF, Partitioned EDF and Pfair. Recently it has been extended to support Adaptive Global EDF (AGEDF) using feedback predictors to predict the execution times of jobs and switch between different service levels of these tasks [8].

Litmus is also implemented as a patch to the Linux kernel, version 2.6.24 and supports currently the x86 and Sparc architectures. The CBS resource reservation mechanism is currently not implemented in Litmus.

Partitioned EDF

A partitioned EDF-based scheduler with support for CBS-like CPU-time reservations is currently being implemented in ACTORS. A new scheduling class, `SCHED_EDF` is introduced. Tasks executing under this class will be executed before any other Linux tasks. This combined scheduler and reservation system will be the main target for the ACTORS architecture.

More details about `SCHED_EDF` and reservation based scheduling can be found in deliverable D4b "RBS Specification".

Architecture Diagrams

An overview over the architecture can be seen in 2.3, it shows the components and interfaces an ACTORS system consists of.

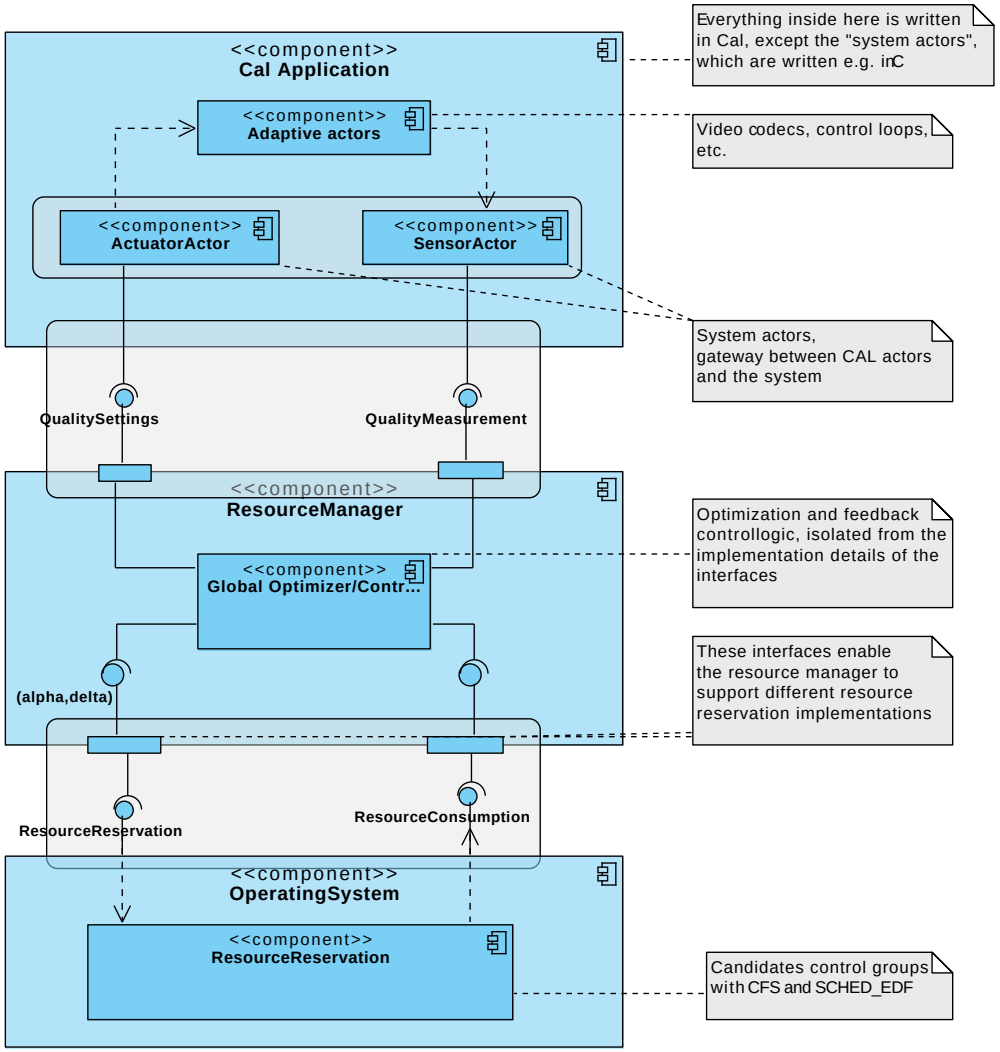

Figure 2.3: ACTORS architecture overview.

An alternative view is provided in Fig. 2.4. Here a snapshot of one concrete ACTORS system at runtime is shown, i.e. it contains instances of the components from 2.3. In this figure it is assumed that partitioned EDF scheduling is used together with bandwidth servers based on, e.g., CBS. The figure contains one static and one dynamic CAL application.

