Università di Pisa

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea Specialistica

Design and development of a mechanism for low-latency real time audio processing on Linux

Relatori:

Prof. Paolo Ancilotti
Scuola Superiore Sant'Anna

Prof. Giuseppe Anastasi
Dipartimento di Ingegneria dell'Informazione

Dott. Tommaso Cucinotta
Scuola Superiore Sant'Anna

Candidato:

Giacomo Bagnoli

Anno Accademico 2009/2010

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.

There is another theory which states that this has already happened.

– D. Adams

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Audio on personal computers, and thus in the Linux kernel too, started with simple hardware support 16 bit stereo, half-duplex pulse code modulation (`PCM`) and it has grown to multi-channel mixed analog-digital I/O, high sample rate design of current sound cards. As hardware became more powerful, supporting higher sample rate, higher sample width, digital I/O over S/PDIF or AES/EBU[1], more complex usage pattern became possible, growing from relatively simple MIDI[2] wavetables or MOD playback to digital multi-track recording or to live set performance with software synthesizers driven by real-time user MIDI input.

Computer Music is becoming the standard way to create, record and produce or post post-produce music. *Digital Audio Workstation* `DAW` are nowadays found in almost every new recording studio, from home recording to professional ones, and digital audio is slowly becoming the standard way of moving audio through the studio itself. Moreover, DJs and VJs are moving to computer based setups, so that mixing consoles are reduced from the classic two turntables or two CD players decks to a single laptop with the mixing

[1]Both S/PDIF and AES/EBU are two data link layers and a set of physical layer specifications for carrying digital audio signals between audio devices. S/PDIF (Sony/Philips Digital Interconnect Format), is a minor modification of the AES/EBU (officially AES3, developed by the Audio Engineering Society (AES) and the European Broadcasting Union (EBU)) better suited for consumer electronics.

[2]MIDI (Musical Instrument Digital Interface is industry-standard protocol that enables electronic musical instruments such as keyboard controllers, computers, and other electronic equipment to communicate, control, and synchronize with each other.

software and the Mp3 collection controlled with a USB or OSC[3] interface. Evolution of the audio subsystem of modern operating systems has followed this needs.

Meanwhile live music is leveraging software for real-time synthesis of sound or for post-processing it with several effects, and spotting on stage a MIDI keyboard attached to a common laptop is becoming the rule rather than the exception. Unlike DAW or the DJ uses, when dealing with live music there is an additional parameter to consider when setting up a computer based system, and that is the latency between user input (i.e. key pressed on keyboard, or the sound coming into the effect) and the produced output (i.e. synthesized sound or effected sound).

This is the critical aspect in these situations. The musician wold like to have as low latency as possible, but low latencies imposes on hardware, operating system and on the software running on the computer strict real-world timings, which the system has to catch up in order to produce sounds.

This work is focused on the operating system part, aiming at improving total system reliability and correctness for low latency real-time audio on the Linux kernel, leveraging the results reached by the real-time operating system research, using Resource Reservations from the real-time system theory to provide a certain Quality of Service (QoS) to solve practical problem of the audio world.

This document is organized as follows: Chapter 2 on the following page serves as an introduction to the state of sound subsystem in the Linux kernel, to the Real Time theory and the current implementations of real-time scheduling in the linux kernel. Chapter 3 on page 41 describes the patches and the software written to support and implement QoS in linux audio programs. Chapter 4 on page 62 illustrates all the results of the experiments, such as performance and overhead evaluation. Chapter 5 on page 90 then sum up results, containing the conclusions of the work and possible future extensions.

[3] *OpenSound Control* (`OSC`) is a content format for messaging among computers, sound synthesizers, and other multimedia devices that are optimized for modern networking technology.

Chapter 2

Background

2.1 The State of Linux Audio subsystem

Initial support for audio playback and capture in the Linux kernel was provided by the *Open Sound System* (`OSS`). OSS API was designed for the audio cards with 16bit two-channel playbacks and captures, and the API followed the standard `POSIX` via `open()`, `close()`, `read()` and `write()` system calls. The main problem with OSS was that, while the file based API was really easy to use for application developer, it didn't support some features needed for high-end audio applications, such as non-interleaved audio, or different sample formats support and digital I/O, for which OSS provided a limited support. While OSS is still supported in current kernels, it has been deprecated since the 2.6.0 release in favor of the ALSA subsystem.

2.1.1 Sound system core: ALSA

Nowadays the sound device drivers in Linux kernel are covered by the *Advanced Linux Sound Architecture* (`ALSA`) project [3], which provides both audio and MIDI functionality. It supports a wide range of different sound cards, from consumer ones to professional multichannel ones, including digital I/O or multiple sound cards setups.

Figure 2.1 on the following page describes the basic structure of ALSA system and its dataflow. Unlike previous sound systems, such as OSS, ALSA-native applications are supposed to access the sound driver using the ALSA user-space library (`libasound`), which provides the common API for applications. The library acts as uniforming layer above the hardware cards in use and to abstract changes eventually made in the kernel API to maintain compatibility with existing ALSA-native applications, as it absorbs changes internally while keeping the external API consistent. An in-depth overview of the ALSA API can be found in [6].

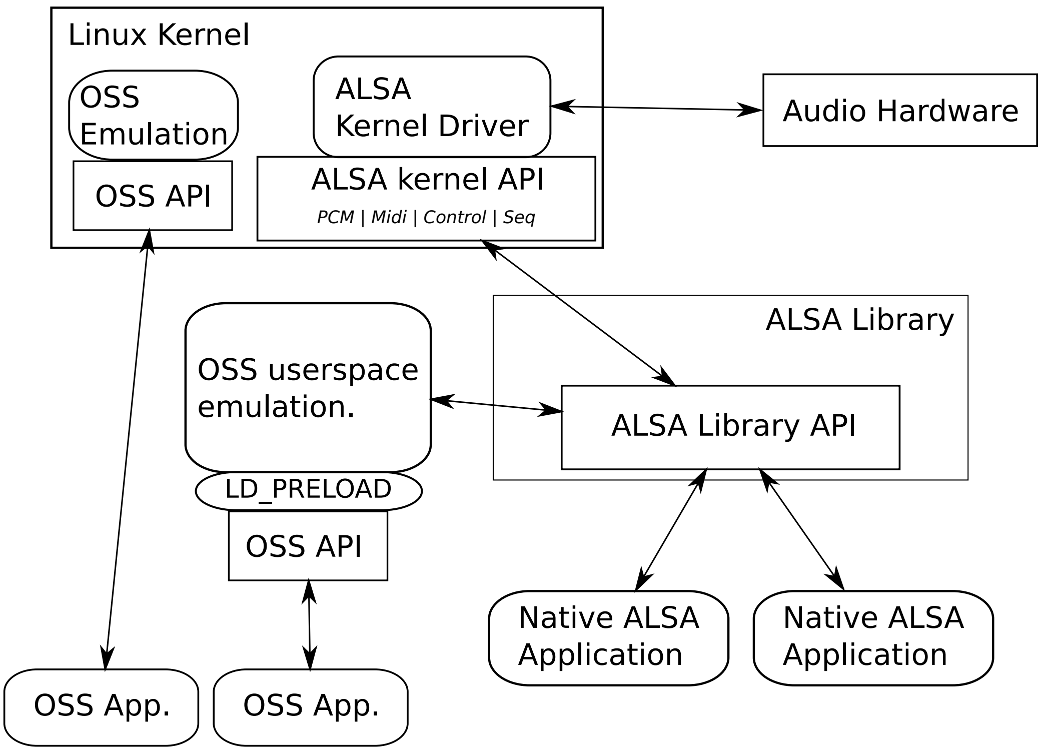

Figure 2.1: An overview of the ALSA architecture

2.1.2 OSS compatibility

The OSS API has been reimplemented to provide backward-compatibility to legacy applications, both in-kernel and in an user space library. As found in Figure 2.1, there are two routes for the OSS emulation. One is through the kernel OSS-emulation modules, and the other is through the OSS-emulation library. In the former route, an add-on kernel module communicates with the OSS applications. The module converts the commands and operates the ALSA core functions. In the other route, the OSS applications run on the top of ALSA library. The ALSA OSS-emulation library works as a wrapper to convert the OSS API to the ALSA API. Since the OSS application accesses the device files directly, the OSS-emulation wrapper needs to be preloaded the OSS applications, so that it replaces the system calls to the sound devices with its own wrapper functions. In this route, the OSS applications can use all functions of ALSA, such as plugins and mmap access, described in section 2.1.3.

