Adaptivity & Control of Resources in Embedded Systems

D4b

COOPERATION

RBS Specification

Responsible: **TU Kaiserslautern** (TUKL)

Karl-Erik Årzén (ULUND), **Enrico Bini** (SSSA), **Gerhard Fohler** (TUKL),
Alexander Neundorf (TUKL), **Claudio Scordino** (EVI)

Project Acronym: **ACTORS**
Project full title: **Adaptivity and Control of Resources in Embedded Systems**
Proposal/Contract no: **ICT-216586**
Project Document Number: **D4b 1.0**
Project Document Date: **2009-01-31**
Workpackage Contributing to the Project Document: **WP4**
Deliverable Type and Security: **R-PU**

Contents

Chapter 1

Introduction

Deliverable context

This deliverable is the result of task 4.2 "Matching the Reservation Schemes with the System State Abstraction for Control", which is part of workpackage 4 "Resource Reservation Framework". This is the final version of this deliverable, but at the current state of the project it can only present the intermediate state, on which further work in ACTORS will be based. Due to the iterative nature of development the plans presented here will be modified and improved during the project.

The task of the resource manager in ACTORS is to distribute available resources among the applications, this will be done by creating CPU reservations and assigning them to the respective applications. This deliverable presents the interface between the resource manager and the resource reservation provided by the operating system. The resource manager itself is presented in deliverable D3a "State Abstractions", which is right now available in an intermediate version. Its interface from the point of view of applications using it is presented in deliverable D1f "Interface Specification". The final versions of these three deliverables together will describe the resource manager from all aspects. We recommend to read first deliverable D3a, followed by D1f before reading this deliverable D4b. This way the reader will have a much better understanding of how all the components work together. The results of this deliverable are necessary for all tasks of workpackage 3, since they are all closely related to the resource manager, and for tasks 4.4 and 4.5. In these two tasks the resource reservation mechanism for ACTORS will be developed and implemented, which has to provide an interface as described here.

Overview

Fig. 1.1 shows the architecture diagram of the ACTORS system. As can be seen, the resource manager is in a central location, on one side it interfaces to the CAL applications, and the other side it interfaces with the resource reservation from the operating system. This deliverable focuses on the "lower" parts, i.e. the interfaces between resource manager and operating system. So this deliverable starts with chapter two at the bottom of the diagram, where approaches to resource reservation under Linux are presented. This included CFS and SCHED_EDF, which will be the resource reservation systems supported by the resource manager. Internally the resource manager will use the (α, Δ) server model, which is presented in chapter three. This is an abstract model for specifying both CPU bandwidth as well as delay requirements, which enables the resource manager to work independently from the specific interface details of the respective resource reservation system provided by the operating system, denoted as *ResourceReservation* in the

figure. Chapter four outlines what functionality the interface between the resource manager and the resource reservation system, denoted as *(alpha,delta)* in the figure, must support in order to be powerful enough to fullfill all needs of the resource manager. The chapter also describes how the current resource consumption will be reported by SCHED_EDF, this is done via the interface denoted as *ResourceConsumption* in Fig. 1.1. Now that the resource reservation mechanism, the theoretical server model to be used as well as requirements for the actual interface have been described, chapter five shows how the resource manager will use the available information items to determine parameters for the (α, Δ) servers, which will then have to be implemented e.g. by SCHED_EDF.

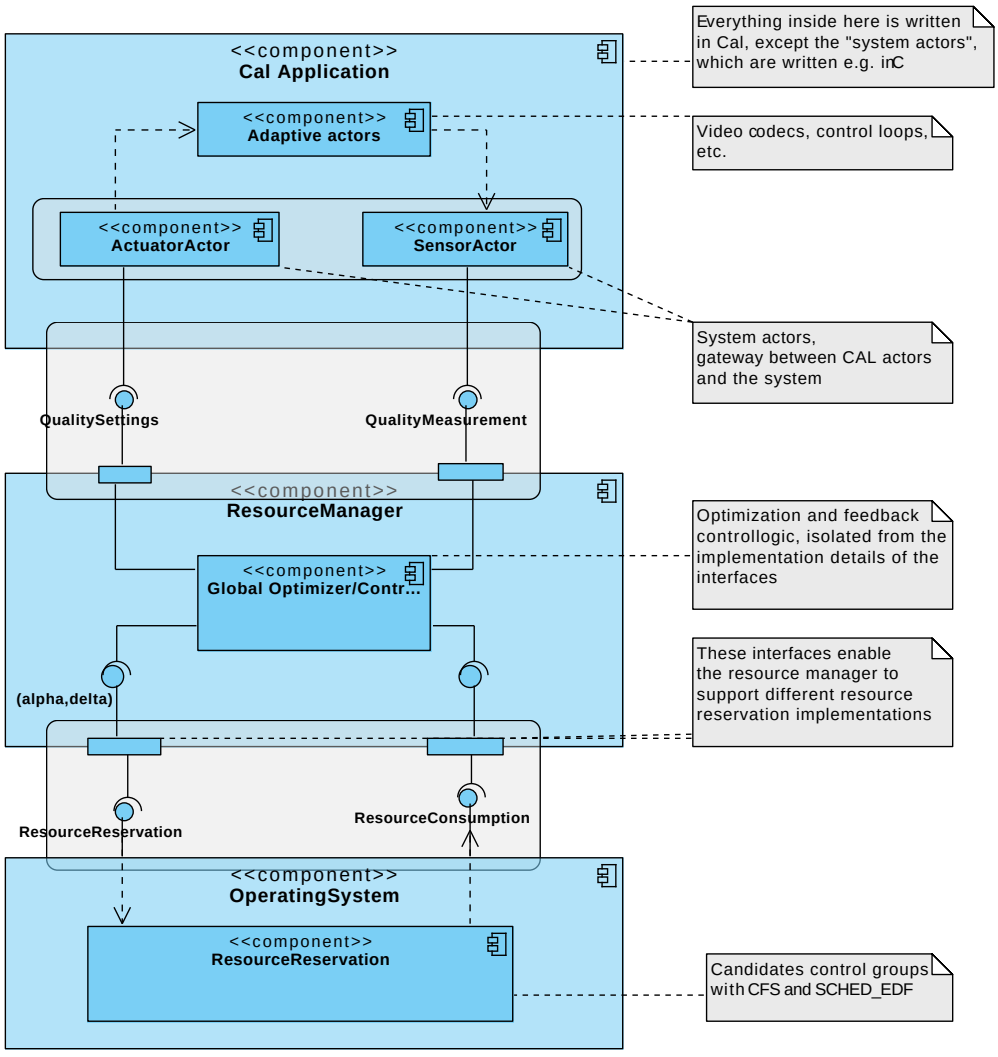

Figure 1.1: ACTORS architecture overview.

In the following the terms reservation and virtual processor will be used with the same meaning.

Chapter 2

Towards a new Resource Reservation scheduler for Linux

The process scheduler is the component of the kernel that selects which process (or processes, on multi-core platforms) to execute at any instant of time. The process scheduler (or simply the *scheduler*) is the subsystem of the kernel that divides the finite resource of processor time between all runnable processes on a system. It is an important part of the kernel (responsible for a good utilization of the processor, which is a shared resource) and the basis for multitasking (which gives the illusion of multiple processes running simultaneously even on single-core platforms).

In this chapter we will introduce the concept of Resource Reservations, and we will explain why a scheduler based on Resource Reservations is needed to have a deterministic system. Then, we will provide an overview of the past and current Linux schedulers which provide some kind of reservation mechanism. Finally, we will describe `SCHED_EDF`, the Resource Reservation mechanism that will be implemented and used in the Actors project.

2.1 The need for Resource Reservations

Description of the old O(1) scheduler

The old scheduler of Linux (called *O(1)* [1]) has been designed and implemented by Ingo Molnar. It is a fully preemptive algorithm, with good interactive performance and SMP scalability. Its main feature is the O(1) complexity: all computations completes in a constant time regardless of the number of running processes.

It provided three policies to schedule the processes:

- `SCHED_NORMAL`
- `SCHED_FIFO`
- `SCHED_RR`

The `SCHED_NORMAL` policy is a time-sharing policy for normal processes, with priorities mapped internally in a range between 100 and 140.

A slice of the processor time (called *"timeslice"*) is assigned to each runnable process, and decremented during process execution. When the timeslice of the running process expires, the scheduler replaces the process with another runnable process. In other words, the timeslice specifies how long a process can run before being preempted — i.e., when a process' timeslice reaches zero, the process is preempted and cannot run until all the other processes have exhausted their timeslices.

Setting the default timeslice is not a trivial task, of course: longer timeslices give poor interactive performance, while shorter timeslices create much overhead due to context switches. In the O(1) scheduler the timeslice computation is based on the static priority [1] called *"nice value"*. This is a value between -20 (i.e., highest priority) and +19 (i.e., lowest priority) that can be changed using the `nice()` system call. Timeslices are expressed as percentage of the `HZ` variable. This variable specifies the timer's *tick rate*: at boot time Linux programs the registers of the hardware timer to issue interrupts at this frequency.

`SCHED_FIFO` and `SCHED_RR` are policies for high-priority processes. In the Linux community, these policies are called *"real-time"* with a meaning very different from the common meaning in the real-time literature: processes scheduled with these policies have highest priority in the system, but no timing constraints (like deadlines) are associated with them. Internally, these processes have a priority mapped in a range from 0 to 99.

`SCHED_FIFO` is a simple First-In First-Out algorithm, without timeslices: the process runs until it explicitly yields the processor, and can be preempted only by higher priority processes.