2.2 Feedback Mechanisms

The ACTORS resource management system contains at least three different types of feedback mechanisms:

- Global resource distribution

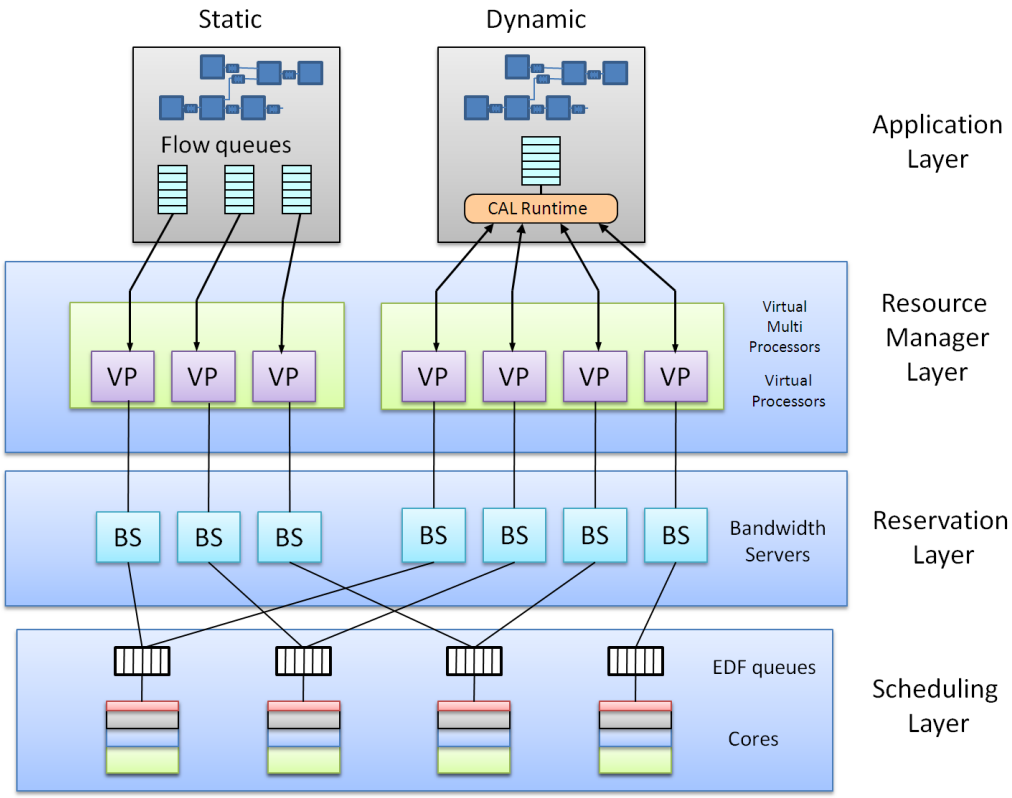

Figure 2.4: ACTORS architecture layers.

- Reservation feedback
- Resource feedback

The global resource distribution can be seen as a combined optimization and feedback control problem. Here the resource levels for the different applications are decided based on the quality tables provided and the optimization objective. The feedback appears in the form of information from the applications that they have modified their execution characteristics or that the current level of resources allocated is insufficient, or that a new application has arrived. The control loop is typically executed fairly seldom and in an event-driven fashion.

The reservation feedback loops are used to dynamically adapt the size of the reservations for the applications. Based on the how much of the allocated resources that really are used the size of the reservations are adjusted. Each reservation has its own feedback loop which is executed either at the period of the reservation or at some multiple of this period. In the latter case the measured resource consumption error has to be averaged. Since a reservation is determined by two parameters, the bandwidth α and the delay Δ, the possibility to use two feedback loops for each reservation will be investigated.

The resource feedback loops, finally, are optionally used in adaptive applications in order to ensure that an application really consumes the amount of resources that it has promised in its contract with the resource manager. The reason for having this type of feedback is the assumption that it is in general better for an application to, e.g., reduce its resource consumption itself in a controlled way, then to let this reduction be performed by the underlying reservation mechanism.

2.3 Interfaces of the Resource Manager

Interface between Applications and Resource Management

In an ACTORS-device multiple applications will be running. The whole system should behave adaptively, e.g. when the battery power becomes low it should still be able to offer some services, maybe with lower quality; if the user runs multiple applications at once, like displaying an animated webpage and at the same time doing a video call, which requires video encoding and decoding at the same time, the quality of the applications should degrade in a controllable and predictable way. To achieve this goal the applications themselves must be able to run with different CPU requirements and producing for the respective CPU availability optimal results. For that purpose ACTORS-aware applications will specify their resource requirements, so the resource manager can distribute resources according to the actual applications demands instead of just distributing them in an ad-hoc way. The specification of these demands should be done in a platform-independent, portable way. This would have the advantage that the requirements would still be valid if the application is ported to another system, if the processor is upgraded, etc. Resource requirements can potentially be a tuple of multiple required resources, e.g. processing power, latency, memory, harddisk throughput etc. In ACTORS we will focus on managing the CPU, i.e. processing power and latency.

When an ACTORS-aware application starts, it will register with the global ACTORS resource manager. The application will report its supported quality levels together with the required resources for each level. Not only discrete levels will be supported, but also continuous mappings from available resources to achievable quality. This information alone is not enough to find an optimal resource distribution. The issue here is how optimal is defined. Not every application will be of the same importance for the overall system quality level. So in addition to the resource requirements the developer must be able to specify relative priorities or importances for the various applications. With all this information the resource manager will decide how to distribute the resources, adjust the reservations in the operating system accordingly and notify the applications. To enable the resource manager to evaluate the current distribution of resources the applications will report their current level of "satisfaction" with the current resources back to the resource manager. This will be fed into the control- and optimization logic of the resource manager, which will then derive the new distribution of resources.