2.1.3 Access modes

An application that needs to read/write sound data to/from the sound card can use the ALSA library to open one or more of the PCM supported by the device, and then use the other functions provided to read or write data to the card. The PCM is full-duplex as long as the hardware supports. The ALSA PCM has multiple layers in it. Each sound card may have several PCM devices. Each PCM device has two *streams* (directions), playback and capture, and each PCM stream can have more than one PCM *substreams*. For example, a hardware supporting multi-playback capability has multiple substreams. At each opening of a PCM device, an empty substream is assigned for use. The substream can be also specified explicitly at its opening.

Alternatively, an application can use Memory Mapping to transfer data between user space and kernel space. Memory mapping (`mmap`) is the most efficient method to transfer data between user-space and kernel-space. It must be supported by the hardware and by the ALSA driver, and the supported size is restricted by them. With this access method the application can map the DMA buffer of the driver in user-space, so that the transfer is done by writing audio data to the buffer, avoiding an extra copy. In addition to data, ALSA maps also status and control records, which contains respectively the DMA (also called *hardware pointer*) and the *application pointer*, to allow the application to read and write the current state of the writer, without extra context switching between user and kernel mode. Additionally, the capture buffer is mapped as read-write, allowing the application to "mark" the buffer position it has read to. Due to this requirement, capture and playback buffers are divided to different devices.

2.1.4 Sound Servers

With ALSA drivers, and with OSS drivers as well, accessing the sound card is an exclusive operation. That is, as many cards support only one PCM stream for each playback and capture device, the driver accepts only one process, resulting in only one application accessing audio playback while others are blocked until the first quits. The approach chosen to solve that problem was to introduce an intermediate, broker, server, called *sound server.* A sound server gains access to the sound card, and than provides an *inter-process communication*`IPC` mechanism to allow application to play or capture audio data.

All the mixing between the streams is done by the sound server itself, as well as all the necessary processing like resampling or sample format conversions. On Linux two major sound servers emerged as the *de-facto* standard ones: the first is called Pulseaudio, the other is called `JACK` (*Jack Audio Connection Kit*). Both of them use mmap access method (thus blocking the sound card driver), but they diverge on main focus: while the former is focused to enhance the desktop experience for the average user, providing per-application mixing levels, networking audio with auto discovery of the sound server, power saving, and more, the latter is focused on real time low-latency operation and on tight synchronization between clients, making it suitable for pro audio applications as Digital Audio Workstations, live mixing, music production and creation.

2.2 JACK: Jack Audio Connection Kit

2.2.1 Audio application design problems

Basic constraints that affect audio application design come from the hardware level. After the card initialization, in fact, the audio application must transfer audio data to and from the device at a constant rate, in order to avoid buffer *underruns* or *overruns*. This is referred in general as *xruns* in ALSA and jack terminology, and denote a situation in which the program failed to write (underruns) or read (overruns) data to/from the sound card buffer.

These constraints can be very tight in particular situations. In fact, while normal playback of a recorded track such an MP3 file or an internet stream can make heavy use of buffering to absorb eventual jitter while decoding the stream itself, a software synthesizer driver by MIDI data from an external keyboard can not, as the musician playing the instrument needs to receive the computed sound within a short interval of time. This interval is typically around the 3 milliseconds, and it may become unacceptable for live playing when over 5 or 7 milliseconds, depending on the distance of the musician itself from the loudspeakers.[1]

Applications need to work in respect of these real time constrains, as they can't do their processing neither much in advance, nor they can lag behind. On modern preemptive operating systems applications contend for hardware resources, mainly the CPU, and this contention, among other problems, can lead to miss the real-time constraint in reading or writing audio data with correct time. Moreover, audio applications need non-audio related additional code for their operation, such as I/O handling code, graphical user interface code or networking code. All this additional work can lead to miss the deadline for audio related constraints.

One way to solve these problems is to have a high priority thread dedicated to audio processing that handles all sound card I/O operations, thus decoupling it from other work done within the application itself. Implement-

[1]As the speed of sound in air is about 343 meters per second, a distance of 5 meters from the loudspeaker results in about 14.6 milliseconds of delay, resulting in approximately 3 ms every meter of distance.

ing the audio thread needs very careful planning and design, as it is needed to avoid all operations that can block in a non-deterministic way, such as I/O, memory allocation, network access etc. When the audio thread needs to communicate with other software modules, various non blocking techniques are required, most of which relies on atomic operation and shared memory.

Another issue in audio application design is modularization. Modularization is a key concept in software development, as it's easier to handle complex problem by dividing them first into smaller components used as building blocks. A good example is the text processing tools in UNIX world: every tool is simple, it makes one specific thing but in a very powerful way, and various tools can be combined together to provide complex functionalities.

JACK, which is described in depth in 2.2.2, tries to shift modularization to that extent in the audio world, by providing sample-accurate, low-latency, IPC so that to permit application developers to focus on a particular task. A system of various JACK-capable applications, called JACK clients, can then solve the complex problem, providing a great degree of flexibility.

This, however, brings synchronization costs, as well as those costs from the IPC mechanism plus, as described in 2.2.7 on page 14 it forces application developers to design carefully their applications with respect to the audio processing thread, as lock-free, thread-safe access methods to shared structure are needed to pass data and messages between the main thread and the realtime one.

2.2.2 JACK design

`JACK`[7] (*Jack Audio Connection Kit*) is a low-latency audio server, written for POSIX conforming operating systems such as Linux, and its main focus is to provide an IPC for audio data while maintaining sample synchronization. It was started in 2001 by the Linux audio community as a way to make application focus on their job without the need to worry about mixing, hardware configuration and audio routing. Typically the paradigm used in audio application was to make one application, often called `DAW` (*digital audio workstation*), the *master* application and then have all other applications, such as effects, synthesizers, samplers, tuners, as plugins, i.e. as shared libraries loaded in the main application process space and following a well specified API to communicate with it. While this is certain possible also on POSIX complaint operating systems, it was unfeasible for various problems, mainly in respect of graphics libraries used to paint the graphical user interfaces.

On Unix, in fact, there are at least two major toolkit to write graphical user interfaces, QT and GTK+; since imposing one of the two was considered impractical, as it could led to fragmentation in the user base, a more general approach was needed. As stated above, JACK uses multi-processing and multi-thread approach to decouple all non-audio processing from the main, real-time, audio dataflow. JACK clients are then normal system processes which link to the jack client library. The main disadvantages of the multi process approach are intrinsic in its design, coming from synchronization and scheduling overhead (i.e. context switch). Thus every client has a real-time thread that does computations of audio data while all the other functionalities, as I/O, networking, user interface drawing, are done in separate threads.

Currently there are two main implementations of JACK. The first is the legacy one, which runs on Linux, BSD and MAC OSX operating systems; it was the first to be released and it was written in C99 and referred as *jack1*. The second one, referred as *jackdmp* or *jack2*, is intended to replace the former in the immediate future and it is a complete rewrite in the C++ language; it provides many optimizations in the graph handling and client scheduling, as well as more supported operating systems (it adds support for Windows and Solaris). This work has been focused on JACK version 2, as it represents the future of the JACK development.

This that follows is a brief explanation of the design of the JACK architecture, more details can be found in [14, 13].