`SCHED_RR` is similar to `SCHED_FIFO` with timeslices: among processes with the same priority, a process executes until it exhausts a predefined timeslice (i.e., Round-Robin). However, a process cannot be preempted by processes with lower priority (as in `SCHED_FIFO`).

The basic data structure of the O(1) scheduler is the `runqueue`, which contains the list of all runnable processes on a given processor (i.e., one runqueue per processor). Each runqueue has two priority arrays called `active` and `expired`, respectively. `active` contains all processes having timeslice left, while `expired` contains all processes that exhausted their timeslice. When a process' timeslice reaches zero, its timeslice is recalculated and the process is moved to the `expired` array. These priority arrays are the base for the O(1) complexity: when there are no more processes in the `active` array, the pointers to the two arrays are switched. This way, recalculating all the timeslices is equal to swapping two pointers (i.e., it can be done in a constant time, regardless of the number of running processes).

Issues with the O(1) scheduler

A scheduling algorithm has to satisfy very conflicting goals:

1. A fast response time (i.e., low latency for interactive processes become runnable)
2. A high throughput (best utilization of the processor by reducing the number of context switches)
3. No starvation: ready processes should not wait too long before obtaining the processor

Unix variants (Linux included) tend to explicitly favor I/O-bound and interactive processes. The O(1) scheduler had very good interactive performance, indeed, but it did not provide any kind of reservation between processes. Thus, developers were able to implement some tests which exploit the knowledge of the heuristics used by the scheduler to force it behaving differently than expected. Some examples of these tests are `fiftyp.c`, `fthud.c`, `fchew.c`, `fring-test.c` and `fmassive_intr.c` [2]. These tests showed that, without some kind of reservation algorithm, a process could affect the processor share given to other processes.

[1] Note that there is an error in [1] about the computation of timeslices.

The operating system should instead support scheduling policies providing *temporal protection* among the running processes. This means that the timely execution of a process should not be affected by the behaviour of the other processes running on the system. This way, if a process misbehaves, and tries to use all the resources of the system, it cannot starve the other processes. It is important to provide *temporal protection* among different processes, similarly to the way the Linux kernel provides memory protection.

Introduction to Resource Reservations

The Resource Reservation mechanism [3, 4] is an effective way for providing such temporal protection in General Purpose Operating Systems (GPOSs) like Linux. The basic idea behind the resource reservation technique is to *reserve* a share of the resource to each process — i.e., reserve the resource for a fraction of the time to the process. During its execution, the process is executed at an appropriate priority. However, if the process tries to execute for a longer time, then it is suspended and resumed later. This way, each process is constrained to not use more than its reserved share.

A general mechanism to enforce Resource Reservations in a GPOS is to create a set of *"servers"* or *"virtual processors"*. Each virtual processor is characterized by a *"budget"* Q_i and a *"period"* P_i and serves a specific process of the operating system. Such process is then allowed to execute for an amount of time equal to Q_i every period P_i.

The budget Q_i acts as a timeslice: it is decreased when the process belonging to the virtual processes executes. When the budget reaches zero, the priority of the virtual processor is decreased and the processor can be given to processes belonging to higher priority virtual processors (if any).

2.2 Related work

During the last years, several approaches and mechanisms have been proposed to add Resource Reservations to the Linux scheduler. This section provides an overview of the state of the art of Resource Reservation schedulers available for Linux. Eventually, none of the schedulers listed in this section was accepted in the mainstream kernel, and Linux users had to wait the CFS scheduler (see Section 2.3) before having some kind of reservation inside the standard kernel.

OCERA

A real-time scheduler based on Resource Reservations has been developed for Linux 2.4.18 within the OCERA (*"Open Components for Embedded Real-time Applications"*) European project, and it is available as Open Source code [5, 6, 7, 8]. To minimize the modifications to the standard kernel code, the real-time scheduler has been developed as a further external scheduler implemented in a loadable kernel module [9].

A small patch (called *"Generic Scheduler Patch"*) applied to the Linux kernel exports the necessary symbols and the relevant events to the real-time scheduler. Based on the information provided by the patch, the real-time scheduler modifies the process priority, raising the selected process to the maximum priority, and then calls the standard Linux scheduler. In practice, the two schedulers co-exist, but the standard Linux scheduler acts only as a dispatcher for the external real-time scheduler. The interface to the scheduler has been exported through the standard

`sched_setscheduler()` system call, adding a new scheduling policy, and extending the structure `sched_param`. The scheduler implements the CBS [10, 5] and the GRUB [11, 12] scheduling algorithms.

The real-time scheduler needs to know all the relevant events happening in the system related to the processes (i.e., process creation, termination, blocking and unblocking). For this reason, the patch puts some *hooks* inside the kernel code that are used to intercept and export the interesting scheduling events.

This approach is very straightforward and flexible, but it adds some overhead (acceptable for most soft real-time applications) and much complexity to the system due to the additional scheduler.

During the implementation and maintainance of the project, the developers noticed that the position of the *hooks* inside the Linux kernel code is the real issue in this approach, and it is not easy to understand where they can be safely placed. This issue made the porting of the code to different releases of the Linux kernel very hard, and it is the main reason why the OCERA code was never offically ported to the 2.6 series of the Linux kernel.

AQuoSA

Parts of the original code of OCERA were used to implement the AQuoSA (*"Adaptive Quality of Service Architecture"*) project [13], which is substantially a rewrite of the original code for the 2.6 Linux kernel. This project maintains all the benefits and drawbacks of the original approach. This project is still a work in progress, and lacks some important features like support for multicore platforms and for newest Linux kernels with CFS scheduler (see Section 2.3). These features may be added in the next future.

LITMUS

The LITMUS project [14], led by Dr. James H. Anderson, is a soft real-time extension of the Linux kernel with focus on multiprocessor real-time scheduling and synchronization. The Linux kernel is modified to support the sporadic task model and modular scheduler plugins. LITMUS is a very complex framework, therefore it cannot be integrated in the mainstream Linux kernel.

The project includes plugins for several scheduling policies, but only supports the Intel x86-32 and Sparc64 architectures (i.e., no embedded platforms). Moreover, it is not integrated with the Control Groups filesystem and the support for the CFS scheduler has been added only very recently.

SCHED_SOFTRR

A Resource Reservation scheduling policy for Linux has been developed also by Davide Libenzi with the `SCHED_SOFTRR` project [15]. Using this policy, a process can run with real-time priority, but it is subject to a constraint on the maximum processor time it can consume. Thus, non-privileged users can have deterministic latencies when running time-sensitive applications, while system stability and fairness are enforced by the bound.

SCHED_ISO, RSDL and SD

Con Kolivas is a Linux developer, most notable for his work on processor scheduling. He is very active in the scheduler subsystem and implemented several schedulers for Linux.

The first scheduling policy, called `SCHED_ISO` [16] (which stands for *"Isochronous Scheduling"*) does not require superuser privileges and is starvation-free. Processes running under the `SCHED_ISO` policy actually execute as `SCHED_RR` unless the processor usage exceeds a specified limit (i.e., 70%). The value of this limit can be configured through the Linux `proc` filesystem.

Kolivas developed several other processor schedulers such as the Rotating Staircase Deadline (RSDL) [17] and the Staircase Deadline (SD) [18] schedulers to address interactivity concerns of the Linux kernel with respect to desktop computing.

These schedulers were not been accepted mainstream, mostly for maintainance issues, and on July 2007, Kolivas announced that he would cease developing for the Linux kernel. However, Ingo Molnar stated that Kolivas' implementation of "fair scheduling" inspired him to develop his CFS scheduler (see next section) as a replacement for the earlier O(1) scheduler.

2.3 State of the art: CFS

Eventually, a kind of Resource Reservation scheduler called *"Completely Fair Scheduler"* (CFS) has been implemented by Ingo Molnar and merged in Linux 2.6.23 as replacement for the previous O(1) Linux scheduler [19, 2].

Overview

80% of CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware. "Ideal multi-tasking CPU" is a theoretical CPU that has 100% physical power and which can run each process at precise equal speed, in parallel, each at 1/nr_running speed. For example: if there are 2 processes running, then it runs each at 50% physical power — i.e., actually in parallel.

On real hardware, we can run only a single process at once, so we have to introduce the concept of "virtual runtime". The virtual runtime of a process specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a process is its actual runtime normalized to the total number of running processes.

Few implementation details

In CFS the virtual runtime is expressed and tracked via the per-process `p->se.vruntime` (nanosec-unit) value. This way, it is possible to accurately timestamp and measure the "expected CPU time" a process should have gotten. Notice that on "ideal" hardware, at any time all processes would have the same `p->se.vruntime value` — i.e., processes would execute simultaneously and no process would ever get "out of balance" from the "ideal" share of processor time.

The process picking logic of CFS is based on this `p->se.vruntime` value and it is thus very simple: CFS always tries to run the process with the smallest `p->se.vruntime` value (i.e., the process which executed least so far). CFS always tries to split up processor time between runnable processes as close to "ideal multitasking hardware" as possible.

Most of the rest of the design of CFS just falls out of this really simple concept, with a few add-on embellishments like nice levels, multiprocessing and various algorithm variants to recognize processes that block often.

The rbtree

The Red-Black Tree (*"rbtree"* [20]) is a type of self-balancing binary search tree, used for storing sortable key/value data pairs. It provides bounded worst case performance for insertion and deletion, with slightly slower (but still O(log n)) lookup time.

The design of CFS is quite radical: it does not use the old data structures for the runqueues, but it uses a time-ordered rbtree to build a "timeline" of future process execution, and thus has no "array switch" artifacts (by which both the O(1) and Kolivas' RSDL/SD schedulers were affected).