Interface between Resource Management and Operating System

The ACTORS-aware applications will not directly talk to the operating system to adjust their resource reservations, this will all be done by the resource manager. Currently there is not yet a mature resource reservation mechanism in the Linux kernel. As of 2.6.25 there is the Completely Fair Scheduler and a first version of Control Groups, which together make it possible to assign shares of the CPU time to groups of processes. But this is still very volatile, so we do not want to bind the ACTORS resource manager to just that one implementation. Instead it should be possible to support different resource reservation implementations, including e.g. an Aquosa-based one and a Litmus-based one. To do this, there will be an interface between the resource manager and the operating system implemented in the resource manager, which abstracts these differences away.

The resource manager will not only adjust reservations for the applications, it will also query the operating system for the currently consumed resources by the

applications. There may be different ways how the different implementations in the kernel report resource usage. If possible these differences will also be covered by the interface and translated to a common representation.

The need of an common interface for different mechanisms that are implemented on top of possibly different operating systems, imposes to extract the minimal number of features from the mechanisms used to implement the reservations.

The first key feature that is present in all the reservation interfaces is the *bandwidth*. The bandwidth measures roughly the amount of resource that is assigned to the demanding application. In simple periodic servers, the bandwidth is equal to the ratio between the allocated budget and the period of the server itself.

Indeed the bandwidth captures the most significant feature of a reservation. However the resource allocation of two reservations with the same bandwidth can be significantly different. Suppose that that a reservation allocates the processor for one millisecond every 10 and another one allocates the processor for one second every 10 seconds. Both the reservations have the same bandwidth that is the 10% of the available CPU. However the first reservation is more *responsive* in the sense that it can replenish the exhausted budget more frequently. As a consequence we expect that an application is more reactive if allocated on the first reservation. It is then desirable to add a notion of *time granularity* in the interface between the Resource Manager and the Operating System.

One intuitive measure of the granularity is the period of reservations. However there may be allocation mechanisms that are not periodic, such as Pfair [9]. In these cases it is not clear what is the value of the period. Hence we choose to measure the time granularity by the *delay*, that is the longest amount of time that the application may need to wait for being assigned some resource.

The abstraction by bandwidth and delay is quite common in many other domains, such as communication networks. This resource abstraction is called (α, Δ) server model [10], where α denotes the bandwidth and Δ the delay. Another name for this type of server model is a latency-rate server.

Given any reservation, below we describe how to extract the bandwidth α and the delay Δ from it. First we define a time partition as follows.

Definition 1 *A partition is a measurable function $\pi : \mathbb{R} \to \{0, 1\}$, with the interpretation that the resource is allocated to the application over $A = \{t \in \mathbb{R} : \pi(t) = 1\}$ and it is not allocated to the application over $\mathbb{R} \setminus A$.*

For example a static allocation mechanism pre-computes the partitions off-line, and, at run-time, a dispatch mechanism will make use of a simple table to allocate the resource. On the other hand, an on-line resource allocation mechanism uses some rule for dynamically allocating the resource. Therefore, an on-line algorithm may produce different partitions every time it is executed, depending on the arrival times and execution times of the application tasks. Moreover, these partitions are not necessarily periodic.

For a given partition, we define the minimum amount of time that is available to the application in every interval of length t.

Definition 2 *Given a partition $\pi(t)$, we define the supply function $Z_\pi(t)$ as the* minimum amount of time *provided by the partition in every time interval of length $t \geq 0$, that is*

$$Z_\pi(t) = \min_{t_0 \geq 0} \int_{t_0}^{t_0+t} \pi(x)\, \mathrm{d}x. \tag{2.1}$$

As an example, consider an off-line algorithm that produces a periodic partition $\pi(t)$ with period 8, which allocates the intervals $[1, 4]$ and $[6, 7]$ to the application.

The corresponding function $Z_\pi(t)$ is plotted in Figure 2.5. Note that the worst-case interval starts at time 4.

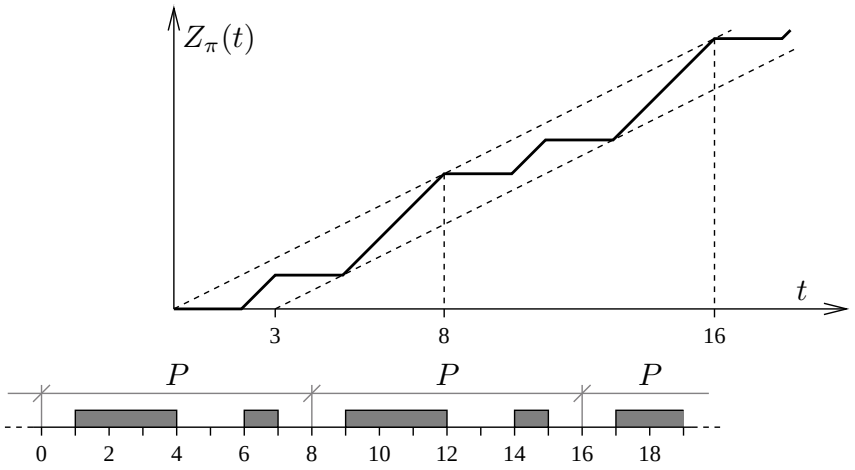

Figure 2.5: An example of $Z_\pi(t)$.

A partition π depends on the particular rules used to allocate the resource and on the events occur ed on line so it is not possible to compute the function $Z_\pi(t)$ in the design phase, because we don't know what π will experience. To generalize the supply function to the server mechanism, we introduce the following definitions.