2.2.3 Ports

JACK client library takes care of the multi-threading part and of all communications from the client to the server and other clients. Instead of writing and reading audio data directly from the sound card using ALSA API or

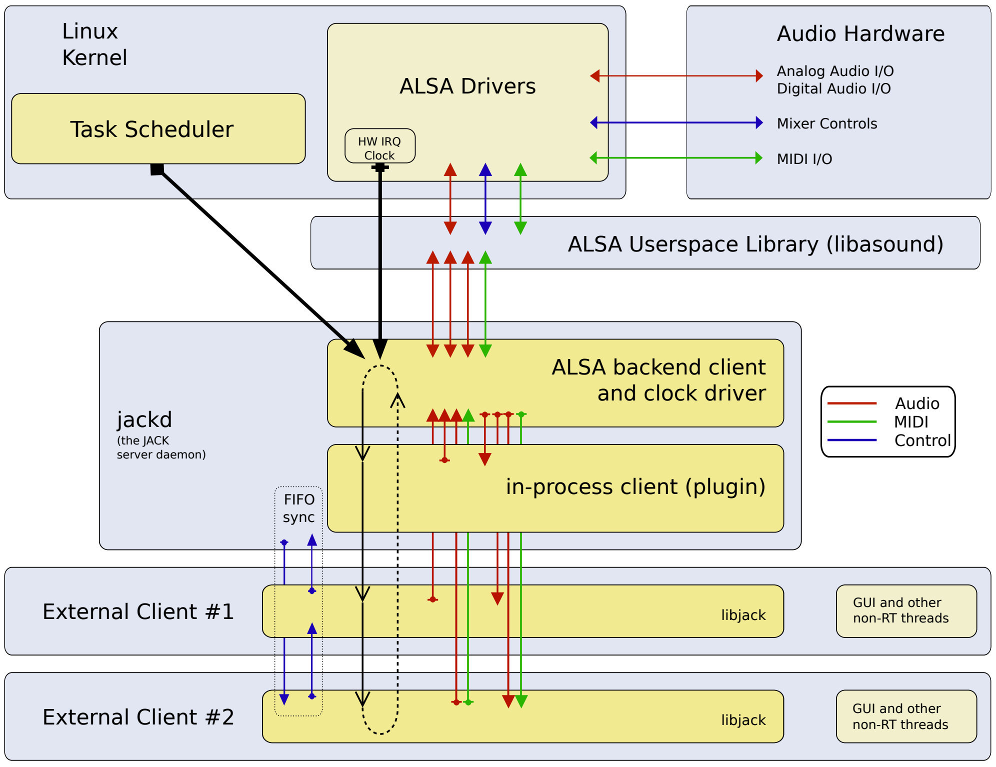

Figure 2.2: An overview of the JACK design

OSS API, a JACK client registers with the server, using JACK own API, a certain number of so called *ports*, and writes or reads audio data from them. Ports size is fixed and dependent from the sample rate and buffer size, which are configured on server startup. Each port is implemented as a buffer in shared memory to be shared between clients and the server, creating a “one writer, multiple reader” scenario, as only one client “owns” the port and thus is allowed to write to it, while other clients can read from it. This avoid the need of further synchronization on port, as the data flow model (further explained in section 2.2.5) already provides the synchronization needed for the reader to find data ready to be read on input ports. This scenario also provides a *zero-copy* semantics on many common simple graph scenarios, and minimal copying on complex one.

2.2.4 Audio Driver

All the synchronization is controlled by a JACK driver which interacts with the hardware, waking the server at regular intervals determined by the buffer size. On Linux, JACK uses mmap-based access with the ALSA driver, and it adopts a double buffering technique: it duplicates the buffer size in two periods, so while the hardware is doing playback on the first half of the buffer, the server writes the seconds half. This fixes the total latency to the size of the buffer, but the jack server can react to input in half of the latency. The basic requirement for the system proper operating is that the server (and all the graph) needs to do all the processing between two consecutive hardware interrupts. This must take in account all the time needed for server/client communications, synchronization, audio data transfer, actual processing of audio data and scheduling latency.

2.2.5 Graph and data flow model

Since ports can be connected to other ports, and since hardware capture and playback buffer are presented as ordinary client ports that do not differ from other clients’ ports, a port-connections paradigm, which can be seen as a directed graph, is obtained. In the graph, ports are nodes and connections are arcs. The data flow model used in the jack graph is an abstract and general representation of how data flows in the system, and generate *data precedence* between clients. In fact, a client cannot execute its processing before data on its input ports is ready: thus a node in the dataflow graph becomes runnable when all inputs port are available.

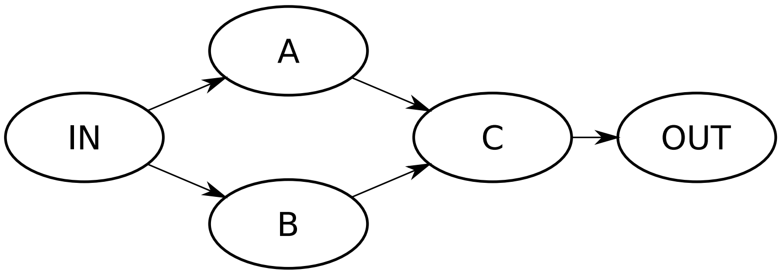

Figure 2.3: A simple client graph. Clients IN and OUT represent physical ports as exposed by the jack driver. Clients A and B can be run in parallel as soon as data is available on inputs, but client C must wait for both A and B to become runnable.

Figure 2.3 contains a simple example of a jack client graph. Each client then use an *activation counter* to count the number of input clients which it depends on; this state is updated every time a connection is made or removed, and it's reset at the begin of every cycle. During normal server cycle processing activation is transferred from client to client as they execute, effectively synchronizing the client execution order on data dependencies. At each cycle clients that depend on input driver ports and clients without any dependency have to be activated first. Then, at the end of its processing, the client decrements the activation counter of all its connected output, so that the last finishing client resumes the following clients.

This synchronization mechanism is implemented using various technologies that depend on the operating system jack is compiled on. Currently on Linux it is using `FIFO` objects on `tmpfs` - a shared memory mounted filesystem which expands on usage - to avoid issuing disk writes for FIFO creation or deletion. The server and client libraries keep those FIFO objects opened upon connections and disconnections, and update the FIFO list[2] that the

[2]The global status of FIFO opened and connections between them, actually represent

client needs to write activations messages to upon graph changes.

2.2.6 Concurrent graph and shared objects management

It appears obvious that, in a multi-process environment, a shared object like the jack graph must be protected from multiple access by concurrent processes. In classic lock-based programming this is usually done using semaphores for mutual exclusion, so that operations appear as *atomic*. The client graph needs to be locked each time a server updates operation access it. When the real-time audio thread runs, it also accesses the client graph to check the connection status and to get the list of the clients to signal upon ending its processing. If the graph has been locked for an update operation, the realtime thread can be blocked for a indefinite, non deterministic amount of time, waiting for the lock to be released by a lower priority thread. This problem is identified as *priority inversion*.

To avoid this situation, in *jack1* implementation the RT threads never block on graph access, but instead generate "blank" audio as output, and skip the cycle, resulting in an interruption in the output audio stream. On the contrary, in *jack2* implementation the graph management has been reworked to use lock-free principles, removing all locks and allowing graph state changes (adding or removal of clients, ports, connections) to be done without interrupting the audio stream. This is achieved by serializing all update operations through the server and by allowing only one specific server thread to update the graph status. Two different graph states are used, the *current* and the *next* states. The current state is what is seen by RT and non RT threads during the cycle, and it's never updated during the cycle itself, so that no locking is needed by the client threads to access it. The next state is what get updated, and it's switched using the `CAS` instruction by the RT audio server thread at the begin of the cycle. `CAS` (Compare And Swap) is the basic operation for lock-free programming: it compares the content of a memory address with an expected value and, if it successes, it replaces

the graph itself.

the content with a new value. Despite this lock-free access method, non-RT threads that want to update the next status are synchronized in a model similar to the mutex-lock/mutex-unlock pair.

2.2.7 A brief overview of the JACK API

As stated before, JACK is composed by multiple parts: it's an API for writing audio programs, an implementation of that API (actually two different implementations, as stated above), and a sound server. The jack API is structured to be *pull-based*, in contrast of the *push-based* mode enforced by both ALSA and OSS APIs. The main difference between the two models, while both of them have their strength and weakness, is that the pull based approach is based on *callbacks*.