CFS also maintains the `rq->cfs.min_vruntime` value, which is a monotonic increasing value tracking the smallest vruntime among all processes in the runqueue. The total amount of work done by the system is tracked using `min_vruntime`. This value is used to place newly activated entities on the left side of the tree as much as possible.

The total number of running processes in the runqueue is accounted through the `rq->cfs.load` value, which is the sum of the weights of the processes queued on the runqueue.

CFS maintains a time-ordered rbtree, where all runnable processes are sorted by the `p->se.vruntime` key (there is a subtraction using `rq->cfs.min_vruntime` to account for possible wraparounds). CFS picks the "leftmost" process from this tree and executes it. As the system progresses forwards, the executed processes are put into the tree more and more to the right — slowly but surely giving a chance for every process to become the "leftmost process" and thus get on the processor within a deterministic amount of time.

Summing up, CFS works like this: it runs a process a bit, and when the process schedules (or a scheduler tick happens) the process' CPU usage is "accounted for": the (small) time it just spent using the physical CPU is added to `p->se.vruntime`. Once `p->se.vruntime` gets high enough so that another process becomes the "leftmost process" of the time-ordered rbtree it maintains (plus a small amount of "granularity" distance relative to the leftmost process so that we do not over-schedule processes and trash the cache), then the new leftmost process is picked and the current process is preempted.

Some features of CFS

CFS uses nanosecond granularity accounting and does not rely on any detail related to the timer's tick rate (e.g., `jiffies` or `HZ` variables). Thus, the CFS scheduler has no notion of "timeslices" in the way the previous scheduler had, and has no heuristics whatsoever. There is only one central tunable (you have to switch the `CONFIG_SCHED_DEBUG` option):

`/proc/sys/kernel/sched_min_granularity_ns`

which can be used to tune the scheduler from "desktop" (i.e., low latencies) to server (i.e., good batching) workloads. It defaults to a setting suitable for desktop workloads. `SCHED_BATCH` is handled by the CFS scheduler module too.

Due to its design, the CFS scheduler is not prone to any of the existing tests explicitly designed to make the scheduler behave not fairly. Under CFS all the existing tests work fine: they do not impact the interactivity of the scheduler, which produces the expected behavior in any circumstance.

The CFS scheduler has a much stronger handling of nice levels and `SCHED_BATCH` than the previous scheduler: both types of workloads are isolated much more aggressively.

Scheduling classes

Actually the "completely fair" algorithm is only a part of the CFS scheduler. The new CFS scheduler has been designed in such a way to introduce *"Scheduling Classes"*, an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming too much about them. In other words, CFS is a modular scheduler, composed by some (currently only two) inner schedulers (called scheduling classes).

The first inner scheduler resides in the `sched_fair.c` file. It implements the CFS algorithm described above, and it handles processes having the following policies:

- `SCHED_NORMAL` (traditionally called `SCHED_OTHER`): this is the scheduling policy used for regular processes.
- `SCHED_BATCH`: this policy does not preempt as often as regular processes would, thereby allowing processes to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.
- `SCHED_IDLE`: this policy is even weaker than nice 19, but it is not a true idle time scheduler in order to avoid to get into priority inversion problems which would deadlock the machine.

The second scheduling class has higher priority than the previous one, and it resides in the `sched_rt.c` file. This class handles processes having `SCHED_FIFO` or `SCHED_RR` policies, and behaves according to the POSIX semantics. The implementation is simpler than in the previous O(1) scheduler: it uses 100 runqueues (for all 100 RT priority levels, instead of 140 as in the previous scheduler) and it needs no expired array (which simplifies the implementation).

Implementation of scheduling classes

Scheduling classes are implemented through the `sched_class` structure, which contains hooks to functions that must be called whenever an interesting event occurs. This is the (partial) list of the hooks:

- `enqueue_task(...)` Called when a task enters a runnable state. On the `sched_fair.c` scheduling class, it puts the scheduling entity (task) into the red-black tree and increments the nr_running variable. `SCHED_EDF` as well, will use a red-black tree data structure to reduce the time of access and manipulation of data.
- `dequeue_tree(...)`

  When a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the nr_running variable.
- `yield_task(...)`

  This function is basically just a dequeue followed by an enqueue, unless the `compat_yield` sysctl is turned on; in that case, it places the scheduling entity at the right-most end of the red-black tree.
- `check_preempt_curr(...)`

  This function checks if a task that entered the runnable state should preempt the currently running task.

- `pick_next_task(...)`

  This function chooses the most appropriate task eligible to run next.

- `set_curr_task(...)`

  This function is called when a task changes its scheduling class or changes its task group.

- `task_tick(...)`

  This function is mostly called from time tick functions; it might lead to process switch. This drives the running preemption.

- `task_new(...)`

  The core scheduler gives the scheduling module an opportunity to manage new task startup. The CFS scheduling module uses it for group scheduling, while the scheduling module for a real-time task does not use it.

Control Groups

Normally, the scheduler operates on individual tasks and strives to provide fair processor time to each task. Sometimes, it may be desirable to group tasks and provide fair processor time to each task group. For example, it may be desirable to first provide fair processor time to each user on the system and then to each task belonging to a user.

Control Groups (cgroups) are an extension to CFS which allows to group tasks and divide processor time in several different ways among such groups. The mechanism is controlled by the following kernel options:

- `CONFIG_GROUP_SCHED` is the generic option to enable group scheduling
- `CONFIG_RT_GROUP_SCHED` permits to group real-time (i.e., `SCHED_FIFO` and `SCHED_RR`) tasks.
- `CONFIG_FAIR_GROUP_SCHED` permits to group CFS (i.e., `SCHED_NORMAL` and `SCHED_BATCH`) tasks.

At present, there are only two (mutually exclusive) options to group tasks for processor bandwidth control purposes:

- `CONFIG_USER_SCHED` groups tasks based on user id
- `CONFIG_CGROUP_SCHED` allows to manually group tasks using the `cgroups` pseudo filesystem

  This options needs the `CONFIG_CGROUPS` option to be set, and lets the administrator create arbitrary groups of tasks, using the `cgroup` pseudo filesystem. See `Documentation/cgroups.txt` for more information about this filesystem.

Notice that only one of these two options to group tasks can be chosen at compile time, and not both.

When `CONFIG_USER_SCHED` is defined, a directory is created in sysfs for each new user and a `cpu_share` file is added to that directory.

```
# cd /sys/kernel/uids
# cat 512/cpu_share           # Display user 512's CPU share
1024
```

```
# echo 2048 > 512/cpu_share    # Modify user 512's CPU share
# cat 512/cpu_share            # Display user 512's CPU share
2048
#
```

CPU bandwidth between two users is divided in the ratio of their CPU shares. For example: if you would like user `root` to get twice the bandwidth of user `guest`, then set the `cpu_share` for both the users such that `root`'s `cpu_share` is twice `guest`'s `cpu_share`.

When `CONFIG_CGROUP_SCHED` is defined, a `cpu.shares` file is created for each group using the `cgroup` pseudo filesystem. Examples below show how to create task groups and modify their CPU share using the `cgroup` pseudo filesystem.

```
# mkdir /dev/cpuctl
# mount -t cgroup -ocpu none /dev/cpuctl
# cd /dev/cpuctl

# mkdir multimedia      # create "multimedia" group of tasks
# mkdir browser         # create "browser" group of tasks

# #Configure the multimedia group to receive twice the CPU bandwidth
# #that of browser group

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox & # Launch firefox and move it to "browser" group
# echo <firefox_pid> > browser/tasks

# #Launch gmplayer (or your favourite movie player)
# echo <movie_player_pid> > multimedia/tasks
```

Limits of CFS

The biggest limit of the CFS scheduler is that it is not a real-time scheduler. To understand why, we first need to introduce some basic concepts about real-time theory [21].

A real-time system is *a computing system in which computational activities must be performed within predefined timing constraints* [22]. Typically, a real-time system is implemented as a set of concurrent tasks that are executed on a Real-Time Operating System (RTOS) [21]. These tasks, called *"real-time"*, are *executable entities of work characterized, at least, by a worst case execution time and a timing constraint* [23].

Depending on the type of application, different timing constraints (like the jitter on the initial or the finishing time of execution) can be defined. A typical constraint is the *deadline*, which is the instant of time the task's execution is required to be completed. A real-time task must complete before its deadline, otherwise the results could be produced too late to be useful.

By knowing the worst case execution time (i.e., the amount of computational resources needed in the worst case) and the deadline of each task, it is possible to develop some scheduling algorithm which allows to schedule all tasks and meet all deadlines.

For periodic real-time tasks, a typical assumption in the real-time literature [21] is that the current deadline corresponds to the end of the current period.

Currently, on CFS, the period of the Resource Reservation mechanism is fixed for all tasks and cannot be different from one task to another. This makes the implementation of most real-time scheduling algorithms (like Rate Monotonic or Earliest Deadline First) impossible.

CFS is a kind of Resource Reservation mechanism, but not in the sense the real-time community uses this word. On CFS, in fact, it is not possible to set a period or a deadline for a task, but it is possible only to assign the share of the resource to a task (or to a group of tasks) that will be respected according to a global period.

2.4 A new Resource Reservation scheduler: SCHED_EDF

General description

In the ACTORS project a completely new Resource Reservation mechanism for the Linux kernel will be implemented. Its name is SCHED_EDF, and it is a real-time scheduling algorithm.