Definition 3 *Given a reservation* $\mathbf{S}$*, we define* $\mathsf{legal}(\mathbf{S})$ *as the set of partitions* π *that can be generated by the reservation* $\mathbf{S}$*.*

Definition 4 *Given a server* $\mathbf{S}$*, its supply function* $Z_\mathbf{S}(t)$ *is the* minimum amount of time *provided by the server* $\mathbf{S}$ *in every time interval of length* $t \geq 0$*,*

$$Z_\mathbf{S}(t) = \min_{\pi \in \mathsf{legal}(\mathbf{S})} Z_\pi(t). \tag{2.2}$$

As the supply function of a reservation $\mathbf{S}$ is computed, it is possible to find its bandwidth α and delay Δ. α is the average slope of $Z_\mathbf{S}(t)$, formally defined as:

$$\alpha = \lim_{t \to \infty} \frac{Z_\mathbf{S}(t)}{t}. \tag{2.3}$$

The computation of the value of Δ requires some more efforts. Informally speaking, once we have computed α, the delay Δ is the minimum amount we must right-shift the line αt to be completely below $Z_\mathbf{S}(t)$. Formally:

$$\Delta = \max\{d \leq 0 : \exists t \geq 0 \quad Z_\mathbf{S}(t) \leq \alpha(t-d)\}. \tag{2.4}$$

In the example of supply function $Z(t)$ shown in Figure 2.5 we have $\alpha = \frac{1}{2}$ and $\Delta = 3$.

An (α, Δ) server can be mapped to a CBS with period P and budget Q as follows.

$$\alpha = \frac{Q}{P}$$
$$\Delta = 2(P - Q)$$

The value of Δ corresponds to the worst case delay between two invocations of the server in two successive periods.

Chapter 3

The Application Interface of the Resource Manager

3.1 Overview

The interface between the resource manager and the ACTORS-aware applications serves multiple purposes. In general, it enables communication between the resource manager and the applications. The resource manager queries applications for the service levels they support and their accompanying resource requirements. It will also notify the applications about its decisions regarding the resource reservations. And last but not least, the interface will also be used to notify the resource manager about the level of satisfaction the applications are achieving currently. Two types of ACTORS-aware applications will be supported: applications with a number of discrete service levels, and applications with a continuous mapping of resource usage to achieved quality.

The resource manager will not react to each and every minor change in resource availability or resource demand, these have to be handled locally in the applications. Instead it will only make major decisions, which will be necessary, when applications start or quit, when the required processing power for a video changes e.g. from a still scene at night to a fast moving sports scene etc.

The interface should be hardware-independent, so the parameters transported there should be independent e.g. from the current CPU.

3.2 Applications with Discrete Quality Levels

An application which provides discrete service levels is for instance a video-encoder. It could provide full quality, i.e. high resolution, full frame-rate; medium quality, e.g. with lower resolution but still full frame-rate and low quality with lower resolution and also lower frame-rate. Different applications can provide a different number of levels, requiring different sets of resources.

To be useful for the resource manager, these quality levels need to have an indicator to show how "good" they are, i.e. how much quality they provide. For each application there will be one defined measure how to determine the level of quality. For a video application this could be the PSNR, or a value computed in some way from a combination of resolution and frame rate. This measure will be used internally by the applications and will be mapped to an integer quality number. So the outside world, including the resource manager, does not have to know which actual measurable value the quality number represents.

Each application has a set of demands on resources. This can include:

Index	Quality	Resource 1	Resource 2	...	Resource N
0	Q0	R10	R20	...	Rn0
1	Q1	R11	R21	...	Rn1
...	...	...	...	...	...
M	Qm	R1m	R2m	...	Rnm

Table 3.1: Set of quality levels

Application	Importance
A1	I1
A2	I2
...	...
An	In

Table 3.2: Assignment of importances to applications

- CPU time
- CPU time granularity
- Disk bandwidth
- Network bandwidth
- Available energy
- and more

For each quality level the demands on the set of required resources will be specified. These specifications should not be bound to some platform, i.e. not to some specific hardware, some specific resource reservation implementation, etc.

This will form a table-like representation, which each application publishes, as shown in Table 3.1. With this information the resource manager knows about the quality levels the applications can work at and their properties. Based on this information the resource manager can optimize the distribution of resources so that an optimal overall system performance is achieved.

Application Importance/User Preferences

Optimal overall system performance is a vague term. Does a system perform best if application A performs at the highest level but applications B and C only creep along ? Or does it perform best if all three applications are able to work at a medium quality level ? Is it better if the system is able to play a movie at its full resolution and frame rate, but after that the battery power is used up, or is it better if the movie is played at a mediocre quality but therefore there is enough battery power left for one more day of use ?

The software itself cannot decide this, some human must help and specify what behaviour is preferred. This is done by assigning integer importance values to the applications, as can be seen in Table 3.2. These will be taken into account by the resource manager when it optimizes the resource distribution. It is up to the resource manager how to handle different importances.