In JACK, in fact, a client registers its callbacks for every event it needs to listen to, then the server calls [3] the right callback when an event occurs. The `JackProcessCallback` callback is one of the few mandatory callbacks the client has to register, and it's the one in which the client performs its calculation on input data and produces its output audio data. Apart from the `JackProcessCallback` callback, which is called in the realtime audio thread for obvious reasons, all other callbacks are called asynchronously in a specialized non-realtime thread, to avoid notifications to interfere with audio processing. Finally, all GUI or I/O related work is done in another thread running with no special scheduling privilege.

The main API entrance point is the `jack/jack.h` file, which define almost all the provided functions.

From the developer point of view the API masks all the thread-related complexity. The developer needs only to use a non-blocking paradigm when passing data back and forth to the realtime audio thread, of which he or she needs to write only the body that actually does the audio work.

If it's necessary to save data to disk or to the network, a special thread-safe, non blocking ringbuffer API is provided within the jack API itself; the

[3] Actually, the server uses the IPC mechanism to notify the JACK client library, then the callback is called by the library in the application process space.

key attribute of a ringbuffer is that it can be safely accessed by two threads simultaneously – one reading from the buffer and the other writing to it – without using any synchronization or mutual exclusion primitives, and thus the possibility that the realtime thread is blocked waiting the non-realtime thread is avoided.

As can be seen in the example client (ref. 5.1 on page 92), the API is based on registering callbacks, then waiting for the events. Some commonly used jack API callbacks are:

- `JackProcessCallback`: called by the engine for the client to produce audio data. The *process* event is periodic and the respective callback is called once per server period;
- `JackGraphOrderCallback`: called by the engine when the graph has been reordered (i.e. a connection/disconnection event as well as client arrival or removal);
- `JackXRunCallback`: called by the engine when an overrun or an underrun has been detected; xruns can be reported by the ALSA driver or can be detected by the server if the graph has not been processed entirely;
- `JackBufferSizeCallback`: called by the engine when a client reconfigures the server for different buffer size (and thus latencies);
- `JackPortRegistrationCallback`: called by the engine when a port is registered or unregistered;
- `JackPortConnectCallback`: called whenever a port is connected to or disconnected from another port;
- `JackClientRegistrationCallback`: called upon registering or unregistering of a client;
- `JackShutdownCallback`: called when the server shutdown to notify the client that processing will stop ;

A common pattern for jack client is thus[4]:

```
#include <jack/jack.h>
int myprocess(nframes) {
    in_buf = jack_port_get_buffer("in");
    out_buf = jack_port_get_buffer("out");
    // process
}

jack_client_open();
jack_port_register("in");
jack_port_register("out");
jack_set_process_callback(myprocess)
// register other callbacks
jack_client_activate();

//wait for shutdown event or kill signal
while (true)
  sleep(1);
```

Apart from functions and callbacks for core operations, the JACK API provides utilities functions for various tasks, such as thread managing, non-blocking ringbuffer operations, JACK transport control, MIDI event handling, internal client loading/unloading, and, for JACK server, a control API to control the server operations.

- `jack/thread.h`: Utilities functions for thread handling. Since JACK is a multi-platform software, this series of functions masks all the operating system dependent details while creating threads, in particular for creating additional real-time threads a JACK client needs, as well as querying priorities or destroying previously created threads.

- `jack/intclient.h`: Function for loading, unloading and querying information about internal clients. Internal clients are loaded by the server as shared libraries in its process space, thus not requiring extra

[4]in a pseudo-code syntax, a complete client program is shown in 5.1 on page 92.

context switch when the server schedules them, but a bug in an internal client can possibly affect whole server operation.

- `jack/transport.h`: This header file defines all those functions needed for controlling and querying the JACK transport and timebase. They are used by clients for forward, rewind or looping during playback, synchronizing with other applications. They are commonly used by DAWs and sequencer to cooperate and to stay in sync with the playback or recording.
- `jack/ringbuffer.h`: A set of library functions to make lock-free ringbuffers available to JACK clients. As stated before, a ringbuffer is needed when a client wants to record to disk the audio processed in the realtime thread, to decouple I/O from the realtime processing, in order to avoid nondeterministic blocking of the realtime thread. The key attribute of the ringbuffer data structure here defined is that it can be safely accessed by two threads simultaneously, one reading from the buffer and the other writing to it, without the need of synchronization or mutual exclusion primitives. However, this data structure is safe only if a single reader and a single writer are using it concurrently.
- `jack/midiport.h`: Contains functions to work with midi event and to write and read MIDI data. This functions normalize MIDI events, and ensure the MIDI data represent a complete event upon read.

2.3 Theory of real-time systems

This section briefly introduces some of the main real-time scheduling theory results and concepts used by components, as the *Adaptive Quality of Service Architecture* (`AQuoSA`) framework described in section 2.4.3 on page 29.

2.3.1 Real-time Systems

A real-time system can be simply defined as a set of activities with timing requirements, that is a computer system which correctness depends both on the correctness of the computation results as well as on the time at which those results are produced and presented as system output; they have to interact directly with the external environment, which directly imposes its rigid requirements on the systems timing behavior. These requirements are usually stated as constraints on response time or worst case computation time (or even both) and it is this time-consciousness aspect of real-time systems that distinguishes them from traditional computing systems and makes the performance metrics significantly different, and often incompatible.

In traditional computations system, scheduling and resource management algorithms aim at maximizing the throughput, the utilization and the fairness, so that each application is guaranteed to progress and all the system resources are exploited to their maximum capabilities. In a real-time system those algorithms need to focus on the achievement of definitely different objectives, such as the perfect predictability (for hard real-time systems) or the percentage of missed deadlines minimization (for soft real-time systems). A very important distinction is usually drawn between hard and soft real-time systems. While the former completely fails its objective if the time guaranteed behavior is not honored, even only one single time, the latter can tolerate some guarantees miss to happen and it considers such events only as temporary failures, so there is no need to abort or restart the whole system. Such a difference comes from the fact that hard real-time systems handle, typically, critical activities, as nuclear power plants or flight control systems could be, which, in order to avoid total failure in the system itself, need to respect their environmental-driven time constraints. Soft real-time systems, on the contrary, can be considered instead non critical systems in which, while it's still required for them to finish *in time*, a deadline miss is perceived by the user just as a reduced *Quality of Service* (QoS). From this brief description can be evinced that audio applications, such as JACK, fall in the soft-realtime category, that is a deadline miss can bring to a skipped cycle and a click in the produced audio that, while the user can perceive as a degradation of the sound quality, the system is designed to handle and to recover from.

In order to ensure the meeting of all the deadlines by all the involved activities in all workload conditions for hard real-time systems, off-line feasibility and schedulability analysis are performed with all the most pessimistic assumptions adopted. That is, since no conflicts with other tasks are allowed to happen, each task gets statically allocated all the resources it needs to run to completion. Moreover a precise analysis of how tasks cooperate and access to shared resources is done to avoid unpredictable blockage or *priority inversion* situations. This to enforce the system to be robust in tolerating even peak load situations, although it can cause an enormous waste of system resources: dealing with high-criticality hard real-time systems, high predictability can be achieved at the price of low efficiency, increasing the overall cost of the system.

While dealing with soft real-time systems, instead, approaches like those used with hard realtime systems should be avoided, as they waste resources and lower the total efficiency of the system itself. Furthermore in many soft real-time scenarios the hard real-time approach is extremely difficult to adopt, since many times, for example while dealing with a system like JACK, in which user input can greatly change the computation times of synthesizers, calculating the worst-case execution time (`WCET`) is unfeasible.

Generally, in contrast with hard real-time systems, soft-real time systems are implemented on top of general purpose operating systems[5], such as Linux, as to take advantage of existing libraries, tools and applications. Main desirable features of a soft real-time system are, among others:

- maximize the utilization of system resources
- remove the need to have precise off-line information about tasks
- graceful degradation of performance in case of system overload
- provide isolation and guarantees to real time tasks from other real time tasks

[5]A computer system where it is not possible to a-priori know how many applications will be executed, what computation time and periodicity characteristics they will have and the thing that matter at most is the QoS level provided to each of these application, that can be soft real-time, hard real-time or even non real-time, is also often referred, in real-time systems literature, as an Open System.