This mechanism will exploit the modular feature of the CFS scheduler to add a further scheduling class which will handle tasks generated by CAL. In particular, the final Linux scheduler will act as follows:

- `SCHED_NORMAL` and `SCHED_BATCH` tasks will be still handled by the standard Linux proportional scheduler scheduling class of CFS;
- `SCHED_RR` and `SCHED_FIFO` tasks will be still handled by the POSIX scheduling class of CFS;
- Tasks generated by CAL will be handled by our new real-time scheduling class. The scheduling policy associated to these tasks will be called `SCHED_EDF` (which gives the name to the mechanism itself) and will have highest priority in the system.

Features and characteristics

The Resource Reservation mechanism based on SCHED_EDF will have the following features:

- It will be developed from scratch, without starting from any existing project
- It will be aligned with the current mainstream kernel: it will work on Linux kernels having CFS scheduler
- It will be integrated with the CFS scheduler itself
- It will be integrated with the cgroups mechanism and will exploit the cgroups interface
- It will natively support multicore platforms

In particular, the SCHED_EDF scheduling class will implement a partitioned Earliest Deadline First (EDF) algorithm. This means that there will be at least one EDF queue for each processor in the system.

Several differences exist between SCHED_EDF and the current version of AQuoSA [13]. First of all, they have a completely different approach. The current version of AQuoSA puts some hooks inside the Linux scheduler to export relevant events, and implements the Resource Reservation scheduler in a external kernel

module. SCHED_EDF, instead, implements a real-time scheduler inside the Linux scheduler itelf, by exploiting the modular design of CFS.

On AQuoSA, the real-time scheduler manages only aperiodic processes, and periodicity is completely handled at user-level: the `qmgr_end_cycle()` function (implemented at user-level) stops and resumes the process using the common system calls provided by Linux. On SCHED_EDF, instead, we investigate a different approach, by making the kernel aware of process periodicity (if any) through the `sched_setscheduler2()` system call (see below).

Last but not least, AQuoSA does not yet support newest Linux kernels with CFS or multicore platforms. SCHED_EDF, instead, will work only on latest Linux kernels (since it exploits features of CFS itself) and will have native support for multicore systems and for ARM platforms.

API

The SCHED_EDF API deals with budgets and periods [2].

Currently, the following API is exported to user-level to handle SCHED_EDF processes:

- A new system call `sched_setscheduler2(...)` allows to create or modify process' budget and period. Notice that even if the process is not periodic, in order to use `SCHED_EDF`, the user has to provide some values for budget and period. The meaning is that the process is allowed to execute at most for a time equal to "budget" every "period".

- For periodic processes, the `sched_setscheduler2()` system call can be used also to inform the kernel that the current instance of the process has finished execution. In this case, the current process is blocked until the end of the current period (if no argument is given, otherwise the process is blocked for the time provided as argument). This system call is invoked by the process itself at the end of each period. If the process belongs to a group of processes, the other processes inside the group are free to continue execution.

- On multicore platforms, the system call `sched_setaffinity(...)` allows to specify which core must run the process. This is invoked by the Resource Manager to sort the processes between different cores. Notice that `sched_setaffinity(...)` works also with threads: according to the manual page, the value returned from a call to `gettid(2)` can be passed as argument.

For what concerns groups of processes, the cgroups interface allows to group processes on the same virtual processor (in the same way as shown in the previous section) and to set or change the budget and period of the virtual processor.

A group is bounded to a CPU and can accept processes that run on the same core. It is possible to change the period and budget using the `edf_runtime_us` and `edf_period_us` files. These files are created once the cgroup filesystem is mounted.

The following example shows how to manage a SCHED_EDF process.

```
#define gettid() syscall(__NR_gettid)
#define sigev_notify_thread_id _sigev_un._tid

#define SCHED_EDF               4

struct sched_edf_param {
```

[2] We will use (budget,period) instead of (alpha,delta) pairs to encourage the merge of this scheduling class inside the Linux kernel.

```
        int sched_priority;
        struct timespec period;
        struct timespec budget;
};

/* Struct to transfer parameters to the thread */
struct thread_param {
        pthread_t thread_id;
        char *name;
        struct timespec wcet;
        struct timespec deadline;
        struct timespec period;
        int signal;
        int clock;
        struct thread_param *next;
};

static int shutdown;

/*
 * timer thread
 */
void *timerthread(void *param)
{
        struct thread_param *par = param;
        struct sched_edf_param sched_edfp;
        struct sigevent sigev;
        sigset_t sigset;
        timer_t timer;
        struct timespec now, next, interval;
        struct itimerspec tspec;
        int policy;
        int nrun=0;
        int pid;

        policy = SCHED_EDF;

        interval.tv_sec = par->period.tv_sec;
        interval.tv_nsec = par->period.tv_nsec;

        sigemptyset(&sigset);
        sigaddset(&sigset, par->signal);
        sigprocmask(SIG_BLOCK, &sigset, NULL);

        memset(&sched_edfp, 0, sizeof(sched_edfp));

        sched_edfp.sched_priority = 0;
        sched_edfp.period.tv_sec = par->deadline.tv_sec;
        sched_edfp.period.tv_nsec = par->deadline.tv_nsec;
        sched_edfp.budget.tv_sec = par->wcet.tv_sec;
        sched_edfp.budget.tv_nsec = par->wcet.tv_nsec;
        sched_setscheduler2(0, policy,
                        (struct sched_param *)(&sched_edfp));

        /* Get current time */
        clock_gettime(par->clock, &now);
        next = now;
        next.tv_sec++;
        pid = gettid();
        sigev.sigev_notify = SIGEV_THREAD_ID | SIGEV_SIGNAL;
        sigev.sigev_signo  = par->signal;
        sigev.sigev_notify_thread_id = pid;
        timer_create(par->clock, &sigev, &timer);
        tspec.it_interval = par->period;
        tspec.it_value = next;
        timer_settime(timer, TIMER_ABSTIME, &tspec, NULL);

        printf("process %d: START\n", (int)gettid());
        while (!shutdown) {
                int sigs;
                if (sigwait(&sigset, &sigs) < 0)
                        goto out;
                printf("process %d: running cycle %d\n", pid, nrun);
                nrun++;
```

```
        }

out:
        /* switch to normal */
        sched_edfp.sched_priority = 0;
        sched_setscheduler2(0, SCHED_OTHER, (struct sched_param *) &sched_edfp);

        return NULL;
}

static void sighand(int sig)
{
        shutdown = 1;
}



int main(int argc, char **argv)
{
        struct thread_param tpar;

        signal(SIGINT, sighand);
        signal(SIGTERM, sighand);

        pthread_create(&(tpar.thread_id), NULL, timerthread, &tpar);

        /* ... */
}
```

Chapter 3

Introduction to (α, Δ) servers

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 [24, 25]. 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 [26]. In real-time scheduling, this resource abstraction is called "bounded delay time partition" [27, 28]. However we rename this mechanism as (α, Δ) server model, to stress the dependency on the two constituting parameters α (the bandwidth) and Δ (the delay).

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

The supply function

First we introduce some definitions that are common in reservation-based scheduling theory [27, 29, 30, 31], even though introduced with different terminology.

Definition 1 *A partition $P \subseteq \mathbb{R}$ is a countable union of non-overlapping intervals*[1]

$$P = \bigcup_{i \in \mathbb{N}} [a_i, b_i) \quad a_i < b_i \leq a_{i+1}. \tag{3.1}$$

[1] The mathematical development does not change if P is any Lebesgue measurable set.

The characteristic function $I_P : \mathbb{R} \to \{0, 1\}$ of the partition P is defined by

$$I_P(t) = \begin{cases} 1 & t \in P \\ 0 & t \notin P \end{cases}$$

We distinguish between *static partitions* and *dynamic partitions*. 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, a dynamic resource allocation mechanism uses some rule for dynamically allocating the resource (for example allocating Q time units every period P). Therefore, a dynamic 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 P, we define the supply function $Z_P(t)$ as the* minimum amount of time *provided by the partition in every time interval of length $t \geq 0$, that is*

$$Z_P(t) = \min_{t_0 \geq 0} \int_{t_0}^{t_0+t} I_P(x)\, \mathrm{d}x. \tag{3.2}$$

If the partition is static, Equation (3.2) can be readily used to compute the supply function (in Section 3.1 we show some example of computation).

However if the partition is dynamic then it is not known in advance when the time will be allocated. To extend the definition of supply function also to servers allocating time by dynamic partitions, we introduce the following definitions.

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

Notice that if the server $\mathbf{S}$ allocates statically the time by a static partition P, then $\mathsf{legal}(\mathbf{S})$ is constituted by the unique element P.

We now generalize the supply function to any server.

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 $t \geq 0$,*

$$Z_{\mathbf{S}}(t) = \min_{P \in \mathsf{legal}(\mathbf{S})} Z_P(t). \tag{3.3}$$

Defining α and Δ of a server

The supply function $Z_{\mathbf{S}}(t)$ fully describes the time provided by the server to any time consuming entity requiring it. However it is sometimes convenient to extract the most significant features from the supply function $Z_{\mathbf{S}}(t)$. A convenient abstraction of server is based on the only concepts of *bandwidth* α and *delay* Δ.

The bandwidth α is the average slope of $Z_{\mathbf{S}}(t)$, formally defined as:

$$\alpha = \lim_{t \to \infty} \frac{Z_{\mathbf{S}}(t)}{t}. \tag{3.4}$$

The computation of the value of Δ requires some more efforts. Informally speaking, once we have computed α, the delay Δ is the minimum horizontal displacement such that $\alpha(t - \Delta)$ is a lower bound of $Z_{\mathbf{S}}(t)$. Formally:

$$\Delta = \inf\{d \geq 0 : \forall t \geq 0 \quad Z_{\mathbf{S}}(t) \geq \alpha(t - d)\}. \tag{3.5}$$

It can be noticed that this abstraction is very simple, since it is consituted by only two parameters: the bandwidth α and the delay Δ. The advantage of a simple abstraction is that it can be placed on top of very different server mechanisms. On the other hand the price of simplicity is paid in terms of tightness: a more detailed description of the time provided by a server would allow a tighter usage of resources.