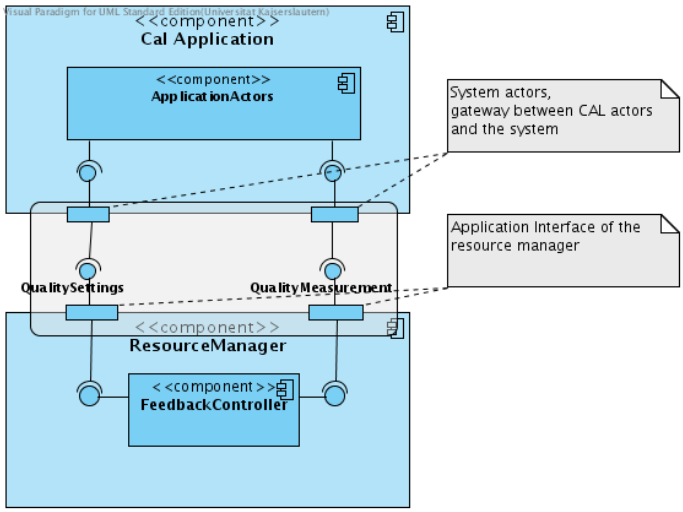

Figure 3.1: Interface between resource manager and applications.

Additionally there can be different objectives by which the resource manager should optimize the resource distribution. As mentioned in the beginning, user A might prefer highest possible quality, while user B might put emphasis on a long battery life. To support this, the resource manager has a global property which determines the current optimization objective.

Reporting Achieved Application Quality

As mentioned before, applications are able to work at different quality levels. Each quality level has an index and an external visible quality number which translates into an internally used actual measure. Applications will monitor themselves for the quality measure they are achieving and determine to which quality level the current output belongs. Whenever this currently achieved quality level changes, the new level will be reported by its index to the resource manager over the reporting interface of the resource manager, denoted as "QualityMeasurement" in Figure 3.1.

Setting Up the Quality Levels of Applications

All ACTORS-aware applications register at the resource manager. The resource manager then queries the quality levels of the applications, so that it has a global view over the participating applications and their resource requirements. Additionally the resource manager knows the importance values assigned to the applications and also the current optimization objective according to the user preferences.

Using all this information the resource manager finds an optimal distribution of resources among the applications. This means it determines the respective quality level indices for the applications and tells the applications to work at these levels. All this is done over the interface denoted as "QualitySettings" in Figure 3.1. At the same time it translates the abstract resource requirements from the applications into concrete parameters for the available reservation system services and adjusts them accordingly.

After the quality levels for the applications have been set, these applications will report back the quality they are achieving currently. If everything goes as

expected, they will all be working at the desired level. But it can also happen that an application achieves a higher or lower level than it should according to the settings. This could happen e.g. if a video switches from a low contrast low movement scene to a high contrast high movement scene, then the processing requirement would suddenly increase. Over its reporting interface the resource manager will be notified about such changes. This forms a control loop.

Representation of Resource Demand Values

The resource demands depend on the individual applications. There are resource demands which are dependent on the platform the application runs on and resource demands which are not. E.g. a video decoder which also writes the decoded stream to a permanent storage has among others the following demands:

- CPU time to decode the video
- the period in which new frames have to be decoded
- disk bandwidth to write the decoded video to the storage device

From these three examples CPU time needed for decoding depends very much on the platform, while the other two are independent from the platform. These are instead properties inherent to the application.

CPU Time

Every application needs some execution time to work successfully on a given hardware. There are several issues here: the execution time can vary, so in some applications assuming the worst case execution time can be very pessimistic and lead to low actual system utilization. Encoded video streams are a good example for this, here the effort required to decode the frames depends heavily on the video contents, which vary over time. There are different options how execution time requirements can be specified.

- An obvious way to express the CPU time requirements is to specify the percentage of the full processor power required for the application. But this has the also obvious disadvantage, that it is completely CPU specific.
- Another option would be to specify a period and the execution time per period. This also has the disadvantage that it is CPU specific.
- The execution time required depends basically on the logic encoded in the source code, which does not change when executed on a different platform. Using e.g. the number of cycles required to solve a particular problem is still not possible, since this can also depend vastly on the actual hardware. E.g. decoding a video stream on a RISC CPU will map to a large number of cycles, while decoding the same video stream e.g. on a CPU with additional multimedia (SIMD) instructions will lead to a much smaller number of execution cycles.
- A good option would be if the application would contain information about the high level operations which will be executed [11]. This information could then be mapped to the actual hardware. Unfortunately this information is usually not existing.

- Platform dependencies would be avoided if only a small set of discrete levels, like “High”, “Medium”, “Low” and “Minimal” would be specified. While this would be very hardware independent, it also would not contain a lot of information. It would then be up to the feedback controllers to figure out the details. While this would work for ordering the quality levels of one application by their need for processing power, this would give no information about the relative requirements of multiple applications. If e.g. an application offers 3 different quality levels, but in general has already low needs for computation time, this cannot be expressed.

To solve these issues, we propose to use two separate measures for specifying the execution time requirements. On one hand it must be possible to express the requirements of different applications relative to each other, and on the other hand it must also be possible to express the differences in resource requirements of the different quality levels within each application.

Specifying the CPU time requirements of applications relative to each other exactly is hard, due to the given reasons. Instead, as above a small set of discrete categories is introduced, and each application is assigned to one of those categories. E.g. a control application would be assigned to the "Low CPU time requirement“ category, while a video encoder would be assigned to the ”High CPU time requirement“ category. The resource manager can then use these categories as a very rough hint how to set up the initial reservations, i.e. when there is no feedback information available yet. So a category ”Low“ might result in an initial reservation of 10 %, a category ”Medium“ might result in an initial reservation of 25 %, and a category ”High“ might result in a reservation of 50 %. These values are pure guesses that the resource manager can make, but they will provide at least some reasonable starting point.