- support coexistence of real-time and normal tasks

2.3.2 Real-time task model

In real-time theory each application is called a task (τ_i, the i-th task), and each task is denoted by a stream of jobs $J_{i,j}(j = 1, 2, \ldots, n)$, each characterized by:

- an execution time $c_{i,j}$
- an absolute arrival time $a_{i,j}$
- an absolute finishing time $f_{i,j}$
- an absolute deadline $d_{i,j}$
- a relative deadline[6] $D_{i,j} = d_{i,j} - a_{i,j}$

Each task than has a *Worst Case Execution Time* (WCET) $C_i = max\{c_{i_j}\}$ and a *Minimum Inter-arrival Time* (MIT) of consecutive jobs $T_i = min\{a_{i,j+1} - a_{i,j}\}\ \forall j = 1, 2, \ldots, n$. A task is considered to be periodic if $a_{i,j+1} = a_{i,j} + T_i$ for any job $J_{i,j}$. Finally, the task utilization can be defined as $U_i = \frac{C_i}{T_i}$.

For hard and soft realtime systems it's obvious that, for each task τ_i, $f_{i,j} \leq d_{i,j}$ must hold for any job $J_{i,j}$, otherwise the deadline miss can bring to system failure (in case of hard real-time systems), or degraded QoS (for soft real-time systems).

2.3.3 Scheduling Algorithms

Real-time scheduling algorithms can be divided in two main categories, respectively *static* (or off-line) and *dynamic*. In the former class all scheduling decisions are performed at compile time based on prior knowledge of every parameter describing the task set, resulting in the lowest run-time scheduling overhead. In dynamic scheduling algorithms decisions are instead taken at run-time, so the scheduler picks the task to run among the tasks ready to be run, providing more flexibility but higher overhead.

[6]Sometimes referred to as minimum response time.

Both classes can be further divided in preemptive and non-preemptive algorithms: preemptive schedulers permit a higher priority that becomes ready task to interrupt the running task, while using non-preemptive schedulers tasks cannot be interrupted once they are scheduled.

Another further division in classifying is between static priority algorithms and dynamic priority algorithms, with respect to how priorities are assigned and if they are ever changed. The former algorithms keep priority unchanged during system evolution, while the latter change priorities as required by the algorithm in use.

An example of an on-line, static scheduling algorithm is the *Rate Monotonic* (`RM`) algorithm, in which each task τ_i is given a priority p_i inversely proportional to its period of activation: $p_i = \frac{1}{T_i}$.

Another example can be the on-line dynamic scheduling algorithm called *Earliest Deadline First* (`EDF`) in which at every instant t the task priority is given as $p_i(t) = \frac{1}{d_i(t)}$, which means that at every instant the task whose deadline is more imminent has the maximum priority in the task set. Both algorithms are described in depth in [9].

With the RM scheduler it has been proved that, in a task set τ with n tasks, it is guaranteed to exist a feasible scheduling if [9]:

$$U = \sum_{i=0}^{n} U_i = \sum_{i=0}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$$

while, by means of EDF scheduler, if:

$$U = \sum_{i=0}^{n} U_i = \sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1$$

As thus can be seen the EDF scheduler, exploiting the dynamic reassignment of priorities, can achieve a 100% of resource reservations, while RM, using static priorities, can only guarantee 0,8284 with $n = 2$, and, in general:

$$\lim_{n \to \infty} n \cdot (2^{\frac{1}{n}} - 1) = \ln 2 \approx 0.693147\ldots$$

which limits the total utilization for which it's guaranteed that a feasible schedule exists using RM, tough a schedule with higher utilization factor can still be found as the condition is only proved sufficient.

While this algorithms and results where developed and obtained thinking mostly to hard real-time systems, they can be extended to soft real-time systems with the aid of server based algorithms described in section 2.3.4.

2.3.4 Server based scheduling

Server based scheduling algorithms are based on the key concept of *server*, which has two parameters:

- Q_i (or C_i^s), the budget or capacity
- P_i (or T_i^s), the period

In this category of algorithms, servers are the *scheduling entities*, so they have an assigned (dynamic or static) priority and are inserted in the system ready queue. This queue can be handled with any scheduling algorithm, such as RM or EDF. Each server *serves* a task or a group of tasks, each one hard or soft as well as either period, sporadic or aperiodic. While a task belonging to a server is executing, the server budget is decreased and periodically recharged, so that its tasks are guaranteed that they will execute for Q_i units of time each P_i period, depending on which type of server scheduling is in use.

There is a large variety of well known server algorithms in real-time literature, such as *Polling Server* (`PS`), *Deferrable Server* (`DS`), *Sporadic Server* (`SS`), *Constant Bandwidth Server* (`CBS`), *Total Bandwidth Server* (`TBS`) and others, and most of them can be used with, or slightly adapted to, both RM and EDF scheduling.

One of the most used server, better described in section 2.3.5 on the next page, is the Constant Bandwidth Server[1], which is a common choice in Resource Reservation frameworks.

2.3.5 Constant Bandwidth Server

The Constant Bandwidth Server [1] algorithm is one of the server based algorithm that have been just introduced.

Each CBS server is characterized by the two classical parameters, Q_i, the maximum budget, and P_i, the period, and also with two additional ones:

- q_i (or better $q_i(t)$), the server budget at exactly the time instant t
- δ_i (or better $\delta_i(t)$), the current server deadline at exactly the time instant t

Server budget and period are often referred as *bandwidth* $B_i = \frac{Q_i}{P_i}$.

The fundamental property of the CBS algorithm is that, defined the fraction of processor time assigned to the i-th server as:

$$U_i^s = \frac{Q_i}{P_i}$$

the total usage of tasks belonging to server S_i is guaranteed not to be greater than U_i^s, independently from overload situations or misbehavior. The CBS algorithm can be described with the following rules:

1. When a new server S_i is created, its parameters are set so that $q_i = Q_i$ and $\delta_i = 0$
2. **rule A**: if the server is idle and a new job $J_{i,j}$ of the task τ_i (associated with S_i) arrives, the server checks if it can handle with the current (q_i, δ_i), evaluating the following expression:

   $$q_i < (\delta_i - a_{i,j}) \cdot U_i^s$$

   if true, nothing has to be done, else a new pair (q_i, δ_i) is generated as:

   $$q_i = Q_i$$

   $$\delta_i = \delta_i + P_i$$

3. **rule B**: when a job of a task belonging to S_i executes for a Δt time units, the current budget is decreased as:

$$q_i = q_i - \Delta t$$

4. **rule C**: when the current budget of a server S_i reaches 0 the server, and thus all the associated tasks, are descheduled and a new pair (q_i, δ_i) is generated as:

$$q_i = Q_i$$

$$\delta_i = \delta_i + P_i$$

Should also be noted that:

- A server S_i is said to be **Active** at instant t if it has pending jobs in its internal queue, that is if exists a server job $J_{i,j}$ such that $a_{i,j} < t < fi, j$
- A server S_i is **Idle** if it's not active
- When a new job $J_{i,j}$ of the task τ_i, associated with S_i, arrives and the server is active, it's enqueued by the server in its internal queue of pending jobs, which can be handled with arbitrary scheduling algorithm.
- When a job finishes the first job in the internal pending queue is executed with the remaining budget, according to the rules above.
- At the instant t the job $J_{i,j}$ executing in server S_i is assigned the last generated deadline.

The basic idea behind the CBS algorithm is that when a new job arrives it has a deadline assigned, which is calculated using the server bandwidth, and then inserted in the EDF ready queue. In the moment the job tries to execute more than the assigned server bandwidth, its deadline gets postponed, so that its EDF priority is lowered and other tasks can preempt it.

2.3.6 Resource Reservations

The CBS server algorithm thus works without requiring the WCET and the MIT of the associated task to be known nor estimated in advance while, moreover, a reliable prediction or the exact knowledge of these two parameters can be used to set up the server parameters Q_i and P_i with much more ease, till obtaining the task behaves exactly as an hard real-time one.