We believe however, that the benefits of a simple abstraction overcomes the loss of computational resource. In the next section we will show how the bandwidth α and the delay Δ are computed from common servers.

3.1 Computing α and Δ of existing servers

In this section we propose several simple examples to demonstrate how to extract the values α and Δ from well known servers.

Static Partition

Suppose we have a reservation $\mathbf{S}$ that statically allocates the first two time units every period eight, starting from time zero. In this case the set of all legal partitions $\mathsf{legal}(\mathbf{S})$ is constituted by only one partition that is

$$P = \bigcup_{k \in \mathbb{N}} [0 + 8k, 2 + 8k)$$

This partition can be easily represented graphically as reported in Figure 3.1. In the

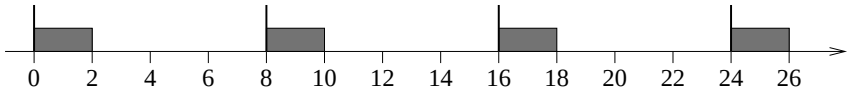

Figure 3.1: A static periodic partition.

figure a thick vertical line represents a period.

Since the partition is statically allocated, then $\mathsf{legal}(\mathbf{S}) = \{P\}$, and we have

$$Z_{\mathbf{S}}(t) = \min_{Q \in \mathsf{legal}(\mathbf{S})} Z_Q(t) = Z_P(t)$$

From Equation (3.2) it follows that for computing $Z_P(t)$, we must find, for all t, what is the worst case starting point t_0. We provide the following simple Lemma that follows directly from the result by Feng and Mok [28].

Lemma 1 *Given the partition*

$$P = \bigcup_{k \in \mathbb{N}} [a_k, b_k)$$

then

$$Z_P(t) = \min_{t_0 \in \{b_k\}} \int_{t_0}^{t_0+t} I_P(x)\, \mathrm{d}x. \tag{3.6}$$

This lemma expresses the very simple intuition that the worst-case condition for an application allocated to a reservation occurs when the time provided has just expired. In fact in Equation (3.6) it can be noticed that the minimum occurs in the set $\{b_k\}$ that are the right boundaries of the intervals. In fact at the right boundary

an idel interval starts. This conrresponds to a candidate worst-case condition fo the supplied resource $Z(t)$.

Applying the lemma to this first simple example we find directly that $Z_P(t)$ can be found when $t_0 = 2$. The resulting $Z_{\mathbf{S}}(t)$ then is the one depicted in Figure 3.2, from which it follows that this reservation has $\alpha = 0.25$ and $\Delta = 6$. More in general,

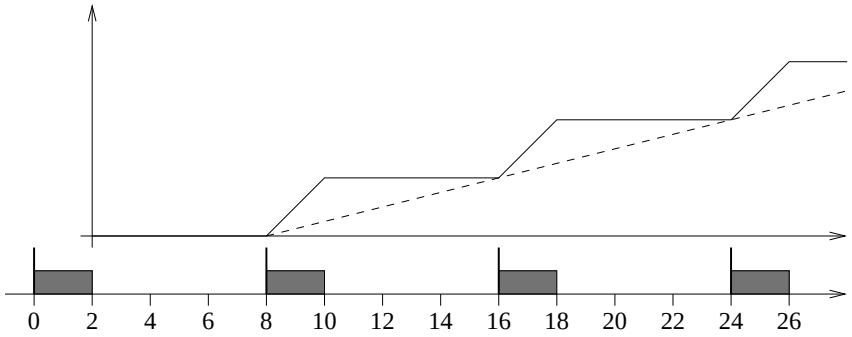

Figure 3.2: The supply function for a static periodic partition.

for a static periodic partition that allocates Q time units every period P we have [28]

$$\alpha = \frac{Q}{P} \qquad \Delta = P - Q \tag{3.7}$$

From this example one would be tempted to affirm the Δ is *always* equal to the length of longest idle interval. However this is not true. We propose the static partition of the Figure 3.3 to show this.

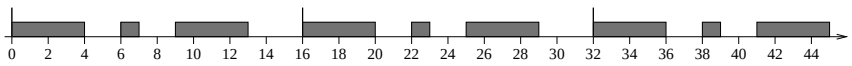

Figure 3.3: A static generic partition.

By the application of the Lemma we must consider t_0 equal to all the possible end of the interval where the resource is allocated. In this case the possibilities for t_0 in one period are $t_0 \in \{4, 7, 13\}$.

In the bottom part of Figure 3.4 we show the possible schedules of the partition starting from each of the candidates. Above we plot the integral from t_0 to $t_0 + t$, as t varies for all the possible t_0. Then we plot the minimum as required by Equation (3.6) by a thick line, that is $Z_{\mathbf{S}}(t)$.

For this supply function we compute both $\alpha = \frac{9}{16}$ and $\Delta = 5 - \frac{16}{9} = 3.222$. Notice that in this case Δ is larger than the longest idle time (that is 3) because the linear lower bound is constrained by the point $(5, 1)$.

Dynamic Partitions

A dynamic partition occurs when the allocation of time is not assigned statically. Instead the allocation obeys to some rules. In this case the set $\mathsf{legal}(\mathsf{S})$ is not constituted by a unique partition, but it contains all the possible partition that can be generated. For example the partitions reported in Figure 3.5 are all legal partitions of a periodic server implemented by a task whose computation time is $Q = 3$, period $P = 6$, and deadline $D = 8$.

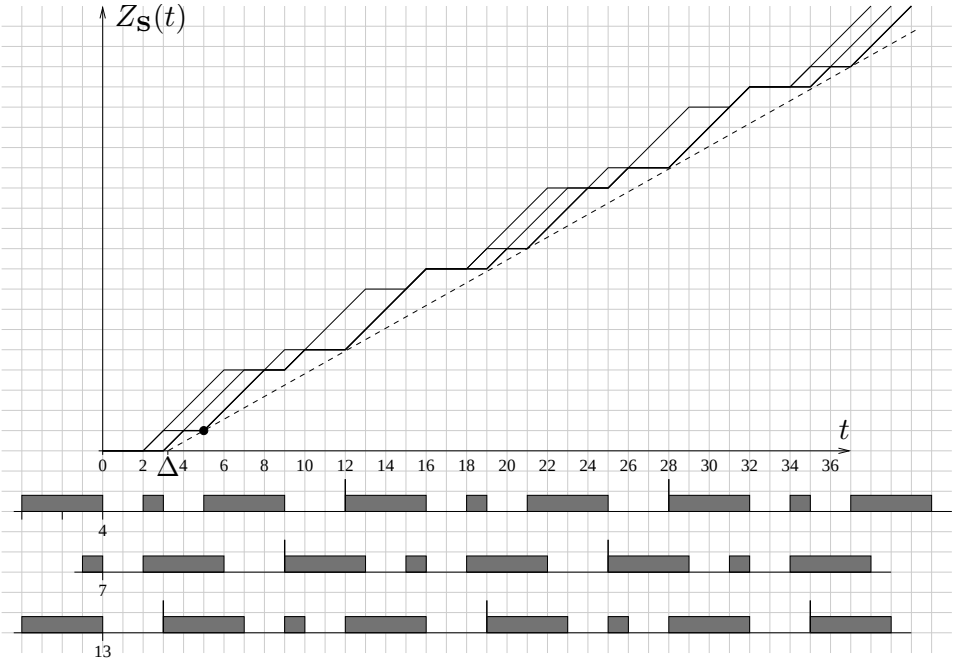

Figure 3.4: A static generic partition.

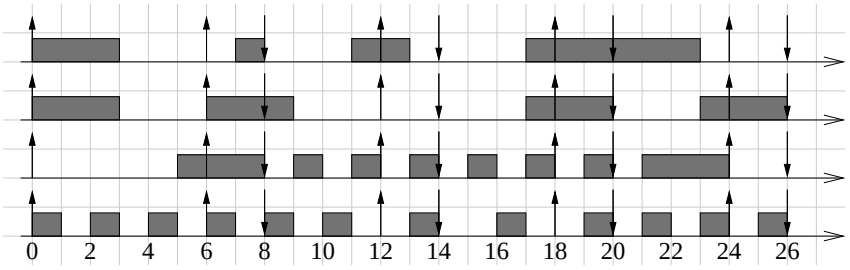

Figure 3.5: Example of dynamic partitions

The worst-case partition occurs when t_0 coincides with the expiration of a time quantum Q that is allocated at the beginning of the allocation period and the next quantum is received at the end the period. This scenario is reported in Figure 3.6. Hence, in the case of a periodic server having parameters (Q, P, D) the correspond-

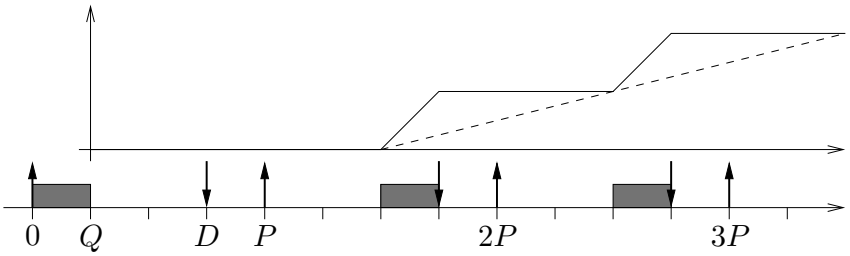

Figure 3.6: The supply function of a periodic server.

ing (α, Δ) parameters are [32]:

$$\alpha = \frac{Q}{P} \qquad \Delta = P + D - 2Q \tag{3.8}$$

Finally, it is also possible to map a pfair task [24, 25, 33] onto an (α, Δ) server. However we do not report here this development, since it is quite complicated. The description of the algorith is available [34].