The CPU time requirements of the different quality levels within one application relative to the highest quality level of that application should be quite exactly known, since these levels have been implemented for the purpose to use less CPU time. So here integer values can be used, which have to be interpreted relative to each other within one application. E.g. for a video decoder it is plausible that a mode with half the frame rate and half the resolution should need around 25 % of the full quality mode, also without knowing how much that maximum actually is. A similar approach is used in AGEDF [8], here ”weight translation functions“ are used to translate quality levels to resource demands.

CPU Time Granularity

In addition to the amount of CPU time an application needs, it also needs to specify at which granularity this time is needed. This granularity is given by the application and independent from the platform the application runs on. This value is known and it should be easily possible to specify it as concrete value, e.g. in seconds. If it is a control application, the sampling rate depends on the plant. If it is a video decoder, the frame rate just depends on the media. For both mentioned examples it is necessary to be able to specify the granularity, otherwise it might happen that although the application received the amount it required, e.g. 40%, but distributed in such a way that it did not receive any CPU time in the first minute and then more than it needed after that. In many cases such a behaviour would be unacceptable. By specifying the granularity the control application can say that it needs the CPU time e.g. every 10 ms, and for video applications in many cases it will be 40 ms, which corresponds to a frame rate of 25 fps.

For CAL applications, where there are actors exchanging tokens, it may be also possible that more information can be extracted automatically from the code by examining token rates.

Disk Bandwidth

This is quite similar to the CPU time requirement. Also an amount of disk bandwidth is necessary as well as the time granularity, for the same reasons. But there is a difference, the required disk bandwidth will usually be known and it will be hardware independent. If an application stores e.g. a video stream to disk, the required bandwidth depends only on the amount of data in the stream, so here also the concrete value can be used, e.g. expressed in kB/s.

Power Consumption

This is similar to the required processing power. It completely depends on the actual hardware. Here it may also be possible to declare the power consumption relative to the one with the highest consumption.

3.3 Applications with Continuous Quality Mapping

An example of an application which supports a continuous mapping from resource usage to achieved quality is a feedback controller. In feedback control the most natural quality candidate is the control performance. In general the definition of control performance is highly application dependent. Most control applications consist of a number of individual control loops whose aggregated performance decides the overall performance. The relationship between the overall performance and the performance of the individual loops is furthermore in most cases very complex. For example, the overall performance of an industrial robot is mainly concerned with the precision of the hand actuator. However, this depends on the performance of the controllers for all the robot joints plus, possibly, additional force and vision feedback loops. Another example is a flight control system where the dynamic behaviour of the airplane is dependent on the control loops involving all the individual actuators (ailerons, rudders, engine thrust, etc).

Also when one considers a single control loop the definition of control performance is not straightforward. One possibility is to define the performance in the time domain based on the response to a step change in the setpoint value of the control loop. Here, parameters such as rise time, overshoot, maximum control signal and settling time all are related to the control performance. Another possibility is to instead consider the frequency response of the control loop. In ACTORS, however, the performance will be measured in terms of the value of a quadratic cost function defined as

$$J_c = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{bmatrix} x(t) \\ u(t) \end{bmatrix}^T Q \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt$$

where Q is a positive semi-definite matrix, $x(t)$ is the state variable vector, and $u(t)$ is the control signal vector.

An advantage with the quadratic cost function definition of control performance is that it for a large class of systems is possible to calculate the expected value of the cost function analytically. Furthermore, this can be done as a function of the amount of CPU resources used by the control loop. Using, e.g., the Matlab toolbox Jitterbug [12] developed by the Control Department at University of Lund (ULUND) it is possible to calculate how, e.g., the control performance of a control

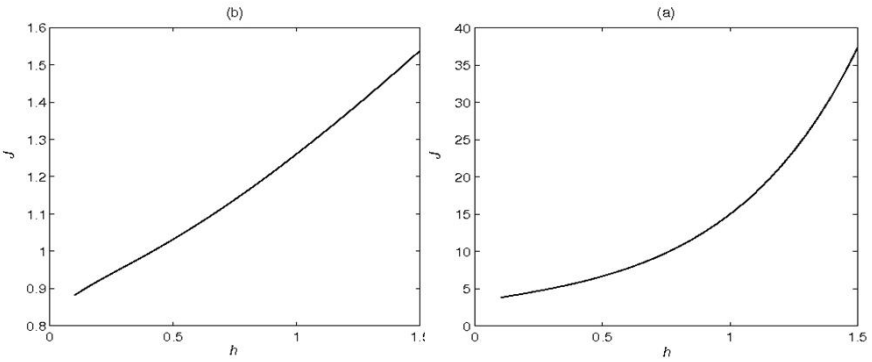

Figure 3.2: Typical appearance of cost function graphs.

loop depends on the sampling period or on the input-output latency of the control loop. It is also possible to define these as arbitrary statistical distribution functions rather than simply as constant values.