One of the most interesting properties of the CBS algorithm is the *Bandwidth isolation property*, that is it can be showed that if

$$\sum_{i=0}^{n} B_i \leq U^{lub}$$

where U^{lub} depends on the global scheduler used and in particular, as said

$$U^{lub} = 1$$

for the EDF scheduler, then each server Si is reserved a fraction $B_i = Q_i/T_i$ of the CPU time regardless of the behavior of other servers tasks. That is, the worst case schedule of a task is not affected by the temporal behavior of the other tasks running in the system.

The CBS algorithm has been expanded to implement both *hard* and *soft* reservations. In the former, when a server exhausts its budget, all its associated tasks are suspended up to the server deadline, then the budget is refilled and the gain chances to be scheduled again. In the latter, tasks are not suspended upon budget exhaustion, but the server deadline is postponed so that the tasks priority decreases and it can be preempted, if some other tasks with shorter deadline are in the ready queue.

There are some algorithm proposed to improve the sharing of the spare bandwidth while using hard reservations. Some of them are `GRUB`, `IRIS` and `SHRUB`. One of the most interesting of these, for this work, is the `SHRUB` (*SHared Reclamation of Unused Bandwidth*) algorithm, which effectively distributes the spare bandwidth to servers using a policy based on weights.

2.3.7 Reclaiming Mechanism: SHRUB

SHRUB is a reclaiming mechanism of the spare system bandwidth, i.e. not associated with any server, based on GRUB[8, 11].

SHRUB is a variant of the GRUB algorithm, which in turn is based on the CBS. In SHRUB each reservation is also assigned a positive weight ω_i , and execution time is reclaimed based on ω_i (the reclaimed time is distributed among active tasks proportionally to the reservation weights).

The main idea behind GRUB and SHRUB is that if $B_{act}(t)$ is the sum of the bandwidths of the reservations active at time t, a fraction $(1B_{act}(t))$ of the CPU time is not used and can be re-distributed among needing reservations.

The re-distribution is performed by acting on the accounting rule used to keep track of the time consumed by each task. In GRUB all the reclaimed bandwidth is greedily assigned to the current executing reservation, and time is accounted to each reservation at a rate that is proportional to the current reserved bandwidth in the system. If $B_{act} < 1$ this is equivalent to temporarily increasing the maximum budget of the currently executing reservation for the current period. In the limit case of a fully utilized system, $B_{act} = 1$ and the execution time is accounted as in the CBS algorithm. In the opposite limit case of only one active reservation, time is accounted at a rate B_i (so, a time Q_i is accounted in a period P_i).

SHRUB, instead, fairly distributes the unused bandwidth among all active reservations, by using the weights. In the two limit cases (fully utilized system and only one reservation), the accounting mechanism for GRUB and SHRUB work in the same way. However, when there are many reservations in the system and there is some spare bandwidth, SHRUB effectively distributes the spare bandwidth to all needing reservations in proportion to their weights. Unlike GRUB, SHRUB uses the weights to assign more spare bandwidth to reservations with higher weights, implementing the equation:

$$B_i = B_i^{'} + \frac{\omega_i}{\sum_j \omega_j}\left(1 - \sum_k B_k^{'}\right)$$

where $B_i^{'} = min\{B_i^{req}, B_i^{min}\}$, B_i^{req} is the requested bandwidth, B_i is the resulting bandwidth after compression.

2.4 Real-time schedulers on Linux

2.4.1 POSIX real time extensions

As Linux is a POSIX-compliant operating system, it implements the POSIX standards, and the scheduler is not an exception. In particular, it implements the realtime scheduling classes that POSIX defines in IEEE Std 1003.1b-1993. There are supported scheduling policies, such as the POSIX required fixed priority (`SCHED_FIFO`), Round-Robin (`SCHED_RR`) and a typical desktop system time-sharing policy (`SCHED_OTHER`), made out of heuristics on tasks being runnable or blocked, with the fixed priority policy (SCHED_FIFO) being the most important one if we are interested in trying to build a real-time systems out of Linux. There are 140 priority levels where the range 0..99 is dedicated to so-called real-time tasks, i.e. the ones with SCHED_FIFO or SCHED_RR policy, and available only to the user in the system with sufficient capabilities (usually only the root user, or, those users and groups the root user authorized to use them). A typical Linux task uses the SCHED_OTHER policy.

The Linux scheduler can described as follows:

- if one or more real-time task (SCHED_RR or SCHED_FIFO) is ready for execution, then the one with the highest priority is run and can only be preempted by another higher priority, SCHED_FIFO task, otherwise it runs undisturbed till completion.

- if SCHED_RR is specified after a time quantum (typically 10 ms) the next task with the same policy and priority (if any) is scheduled, so that all SCHED_RR tasks with the same priority are scheduled Round-Robin.

- if no real-time task is ready for execution, SCHED_OTHER tasks are scheduled with a dynamic priority calculated according to the defined set of heuristics.

Linux, being a general purpose time sharing kernel, has no implementation of either EDF or RM, nor of any other algorithm from classic real-time theory. This is felt by the majority of the community as not being an issue, as rate monotonic, for example, can be implemented out of the priority based SCHED_FIFO scheduling class, without any need for modifications of the kernel itself.

2.4.2 Linux-rt

The patch to the Linux kernel source code known as `PREEMPT_RT`[12] is maintained by a small developer group with the aim of providing the kernel with all the features and the capabilities of both a soft and hard real-time operating system. The patch, basically, modifies a standard Linux kernel so that it could nearly always be preempted, except few critical code regions, and so it can extend the support for priority inheritance to in-kernel locking primitives and move interrupt handling to schedulable threads. A complete coverage of this patchset is out of the scope of this work, thus we give only a brief explanation of the ideas behind its changes to the Linux kernel.

In this patch the interrupt handling[7] has been redesigned to be fully-preemptable, moving all interrupt handling to kernel threads, which are managed by the system scheduler like every other task in the system. This allows a very high priority real-time task to preempt even IRQ handlers, reducing the overall latency.

To achieve full preemption, spinlocks and mutex inside the kernel have been reworked so that the former ones become normal mutexes (thus preemptable) and the latter ones rt-mutexes that implement the *Priority Inheritance Protocol*, to protect all the new-preemptable kernel path from the priority inversion issue.

The resulting patch is very well suited for soft-real time system, as it minimizes the total latency for IRQ handling, which is critical when working

[7]In the main kernel tree, the interrupt handler is separated in two parts, the hard and the soft one. The former is the "classical" interrupt handler that runs with interrupts disabled, while the latter is the *SoftIRQ*, which can be preempted and it's called by the hard part to finish non critical work.

with audio, even if it lacks forms of resource reservations. It is a common choice among users in the Linux audio community: it is, in fact, the recommended kernel patch by the JACK developers and users, as well as other developers from other project in the Linux audio community as a whole.

2.4.3 AQuoSA: Adaptive Quality of Service Architecture

AQuoSA [2, 10] stands for Adaptive Quality of Service Architecture and is a Free Software project developed at the Real-Time Systems Laboratory (RETIS Lab, Scuola Superiore SantAnna, in Pisa) aimed at providing Quality of Service management functionalities to a GNU/Linux system. The project features a flexible, portable, lightweight and open architecture for supporting soft real-time applications with facilities related to timing guarantees and QoS, on the top of a general-purpose operating system as GNU/Linux is.