3.2 The design problem

By the design problem we mean the selection of the (α, Δ) server that minimizes some goal function (typically, to minimize the used bandwidth) such that some constraint is satisfied (typically, the application mapped onto the server must be fulfill the timing requirements).

Suppose that a server must schedule a periodic task which requires C units of time every period T (we call $U = \frac{C}{T}$ the task utilization). Then the (α, Δ) servers that can schedule is are such that:

$$Z(T) = \alpha(T - \Delta) \geq C \tag{3.9}$$

meaning that the time provided must be greater than or equal to the computation time C. This constraint is also represented in Figure 3.7. The left part of the figure

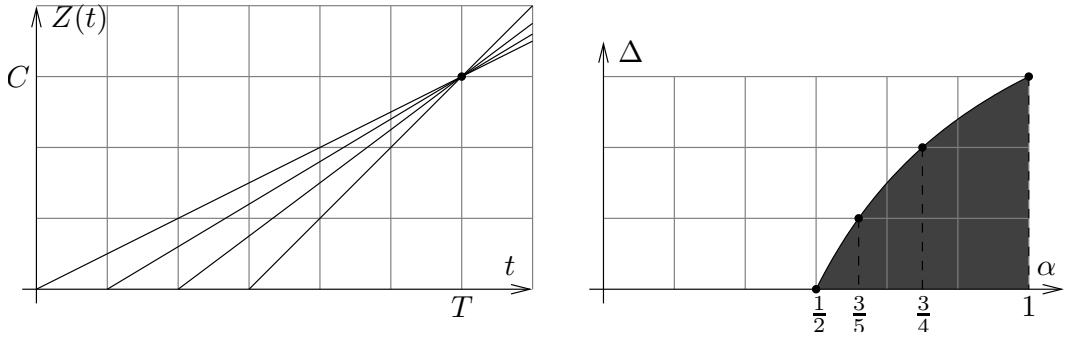

Figure 3.7: Depicting the possible (α, Δ) servers.

shows the point (T, C) (that is $(6, 3)$ in the figure) and four possible supply functions. The same situation is reported on the right part of the figure. In this case, instead, we plot in gray the region of (α, Δ) servers that can guarantee the service specified by Eq. (3.9). The dots along the curve represents the four special cases that are also plot in the left part.

Suppose that the goal the we want to achieve is to minimize the used bandwidth, then the solution of the design problem is

$$\min \alpha = \min \frac{C}{T - \Delta} \quad \Rightarrow \quad \alpha = \frac{C}{T} = U, \Delta = 0$$

However, this solution (also indicated by the leftmost dot in the right part of Figure 3.7) is quite not satisfactory. In fact it tells that the best choice is to set $\Delta = 0$ that is not possible, because if $\Delta = 0$ the impact of the overhead becomes huge. In fact if we account also for the cost C_s of switching between two reservations in the goal to be minimizes, we are able to find a solution that is more significant. In this case the goal to be minimized becomes [30]

$$\min \alpha + \frac{C_s}{P}$$

If we suppose that the server is implemented by a periodic server, then from Eq. (3.7) we have that

$$P = \frac{\Delta}{1 - \alpha}$$

meaning that the bandwidth to be minimized is

$$\min \alpha + C_s \frac{1-\alpha}{\Delta}$$

Since α and Δ must also guarantee the constraint of providing at least C in T (see Eq. (3.9)), α and Δ must satisfy the following constraint

$$\Delta = \frac{\alpha T - C}{\alpha} \tag{3.10}$$

and the overall bandwidth to be minimized that account also for the constraint of Eq. (3.10) is

$$\min \alpha + C_s \frac{\alpha(1-\alpha)}{\alpha T - C}$$

Fortunately, this problem can be solved analytically. The solution is

$$\alpha = U \left(1 + \sqrt{1 - \frac{1 - \frac{C_s}{C}}{1 - \frac{C_s}{T}}} \right) \tag{3.11}$$

If we assume that C_s is small enough w.r.t. C and T, we can approximate Equation (3.11) at the first order. This allows to find

$$\alpha \approx U \left(1 + \frac{C_s}{T\sqrt{U}} \right) \tag{3.12}$$

Notice that if $C_s = 0$ we find again $\alpha = U$.

Chapter 4

Functionality supported by the interface

The following chapter describes the functionality that the interface between the resource manager (RM) and the underlaying reservation and scheduling (RS) layer should support and the type of information that should be passed over this interface. How the interface is implemented and the actual API will not be detailed here, although it is likely that the cgroup file system interface will be involved. It is also likely that the interface internally is implemented using several mechanisms, e.g., both the cgroup interface and a D-Bus interface.

4.1 From the resource manager

The RM needs to perform, at least, the following operations involving the RS layer. Some of the operations below may also be performed directly from the applications, i.e., by a call to the corresponding system call.

- Creation and deletion of reservations. It should be possible for the RM to create a new reservation or to remove a reservation.

- Execution allocation. It should be possible for the RM to assign a process to a reservation as well as removing a process from a reservation.

- Reservation allocation. It should be possible for the RM to specify on which core a reservation should run, in the case the reservation is not allowed to migrate between cores. This is realised by the `sched_setaffinity` system call.

- Change global parameters. The RM should be able to modify parameters that affect the operation of the RS layer globally. This could, for example, mean changing the operation of the bandwidth server system in some way or informing the scheduling subsystem that a particular core should be turned off or on again.

- Change reservation parameters. The RM should be able to change the parameters of each server. In the case the reservation system is based on a periodic bandwidth server of the CBS-family then this would correspond to changing the server period, P, and the server budget, Q. If instead a CFS scheduler is used it would only be the share (budget) that could be changed per reservation, whereas the period would be a global parameter that is common to all

reservations. Since the RM interally uses the (α, Δ) representation a conversion is needed between these parameters and the parameters applicable to the RS system being used.

4.2 To the resource manager

The following information, at least, needs to be provided from the RS layer to the RM.

- Resource consumption. The interface should support that the RS layer informs the RM about how much resources that are consumed within each reservation. In the `SCHED_EDF` case this is expressed in terms of the used budget of the reservation. A value that is less than the assigned budget indicates that the latter is too large, whereas a value that is equal to the assigned budget indicates that the latter is exactly correct or too small. The reason why the used budget may never be larger than the assigned budget is that `SCHED_EDF` enforces hard reservations, in contrast to the classical CBS mechanism.

  In order not to burden the RM with too frequent information the budget information should not be sent too often. Several possibilities exist, the simplest being to report this periodically. The reporting period may be, for example, proportional to the delay parameter Δ, since this parameter measures the sensitivity of the application to the time granularity. Hence it is reasonable to expect that an application that can sustain a large Δ can be notified the bandwidth consumption less frequently. At compile time, an integer variable would specify the default number of periods on which the average should be computed on. At run-time, a file in the cgroup filesystem may allow to set this variable to a different value. This way, it would be possible to set a specific value for each virtual processor. Another possibility is to only report when a sufficiently large change in budget consumption has occurred, i.e., to have event-driven rather than time-driven reporting.

  It could also be useful for the RM to obtain information not only about average values but also of how much the used budget varies from one server period to another, i.e., providing some higher order moment or the maximum and minimum values over the reporting period.

  The budget actually used by the application will be made available by the Reservation Layer through some mechanism. This may be a read-only file in the cgroup filesystem or a particular system call. In the former case, the information would be available only for groups of file (i.e. virtual processors) and it would not be possible to have this information for a `SCHED_EDF` process running outside a virtual processor. This is not a problem, however, because we can always create a virtual processor containing only one process.

  Since the RM uses (α, Δ) representation the used budget needs to be concerted into used bandwidth.

- Core availability. Assuming that it is possible for the scheduling or hardware layer to turn off cores or execute cores at a lower clock frequency in order to save power, this is also something that needs to be reported to the RM.

Chapter 5

Resource Allocation

The resource manager (RM) is responsible for the allocation of resources to the different application. Although several resource types could be considered, here the focus will be on CPU resources. The output from the resource manager consists of information to the underlying reservation and scheduling layer and information to the applications that are executing within the control of the resource manager. The resource allocation will be performed when the application set changes, e.g. when a new application is started, and during the execution of the applications, e.g., when the consumed resources as measured by the reservation layer deviate too much from the allocated resources.

Performing the resource allocation will for realistic application sets be rather time consuming. Therefore it is necessary to avoid performing it too often. The allocation problem can be viewed as a combined optimization and control problem. Several solution techniques are plausible, e.g., heuristic approaches, linear programming, mixed integer linear programming, quadratic programming, etc. The details of this are not part of this document. However, it is likely that the implementation will contain a substantial element of heuristics, moving the focus from optimal solutions to sub-optimal and less time consuming solutions.

The resource manager should be able to support both CAL applications and non-CAL applications. However, since the focus of ACTORS is CAL applications this is the only case that is described here. The CAL applications considered can be divided into two types:

- 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 and the requirements are highly data-dependent. As a consequence of this it is also for these applications that dynamic resource allocation is most needed.