In ACTORS it is the possibility to evaluate the quadratic cost functions for a wide range of constant sampling periods that is most relevant. Evaluation of a large number of process types and different controller structures have shown that the relationship between sampling period and cost function values in most can be described by a linear or quadratic relationship, i.e., the larger the sampling period the larger the value of the cost function (and, hence, the worse the performance) [13], see Fig.3.2. A value of infinity for the cost function indicates that the closed loop system is unstable. The cost function can be evaluated analytically when the plant under control and the controller are linear, and all stochastic variables are independent. In other cases, the function can still be evaluated through simulation using, e.g., the TrueTime toolbox [14].

The sampling period of a control loop corresponds to the amount of CPU resources consumed by the application. If the period equals the worst case execution time the application will consume 100% of the CPU resources on a single core. The sampling rate in combination with the worst-case execution time together influence the CPU time and the CPU time granularity. The CPU time granularity typically corresponds to the sampling period whereas the CPU time required should be large enough to allow the controller code to be executed within a single sample.

Assuming that there is a linear or quadratic relationship between the sampling rate and the cost (the inverse of the performance) it is possible to either use these functions when defining the relationship between the corresponding resource requirements and the quality levels, or one can discretize them into integer values. If discretization is used then two things must be kept in my mind. First, the discretization applies both to the performance and to the quality levels. Second, there must be sufficiently many discrete levels in order to fit the continuous curve with reasonable accuracy.

3.4 Translating Application Demands to Resource Reservation Parameters

As discussed before, applications will specify their resource requirements in a system independent way, so they are not tied to specific hardware. In the end these abstract parameters must be translated to specific parameters for the current resource reservation implementation on the current hardware. This will be done using the

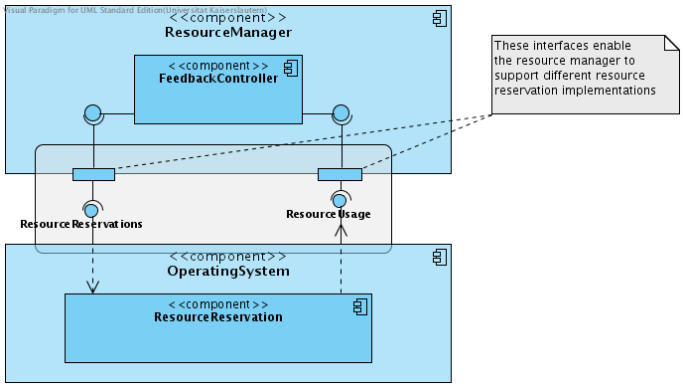

Figure 3.3: The interface to the resource reservation implementation.

optimization and feedback control logic in the resource manager. The feedback control system will work on one side with the abstract quality levels of the applications and on the other side with the abstract parameter settings for the resource reservation. In the interface to the resource reservation these parameters will then be translated into parameters for the actual resource reservation implementation, as depicted in Figure 3.3

Chapter 4

Example Walkthrough

4.1 Control Application

A control application in which the the performance depends linearly on the adjustable sampling rate is implemented on an ACTORS platform. For the sake of simplicity a discretized version of the continuous-valued performance functions is used in the example.

- The possible quality levels are

Service level	Quality	Processing demand	CPU time granularity
1	1	33	20 ms
2	2	40	50 ms
3	3	50	40 ms
4	4	66	30 ms
5	5	100	20 ms

- The overall CPU time requirement category: Low
- The application starts and registers itself at the resource manager together with the service table above.
- Since there is enough free resources the resource manager decides that the application may run at service level 5.
- The resource manager maps the processing demand and the latency requirements for this quality level to parameters for the current hardware and the current reservation mechanism used, e.g., α and Δ in a latency-rate server.
- The resource manager informs the application about the service level it should execute at.
- The application uses this information to change the sampling rate for the controller to the corresponding value and to recalculate the controller parameters accordingly.
- After a while one of the already hosted applications needs to change operating mode. This forces the resource manager to redistribute the resources. Now the control application may only operate at quality level 2. The resource manager informs the controller about the new reservation parameters.
- The application modifies the sampling rate and the controller parameters according to the new service level.

4.2 Multimedia Application: MPEG2 decoder with QAFS

In this example we will talk about an MPEG 2 decoder with Quality Aware Frame Skipping (QAFS) [15]. The basic idea of QAFS is that if not enough CPU time for decoding all frames is available, then some frames are skipped, so at least all frames which are actually decoded can be decoded in time. With QAFS skipping frames doesn't start with a random frame, but with the least "important" frames.

- We assume the following service levels:

Service Level	Quality	Processing demand	CPU time granularity	Description Description
0	100	100	40 ms	skip no frames
1	50	80	40 ms	skip 20 % of frames
2	10	50	80 ms	skip 50 % of frames

- Overall CPU time requirement category of the MPEG2 decoder: High
- The application starts and registers itself at the resource manager.
- The resource manager queries the application for its resource demands and gets the above table as response.
- The resource manager uses some clever technique to map the "High" processing demand and the latency requirements to parameters for the current hardware.
- The resource manager allocates a big part of the CPU for the decoder, since it has "High" processing demands.
- E.g. when another application starts and requires CPU, not enough CPU resources are left for the MPEG decoder.
- The MPEG decoder reports to the resource manager that it is not able to achieve a quality appropriate for level 0 anymore.
- The resource manager considers the importance values of the applications and comes up with a new distribution of resources.
- This could mean that the MPEG decoder now has to run at level 1, where it may skip 20 % of frames.
- Using a local controller the MPEG decoder tries to keep its CPU consumption appropriate.
- Beside telling the application to perform at level 1, the resource manager also adjusts the reservation in the operating system accordingly.
- Therefor it has to translate the portable values to concrete values.