The basis of the architecture is a patch to the Linux kernel that embeds into it a generic scheduler extension mechanism. The core is a reservation based process scheduler, realized within a set of kernel loadable modules, exploiting the patch provided mechanism, and enforcing the timing behavior according to the implemented soft real-time scheduling policies. A supervisor performs admission control tests, so that adding a new application with its timing guarantees does not affect the behavior of already admitted ones. The AQuoSA frameworks is divided in various components:

- the *Generic Scheduler Patch* (`GSP`): a small patch to the kernel extending the Linux scheduler functionalities by intercepting scheduling events and executing external code from a loadable kernel module;
- the *Kernel Abstraction Layer* (`KAL`): a set of C functions and macros that abstract the additional functionality we require from the kernel (e.g. time measuring, timers setting, task descriptor handling, etc.);
- the *QoS Reservation component*: two kernel module and two user-space library communicating between each other through (two) Linux virtual

device:

  - the resource reservation loadable kernel module (`rresmod`) implements the EDF and the RR scheduling algorithms, making use of both the GSP exported hooks and the KAL facilities;
  - A set of compile-time options can be set up to enable the use of different RR primitives and customize their semantics (e.g. soft or hard reservations or SHRUB);
  - the resource reservation supervisor loadable kernel module (`qresmod`) grants no system overload to occur and enforces system administrator defined policies;
  - the resource reservation library (`qreslib`) provides the API that allow an application to use the resource reservation module supplied facilities;
  - the resource reservation supervisor library (`qsuplib`) provides the API that allow an application to access the supervisor module supplied facilities;

- the *QoS Manager component*: a kernel module and an application library:

  - the QoS manager loadable kernel module (`qmgrmod`) providers kernel-space implementation of prediction algorithms and feedback control;
  - the QoS manager library (`qmgrlib`) provides an API that allows an application to use QoS management functionalities and directly implements the control loop, if the controller and predictor algorithms have been compiled to be in user-space, or, otherwise, redirects all requests to the QoS manager kernel module;

The core mechanism of the QoS Reservation component is implemented in two Linux kernel loadable modules, `rresmod` and `qresmod` and the new scheduling service are offered to applications through two user libraries (qreslib and qsuplib). Communication between the kernel modules and the user (the libraries) level happens through a Linux virtual device using the `ioctl` family of system calls.

Resource Reservation module

The `rresmod` module is responsible for implementing both the scheduling and the resource reservation algorithms, on top of the facilities provided by the hooks of the GSP and the abstractions of the KAL. When the module is loaded in the kernel the GSP hooks are set to their handlers and all the reservation related data structures are initialized.

As for scheduling, the module internally implements an Earliest Deadline First (EDF) queue, since the purpose is to implement, as RR algorithm, the CBS or one of its variants, and affects the behavior of the reserved tasks by simply manipulating the standard Linux kernel ready task queues (*runqueues* or simply `rq`).

Tasks using the reservation mechanisms are forced to run according to EDF and CBS order by assigning them a SCHED_RR policy and a statically real-time priority, while they are forbidden to run (hard reservations) by temporarily removing them from the Linux ready queue.

QoS supervisor component

The QoS supervisor main job is to avoid that a single user, either maliciously or due to programming errors, causes a system overload or even forces tasks of other users to get reduced bandwidth assignments, as well as to enforce maximum bandwidth policies defined by the system administrator for each user and users group.

The main advantages of implementing the supervisor module and library as a separate component is an user level task, provided it has the required privileges, can be utilized to define the rules and the policies, as well as put them in place via the functions of the supervisor user level library, which communicate with its kernel module through a device file and the ioctl system call. As an example a classical UNIX daemon run by the system administrator and reading the policies from a configuration file, living in the standard

`/etc` directory, can be our reservation security manager.

Moreover, if the QoS Manager is also used, the bandwidth requests coming from the task controllers can be accepted, delayed, reshaped or rejected by the QoS Supervisor.

Resource reservation and QoS supervisor libraries

The resource reservation and the QoS supervisor libraries exports the functionality of the resource reservation module and of the QoS supervisor module to user level applications, defining a very simple but appropriate API. All functions can be used by any regular Linux process or thread, provided, in the case of `qsuplib`, the application has enough privileges.

The communication between the libraries and the kernel modules happens, as stated, throughout two virtual device, called `qosres` (major number 240) and `qossup` (major number 242), created by the kernel modules themselves and accessed with the powerful ioctl `Linux` system call, inside the libraries implementation, described in section 2.4.3 on the following page.

QoS Manager component implementation

QoS Manager is responsible for providing the prediction and control techniques usable in order to dynamically adjust a task assigned bandwidth and the algorithm for their implementation can be compiled in both an user level library or a kernel loadable module, although the application is always linked to the QoS Manager library and only interacts with it.

The main difference between kernel and user level implementation is the number of user-to-kernel space switch needed is only one in the former case and two in the latter, and so the possibility to compile the control algorithms directly at kernel level is given in order to reduce to the bare minimum the overhead of the QoS management.

An application that wants to use the QoS Manager has to end each cycle of its periodic work, that is each job, with a call to the library function `qmgr_end_cycle` which, depending on the configured location for the required sub-components, is redirected by the library implementation to either user level code or, as usual, to a ioctl system call invocation on a virtual device.

AQuoSA Application Programmable Interface

The QoS resource reservation library allows applications to take advantage of the RR scheduling services available when the QoS resource reservation kernel module module is loaded.

The main function it provides are:

`qres_init` initializes the QoS resource reservation library;

`qres_cleanup` cleanup resources associated to the QoS resource reservation library

`qres_create_server` create a new server with specified parameters of budgets and period and some flags

`qres_attach_thread` attach a thread (or a process) to an existing server

`qres_detach_thread` detach a thread (or a process) from a server. It may destroy the server if that was the last one attached (also depending on server parameters)

`qres_destroy_server` detach all threads (and processes) from a server and destroy it

`qres_get_params` retrieve the scheduling parameters (budgets and period) of a server

`qres_set_params` change the scheduling parameters (budgets and period) of a server

`qres_get_exec_time` retrieve running time information of a server.

2.5 Linux Control Groups

Control Groups (`cgroups`) provides a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behavior. A particular terminology applies to Linux control groups:

cgroup associates a set of tasks with a set of parameters for one or more subsystems.

subsystem is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a *resource controller* that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem.

hierarchy is a set of cgroups arranged in a tree, so that every task in the system is exactly in one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual filesystem associated with it.

At any one time there may be multiple active hierarchies of task cgroups. Each hierarchy is a partition of all tasks in the system. User level code may create and destroy cgroups by name in an instance of the cgroup virtual file system, may specify and query to which cgroup a task is assigned, and may list the task PIDs assigned to a cgroup. Those creations and assignments only affect the hierarchy associated with that instance of the cgroup file system.

On their own, the only use for control groups is for simple job tracking. The intention is that other subsystems hook into the generic cgroup support to provide new attributes for cgroups, such as accounting/limiting the resources which processes in a cgroup can access.

There are multiple efforts to provide process aggregations in the Linux kernel, mainly for resource tracking purposes. Such efforts include `cpusets`, `CKRM/ResGroups`, `UserBeanCounters`, and virtual server namespaces. These all require the basic notion of a grouping/partitioning of processes, with newly forked processes ending in the same group (cgroup) as their parent process.

The kernel cgroup patch provides the minimum essential kernel mechanisms required to efficiently implement such groups. It has minimal impact on the system fast paths, and provides hooks for specific subsystems such as `cpusets` to provide additional behavior as desired.

Multiple hierarchy support is provided to allow for situations where the division of tasks into cgroups is distinctly different for different subsystems - having parallel hierarchies allows each hierarchy to be a natural division of tasks, without having to handle complex combinations of tasks that would be present if several unrelated subsystems needed to be forced into the same tree of cgroups.

In addition a new file system, of type *cgroup*, may be mounted to enable browsing and modifying the cgroups presently known to the kernel. When mounting a cgroup hierarchy, you may specify a comma-separated list of subsystems to mount as the filesystem mount options. By default, mounting the cgroup filesystem attempts to mount a hierarchy containing all registered subsystems.

If an active hierarchy with exactly the same set of subsystems already exists, it will be reused for the new mount. If no hierarchy matches exist, and any of the requested subsystems are in use in an existing hierarchy, the mount will fail. Otherwise, a new hierarchy is activated, associated with the requested subsystems.

When a cgroup filesystem is unmounted, if there are any child cgroups created below the top-level cgroup, that hierarchy will remain active even though unmounted; if there are no child cgroups then the hierarchy will be deactivated.

No new system calls are added for cgroups - all support for querying and modifying cgroups is via this cgroup file system.

2.5.1 CPU Accounting Controller

The CPU accounting controller is used to group tasks using cgroups and account the CPU usage of these groups of tasks.

The CPU accounting controller supports multi-hierarchy groups. An accounting group accumulates the CPU usage of all of its child groups and the tasks directly present in its group. Accounting groups can be created by first mounting the cgroup filesystem.