For static CAL applications, e.g. most control applications, the CAL network can be converted to a static precedence graph for which off-line schedulability analysis can be applied. The output of this analysis consists of the number of virtual processors to be used, and the parameters for these. Although this is partly the same information that the resource manager generates dynamically, the off-line schedulability analysis is not considered part of the resource manager. Instead, for these applications the corresponding parameters are considered as fixed and provided to the resource manager as inputs. However, it should be possible for the resource manager to dynamically adjust the resource parameters also for static applications using on-line measurements, e.g., in the case the consumed resources

differ substantially from the allocated resources. A reason for this could, e.g., be that an incorrect worst-case execution time estimate has been used in the schedulability analysis.

5.1 Resource Manager Inputs

The inputs to the resource manager can be divided in two types:

- static inputs, and
- dynamic inputs.

Static inputs can be viewed as configuration information that is not considered to change during the execution of an application, at least not very often. One example could be information from an application about its service levels and resource requirements. Dynamic inputs correspond to sensor information about the actual resource consumption or the achieved quality of an application. This information is provided on-line to the resource manager and can be used to adjust the allocation.

Static Inputs

The static inputs consist of the following information

- Service level tables. The service level tables contain information about the different service levels that an application can execute at. Associated with each level are the resource requirements and the quality obtained. The values are expressed as integers and are relative within each application. Hence, the information in the service level tables can not be used as a basis for deciding how resources should be divided between different applications. The service levels can be both discrete and continuous. It is also possible to associate multiple resource types, e.g., CPU time and disk bandwidth, with an application. Here, we will, however, only consider CPU time. Associated with the CPU time the table also provides the CPU time granularity. This is an absolute value that corresponds to the delay, that is, the longest amount of time that the application may need to wait for being assigned some resource.

- Application importance. The application importance is a weight or priority number that reflects how important an application is in relation to other applications, i.e., how important it is that the resource manager should be able to meet the resource requirements of that application. This number can be used to decide how resources should be distributed between applications.

- CPU time requirement category. In order to better be able to compare the CPU time requirements of different applications each application is also assigned to a certain CPU time requirement category, e.g. “Low CPU time requirements” or “High CPU time requirements”. The resource manager can use these categories as a very rough hint on how to perform the initial resource allocation.

- Maximum parallelity. The maximum parallelity is an optional input that may be used to inform the resource manager about the maximum level of parallelity that the application can benefit from. If a value is given that is smaller than the number of cores available it is not useful for the resource manager to assign more virtual processors to the application than that value.

- Optimization objective. The optimization objective is a way for the user to inform the resource manager about what the objective of the optimization should be. Here we will assume that a number of categories are predefined, e.g., "Maximum quality" or "Long battery life". It should also be possible for the user to indicate whether he/she wants to ensure that the resources should be distributed in such way that all the applications should be able to execute at some service level, or if the resources should be allocated to the applications in strict importance order, i.e., first the most important application is given the resources desired, then the second highest, etc, until all the resources are distributed, possibly not giving any resources at all to some applications.

In addition to the above static inputs the parameters resulting from the off-line schedulability analysis of the static CAL applications are also considered as static inputs to the resource manager. This consists of

- Number of virtual processors. This number decides how many virtual processors that will be used for this application. In each virtual processor a single task is executing corresponding to the flow that has been assigned to that virtual processor.
- Bandwidth. The bandwidth for each virtual processor.
- Delay. The delay for each virtual processor.

In order to guarantee the schedulability of these application it is necessary for the resource manager to reserve a sufficient amount of bandwidth for them. This share of the bandwidth should not be considered part of the available resources for the dynamic resource allocation.

Dynamic Inputs

There are two main dynamic inputs to the resource manager.

- Used budget. The used budget is a nanosecond time interval that measures how much of the reserved budget that a virtual processor has used. A value smaller than the reserved budget indicates that the reserved budget is too large and may be decreased. A value that is equal to the reserved budget indicates that the reserved budget is exactly right or too small. There is, however, no way of knowing how much too small it is. In order to not overload the resource manager the used budget is not reported every server period. This can be implemented in several ways:
  - The average value, possibly with some range or variance measure, are sent periodically to the resource manager, The reporting period would then constitute an additional server parameter that the resource manager should be able to set.
  - The value is only reported when a sufficiently large change has occurred.
  - The two approaches above could be combined. Although the values may be reported periodically a new resource allocation is only performed when a sufficient large change has occurred, i.e., a deadband is used.

  Since the resource manager uses bandwidth and delay, the used budget needs to be converted to used bandwidth before the resource allocation is performed.

- Happiness value. The happiness value is a way for an application to report its perceived quality to the resource manager. It is an integer number ranging from 0 to 100 and above. A value of 100 indicates that the application is satisfied with the amount of resources allocated to it. Values below or above 100 indicate that either the application cannot achieve the expected quality corresponding to the current service level, i.e., that it has received too little resources, or that the quality achieved is higher than what was expected, i.e., it has received more resources than necessary.

It is not possible for all applications to estimate their own quality, i.e., the resource allocation cannot assume that this sensor is always available. A similar situation holds for the used budget. Although the used budget may be available it may not always provide any useful information. This is particularly so for the dynamic CAL applications where the code that is executed on the same virtual processor changes dynamically from one period to the next. As a consequence of this the used budget will vary substantially, and it may therefore not be useful to adjust the bandwidth based on this measurement. A possibility for these types of application would be to instead base the decision of if the total allocated bandwidth to the application is too small or too large, on the aggregated used budget information from all the virtual processors that are executing the application.

In addition to these inputs it is also possible for the resource manager to receive information from the hardware layer about power consumption, the number of cores available, etc. This is also information that could trigger a reallocation of resources.

5.2 Resource Manager Outputs

The resource manager has five main outputs.

- Application bandwidth. This is a real-valued number that decides how much bandwidth has been allocated to the application. The maximum value for this number corresponds to the number of cores on the platform.
- Application delay. This number is the delay that has been assigned to the entire application. Normally it is given directly by the CPU time granularity in the service level table.
- Number of virtual processors. The number of virtual processors to be used for the application. For static CAL applications this is decided by the off-line schedulability analysis. For dynamic CAL applications different approaches are possible. One possibility is to always use the same number of virtual processors as there are physical processors. Another possibility is to only use as many virtual processors as necessary to obtain the available bandwidth. For example if an application has been assigned the bandwidth 2.5 then it is enough to use 3 virtual processors.
- Processor allocation. The processor allocation decides to which physical processor a virtual processor should be allocated. For static CAL applications which physical processor to use does not matter as long as the schedulability condition is fulfilled for each of the processors (for CBS this means that the sum of the assigned bandwidth on a processor should be less than 1). However, for dynamic CAL applications and the case of the same number of virtual processors as cores it makes sense to allocate the virtual processors on different cores.

- Service level. The service level is used to inform an application about which service level that it should execute at. The service level could be used by the application as the basis for selecting among different internal modes of operation. The quality level associated with the service level could also be used as a setpoint value for an internal control loop that modifies the resource consumption of the application on order to obtain this quality.

Since `SCHED_EDF` is based on partitioned EDF scheduling a virtual processor may not span multiple cores. Hence, the application bandwidth needs to be distributed onto the virtual processors. Also here, different possibilities exist:

- Equal distribution. Here the total application bandwidth is divided evenly among the virtual processors. For example, using four cores a total bandwidth of 2.6 would be split into four equal parts of 0.65.
- Bin packing. Here we assume that the number of virtual processors to use has been determined from how many that are maximally needed, i.e. for a bandwidth of α the number of virtual processors used is given by $\lceil \alpha \rceil$. In this case the bandwidth is split into $m = \lfloor \alpha \rfloor$ parts of unit bandwidth plus one additional part with bandwidth $\alpha - m$.

In addition to deciding the bandwidths of the virtual processors it is necessary to also decide the delays of the virtual processors from the delay of the total application. Also, here different possibilities exist. One is to use the the value of the total delay also in the virtual processors. Another would be to assign the delays in a way that also takes the bandwidths into account. Which approach that is best is still an open question. It is also an open question whether it is at all possible to distribute the delay in some optimal way without having access to more detailed application knowledge.

For the static CAL applications the bandwidths and the delays for the virtual processors are given by the schedulability analysis.

Finally the bandwidths and delays for each virtual processor have to be translated into the actual server parameters supported by the underlying reservation system. In the case of periodic servers parametrized by the budget and period this is done as

$$P_i = \frac{\Delta_i}{2(1-\alpha_i)}$$
$$Q_i = \frac{\Delta_i \alpha_i}{2(1-\alpha_i)}$$

In addition to the main outputs above the resource manager also needs to provide output to the reservation layer about the creation and deletion of virtual processors. This is described in more detail in Chapter 4.

The structure of the resource manager is shown in Fig. 5.1. In the figure the delay is indicated as a dynamic output, although it in ACTORS most likely will be implemented as a static output. The reason for this is that it would in principle also be possible to adjust the delay based on, e.g., the happiness value if a change in the delay would effect the application quality in a known way. For some application types this would be conceivable.