Service level	CPU bandwidth	Server period
0	50 %	40 ms
1	40 %	40 ms
2	25 %	80 ms

- These concrete values then have to be translated to parameters for the resource reservation:

Service level	Reservation, (e.g. α/δ)
0	0.02 s / 0.04 s
1	0.016 s / 0.04 s
2	0.02 s / 0.08 s

Appendix A

Terminology

Actor		The basic abstraction in CAL. An entity that communicates asynchronously with other actors by consuming tokens from inports and producing tokens at outports. Computations are performed in a sequence of firings. In each firing the actor may consume tokens from input ports, may modify its internal state, and may produce tokens at output ports.
Action		Describes the computations that are done within a firing of the actor that the action is part of.
Compiled actor	CA	A set of instructions, resulting from CAL compilation, to be executed sequentially on a single processor. No parallelism is possible within a compiled actor. Corresponds to one or a set of actors. A compiled actor may have an associated WCET.
Data flow Graph	DG	Graph consisting of instances of actors connected by FIFOs. In most cases the DG is cyclic.
Task	T	An instance of a compiled actor with a unique set of ports and state. A task is a sequential piece of code that may be associated with a deadline and an execution time.
Job		A particular invocation of a task.
Job Precedence Graph	JP	A precedence relation between the jobs described by a directed acyclic graph (DAG). For static applications the DG can be converted to a JP.
Application		A set of threads or tasks which implement a functionality.
CAL application		An application written in CAL, consisting of connected actors.

ACTORS-aware application		Applications that cooperate with the resource manager. In most cases this will be CAL applications, but it is not a requirement.
Static Application	SA	An application that can be described by a JP. Static applications are activated with a minimum inter-arrival period and may have timing requirements, (e.g., deadline, throughput, or delay). The precedence graph of a static application is known a priori and can be analysed off line.
Dynamic Application	DA	An application that is described by a DG and cannot be translated into a JP.
Flow	F_i	A serialized (totally ordered) set of jobs that are sequentially executed on a single process.
Virtual Processor	VP	Uniprocessor reservation characterized by a bandwidth $\alpha \leq 1$ and a delay $\Delta \geq 0$.
Virtual multiprocessor	VMP	Multiprocessor reservation. When partitioned multiprocessor scheduling is used this corresponds to a set of VPs.
CAL Runtime		A software entity which sets up and executes CAL applications. Each dynamic CAL application has its own instance of the CAL runtime. It contains and actor activator and an actor dispatcher.
Actor Activator		A mechanism that that activates actors as input data become available and marks them as ready. Part of the CAT runtime.
Actor Dispatcher		A mechanism that selects an active actor for execution and executes it until completion.
Job scheduler		The operating system level algorithm that selects a job among a set of ready jobs and assigns it to a processor for execution, e.g. an EDF scheduler.
Process		An operating system process in user space, with its own address space, potentially containing multiple threads, etc.
Thread		A thread in its conventional meaning: a thread of execution without a separate address space. Multiple threads can be executed within one process.

Bibliography

[1] J. W. Jannek, "Tokens? what tokens ? - a gentle introduction to dataflow programming," 2007.

[2] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, pp. 24–35, Jan. 1987.

[3] E. Lee and T. Parks *Proceedings of the IEEE*, vol. 83, pp. 773–801, May 1995.

[4] J. Corbet, "Process containers," 2007.

[5] L. Abeni, *Resource Reservation in Dynamic Real-Time Systems*, pp. 123–167. 2004.

[6] "Frescor: Framework for real-time embedded systems based on contracts."

[7] B. B. Brandenburg, A. D. Block, J. M. Calandrino, U. M. Devi, H. Leontyev, and J. H. Anderson, "Litmus(rt) - a status report," in *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[8] A. D. Block, B. B. Brandenburg, J. Anderson, and S. Quint, "An adaptive framework for multiprocessor real-time systems," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[9] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, (Stockholm, Sweden), pp. 35–43, June 2000.

[10] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proceedings of the* 23^{rd} *IEEE Real-Time Systems Symposium*, (Austin (TX), U.S.A.), pp. 26–35, Dec. 2002.

[11] M. Mattavelli and S. Brunetton, *Implementing Real-time Video Decoding on Multimedia Processors by Complexity Prediction Techniques*. 1998.

[12] A. Cervin and B. Lincoln, "Jitterbug 1.1—Reference manual," Tech. Rep. ISRN LUTFD2/TFRT--7604--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, Jan. 2003.

[13] H. Zaben, "Utvärdering av latens och jitter för LQG och PID," Master's Thesis ISRN LUTFD2/TFRT--5807--SE, Department of Automatic Control, Lund University, Sweden, Dec. 2007.

[14] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén, "How does control timing affect performance?," *IEEE Control Systems Magazine*, vol. 23, pp. 16–30, June 2003.

[15] D. Isovic and G. Fohler, "Quality aware MPEG-2 stream adaptation in resource constrained systems," in *16th Euromicro Conference on Real-time Systems (ECRTS 04)*, (Catania, Sicily, Italy), 2004.