5.3 When to perform the resource allocation

The resource allocation should be performed in the following situations:

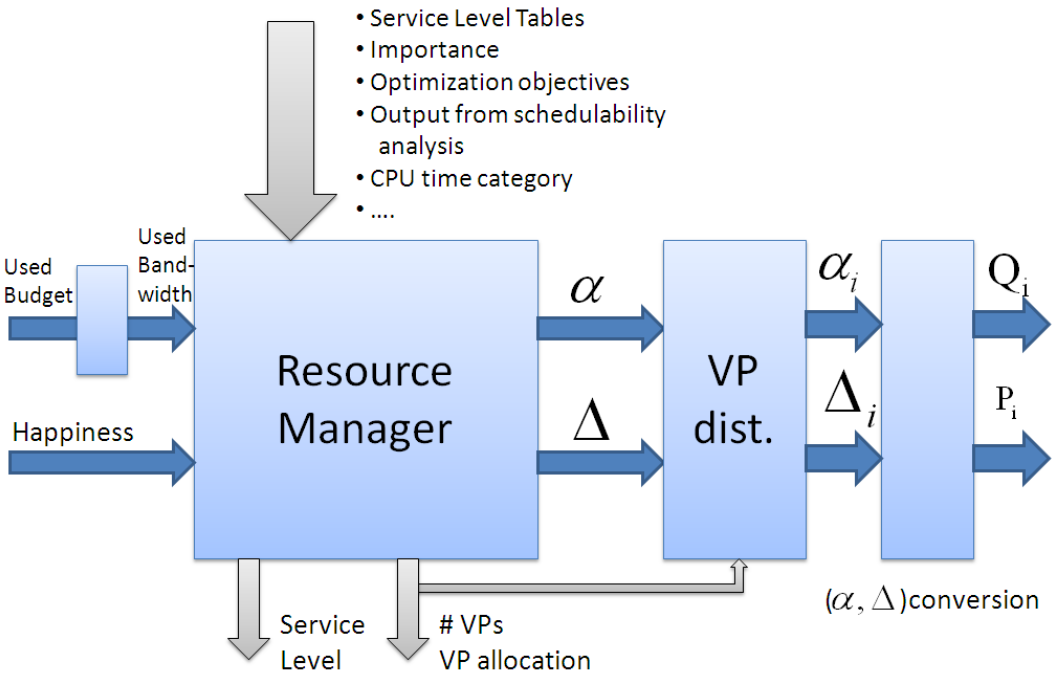

Figure 5.1: The input-output of the resource manager. The inputs and outputs in the horizontal direction corresponds to inputs provided and outputs generated during application execution. The inputs and outputs in the vertical direction correspond to static inputs and outputs. In most cases the signals are vector-valued with one entry in the vector for each application.

- During startup. Here we assume that we have an application set available but since they have not executed there is no knowledge about the used bandwidth or the perceived application quality. Hence, the resources must be allocated in open loop using mainly blind guesses that later can be improved through the use of feedback.

- When an application finishes. When an application finishes resources are returned to the resource manager and these could be used to improve the quality of the remaining applications.

- When an application arrives. When a new application arrives the resource manager needs to decide whether the application should be accepted or not. If unused resources are available those may be used, otherwise the resources provided to the already executing applications have to be decreased, some of them possibly terminated if the new application is important enough.

- When the used bandwidth for a virtual processor is substantially smaller than the allocated bandwidth. In this case the RM also knows how much bandwidth becomes available.

- When the used bandwidth for a virtual processor is equal to the allocated bandwidth. The likely cause is a too small bandwidth, however, the RM has no information about how much too small it is. In this case the RM needs to gradually increase the allocated bandwidth until the used bandwidth is slightly below the allocated, i.e., on the safe side.

- When the application quality is too low. When the perceived quality of an application does not match the expected quality for the given service level the resource manager could respond in a number of ways. One possibility is to increase the total reservation bandwidth, another is to decrease the total delay.
- When the application quality is too high. This is the opposite situation from before. This could indicate that the application receives too much resources.
- When the amount of resources changes caused by, e.g., switching off cores to save battery life or scaling down CPU frequencies to save battery life.
- When the application importance or optimization objectives are changed.

It should be noted that there are two sensors that may be used for deciding if the resources that have been allocated to an application are sufficient, the application quality (the happiness) and the used bandwidth. It is, however, important to have in mind that it is probably rarely the case that both of these are used for the same application. For dynamic CAL applications it will primarily be the application quality that is used, whereas as for static CAL applications it will mainly be the used bandwidth that will be used.

When a resource reallocation is performed caused by the used bandwidth sensor it should be noted that his information is local to a particular virtual processor. If the used bandwidth in the other virtual processors for the application at hand correspond to the allocated bandwidth, it should be sufficient to recalculate the bandwidth (α_i) for that particular virtual processor, and based on that update the global bandwidth for the application, rather than update the bandwidths for all the virtual processors associated with the application.

The resource manager contains feedback at several levels. At the lowest level it consists of a feedback loop that uses the used budget reports to modify the parameters of individual virtual processors. On an intermediate level feedback from the perceived application quality together with aggregated information about the used budgets of all the virtual processors assigned to a particular application can be used to modify the parameters of all the involved virtual processors. Finally, at the highest level feedback from the perceived application quality can be used to modify the applications by changing their service levels.

How these different feedback levels should be designed and implemented will be one of the main topics of year 2 in ACTORS.

Bibliography

[1] R. Love, *Linux Kernel Development, 2nd edition*. Novell Press, 2005.

[2] I. Molnar, ""modular scheduler core and completely fair scheduler"." `http://lkml.org/lkml/2007/4/13/180`.

[3] 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*, 1993.

[4] G. Lipari and C. Scordino, "Linux and real-time: Current approaches and future opportinities," in *IEEE International Congress ANIPLA '06*, (Rome, Italy), Nov. 2006.

[5] L. Abeni and G. Lipari, "Implementing resource reservations in linux," in 4^{th} *Real-Time Linux Workshop*, (Boston, MA), Dec. 2002.

[6] I. Ripoll, P. Pisa, L. Abeni, P. Gai, A. Lanusse, S. Saez, and B. Privat, "WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis," tech. rep., OCERA, 2002.

[7] C. Scordino and G. Lipari, "Energy saving scheduling for embedded real-time linux applications," in 5^{th} *Real-Time Linux Workshop*, (Valencia, Spain), 2003.

[8] C. Scordino and G. Lipari, "Using resource reservation techniques for power-aware scheduling," in *Proceedings of the* 4^{th} *ACM International Conference on Embedded Software (EMSOFT)*, (Pisa, Italy), pp. 16–25, Sept. 2004.

[9] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3^{rd} edition*. O'REILLY, Feb. 2005.

[10] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium*, (Madrid, Spain), IEEE, Dec. 1998.

[11] G. Lipari and S. K. Baruah, "Greedy reclamation of unused bandwidth in constant bandwidth servers," in *IEEE Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, (Stokholm, Sweden), June 2000.

[12] G. Lipari and S. K. Baruah, "A hierarchical extension to the constant bandwidth server framework," in *IEEE Proceedings of the Real-Time Technology and Applications Symposium*, (Taipei, Taiwan), IEEE Computer Society Press, May 2001.

[13] "Aquosa: Adaptive quality of service architecture for the linux kernel."

[14] D. J. H.Anderson and T. U. o. N. C. a. C. H. students, "Litmus rt: Linux testbed for multiprocessor scheduling in real-time systems." `http://www.cs.unc.edu/~anderson/litmus-rt/`.

[15] D. Libenzi, "SCHED_SOFTRR linux scheduler policy." `http://xmailserver.org/linux-patches/softrr.html`.

[16] C. Kolivas, "Isochronous class for unprivileged soft RT scheduling." `http://ck.kolivas.org/patches/SCHED_ISO`.

[17] L. W. News, "The rotating staircase deadline scheduler." `http://lwn.net/Articles/224865/`.

[18] C. Kolivas, "The staircase scheduler." `http://lkml.org/lkml/2004/3/24/208`.

[19] C. Scordino, "Design of the cfs scheduler." `linux/Documentation/scheduler/sched-design-CFS.txt`.

[20] R. Landley, ""red black trees in linux"." `linux/Documentation/rbtree.txt`.

[21] C. Scordino, *Dynamic Voltage Scaling for Energy-Constrained Real-Time Systems*. PhD thesis, University of Pisa, Pisa, Italy, Dec. 2007.

[22] G. C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, Jan. 2005.

[23] J. A. Stankovic, K. Ramamritham, M. Spuri, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[24] S. K. Baruah, N. K. Cohen, G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, June 1996.

[25] 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.

[26] D. Stiliadis and A. Varma, "Latency-rate servers: A general model for analysis of traffic scheduling algorithms," *IEE/ACM Transactions on Networking*, vol. 6, pp. 611–624, Oct. 1998.

[27] A. K. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the* 7^{th} *IEEE Real-Time Technology and Applications Symposium*, (Taipei, Taiwan), pp. 75–84, May 2001.

[28] 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.

[29] L. Almeida, P. Pedreiras, and J. A. G. Fonseca, "The FTT-CAN protocol: Why and how," *IEEE Transaction on Industrial Electronics*, vol. 49, pp. 1189–1201, Dec. 2002.

[30] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, (Porto, Portugal), pp. 151–158, July 2003.

[31] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the* 24th *Real-Time Systems Symposium*, (Cancun, Mexico), pp. 2–13, Dec. 2003.

[32] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using edp resource models," in *Proceedings of the* 28th *IEEE International Real-Time Systems Symposium*, (Tucson, AZ, USA), pp. 129–138, 2007.

[33] J. H. Anderson, A. Block, and A. Srinivasan, "Quick-release fair scheduling," in *Proceedings of the* 24th *IEEE Real-Time Systems Symposium*, (Cancun, Mexico), pp. 130–141, Dec. 2003.

[34] E. Bini, *The Design Domain of Real-Time System*. PhD thesis, Scuola Superiore Sant'Anna, Pisa, Italy, Oct. 2004. available at `http://retis.sssup.it/~bini/thesis/`.