Upon mounting, the initial or the parent accounting group becomes visible at the mounting point chosen, and this group initially includes all the tasks in the system. A special file `tasks` lists the tasks in this cgroup. The file `cpuacct.usage` gives the CPU time (in nanoseconds) obtained by this group which is essentially the CPU time obtained by all the tasks in the system. New accounting groups can be created under the parent root group.

The `cpuacct.stat` file lists a few statistics which further divide the CPU time obtained by the cgroup into user and system times. Currently the following statistics are supported:

user † time spent by tasks of the cgroup in user mode.

system † time spent by tasks of the cgroup in kernel mode.

`cpuacct` controller uses `percpu_counter` interface to collect user and system times. It is thus possible to read slightly outdated values for user and system times due to the batch processing nature of `percpu_counter`.

2.5.2 CPU Throttling in the Linux kernel

The current kernel code already embeds a rough mechanism, known as *CPU Throttling*, that has been designed for the purpose of limiting the maximum CPU time that may be consumed by individual activities on the system. The mechanism used to be available on older kernel releases only for real-time scheduling policies for stability purposes. Namely, it was designed so as to prevent real-time tasks to starve the entire system forever, for example as a result of a programming bug while developing real-time applications. The original mechanism only allowed the overall time consumed by real-time tasks (no matter what priority or exact policy they had) to overcome a statically configured threshold, within a time-frame of one second. This used to be specified in terms of the maximum amount of time (a.k.a., throttling

†user and system are in `USER_HZ` units

runtime, expressed in microseconds, which corresponds to the well-known concept of *budget*, in the real-time literature), defaulting to 950 ms, available to real-time tasks within each second (a.k.a., throttling *period*).

Only recently, core kernel developers recognized the usefulness of such mechanism for purposes related to temporal isolation of tasks among each other (as opposed to being used solely between the group of real-time tasks and the one best-effort tasks). Therefore, a well-defined interface has been defined in order to support throttling both at the task/thread level, and at the task group level, by taking advantage of the cgroup virtual filesystem.

Thanks to this framework, the POSIX semantics of real-time task scheduling in Linux has been recently modified, adding support for group scheduling, following the general trend of adding container support to all the subsystems of the kernel.

With such a framework, whenever a processor becomes available, the scheduler selects the highest priority task in the system that belongs to any group that has some execution budget available, then the execution time for which each task is scheduled is subtracted from the budget of all the groups it hierarchically belongs to. The budget assigned initially to a group is the same on all the processors of the system, and is selected by the user.

The budget limitation is enforced hierarchically, in the sense that, for a task to be scheduled, all the groups containing it, from its parent to the root group, must have some budget left. In the case of a per-task-group throttling configuration, a special file entry inside a task- group folder named `cpu_rt_runtime_us` allows for configuring the maximum budget consumable by the entire sub-tree of tasks having that folder as ancestor. From the real-time guarantees perspective, the throttling mechanism is well suited to prevent real-time tasks from monopolizing the CPU due to unexpected overruns. This basically means that the time granularities the mechanism is based on are quite long, in the order of 1s-10s, and that it is not foreseen to have too many competing groups.

The current throttling mechanism has two major drawbacks:

1. from a real-time theoretical perspective, it works basically like the well-known Deferrable Scheduler algorithm [15], in the literature of Real-

Time scheduling (at least looking at what happens on a single-CPU system): such scheme has been overcome by a number of other schemes that perform much better;

2. the current implementation enforces temporal encapsulation on the basis of a common time granularity for all the tasks in the system, that is one second; this makes it impossible to guarantee good performance on service components that need to exhibit sub-second activation and response times.

2.5.3 Hierarchical multiprocessor CPU reservations

As stated in 2.5.2 on page 36Linux support rt-throttling in the cgroup cpu controller. Each cgroup thus has two files, `cpu.rt_period_us` (P_i) and `cpu.rt_runtime_us` (Q_i) which defines respectively the period and the maximum runtime of realtime tasks belonging to the group itself. This values limits the maximum runtime for SCHED_FIFO and SCHED_RR tasks to Q_i units of time every P_i units. This, however, only *limits* the CPU time consumed by tasks, it does not *enforce* its provisioning, nor it has a form of admission control, that is the total sum of the utilizations $U_i = \frac{Q_i}{P_i}$ can be greater than 1.

A patch to this throttling mechanism have been proposed[4] to use EDF to schedule groups, thus enforcing and guaranteeing groups the assigned bandwidth. With this patchset the hierarchical structure of the cgroup filesystem is used to create a hierarchy of groups in which:

- Every group is a child, direct or indirect, of the `root` group.
- For each level of the hierarchy must hold that:

$$\sum_{i=0}^{n} B_{G_i} + \sum_{j=0}^{m} B_{\tau_j} \leq 1$$

  where $B_{G_i} = \frac{Q_{G_i}}{P_{G_i}}$ is the bandwidth of the i-th group and B_{τ_j} is the assigned bandwidth for tasks belonging to groups of the upper level.

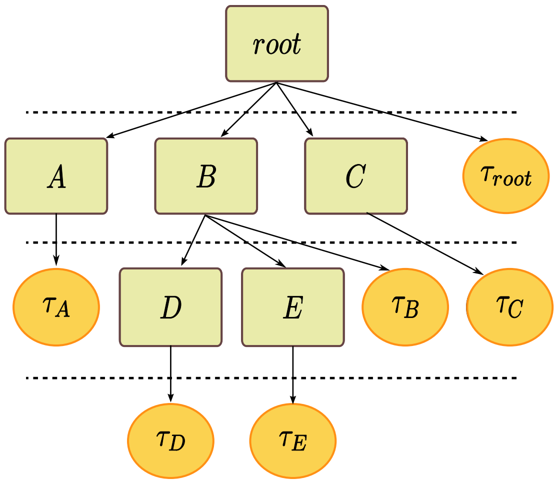

Figure 2.4: *A possible hierarchy of groups and tasks. Squares represent groups, while circle represent tasks. Arrows show parent to child relations. For each horizontal level the total bandwidth (sum of the bandwidths assigned to groups and tasks) must be lesser or equal to the assigned bandwidth of the parent.*

- For each group G_i with n children must be true that:

  $$\sum_{j=0}^{n} B_{C_j} + B_{\tau_G} \leq B_G$$

  where B_{τ_G} is the bandwidth of its tasks, B_{C_j} the bandwidth of the $j-th$ child group and B_G its own bandwidth, that is the sum of the bandwidths assigned to its children group and to its tasks must be lesser than or equal to its assigned bandwidth.

Figure 3.1 on page 41 show a possible hierarchy. Note as individual tasks within a group does not have the period/runtime pair each but they share a common period and runtime setting.

While for admission control purposes the hierarchy is inspected as stated above, when the scheduling decision has to be take all tasks and groups are taken as they were on a single level. This creates a two level scheduler, in which groups are scheduled using EDF while tasks inside the group are scheduled using round robin.

For each scheduling group, thus, the scheduler exposes four files:

`cpu.rt_period_us` the period (in microseconds) of the group

`cpu.rt_runtime_us` the runtime of the group

`cpu.rt_task_period_us` the period for its associated tasks

`cpu.rt_task_runtime_us` the runtime for its associated tasks

For example, to assign $Q = 20$ms and $P = 100$ms to a group of 3 tasks (with PID respectively of 1000, 1001 and 1003) the following command can be issued:

```
$ mount -t cgroups none /cgroup -o cpu
$ cd /cgroups
$ mkdir group0
$ echo 1000000 > group0/cpu.rt_period_us
$ echo 200000 > group0/cpu.rt_runtime_us
$ echo 1000000 > group0/cpu.rt_task_period_us
$ echo 200000 > group0/cpu.rt_task_runtime_us
$ echo 1000 > group0/tasks
$ echo 1001 > group0/tasks
$ echo 1002 > group0/tasks
```

With the support of this patchset (for the RT part) and of the libcgroup library and tools, very complex scenarios can be defined, with per-user or per-user-groups policies on realtime bandwidth, “normal” tasks CPU share, network utilization, device access, memory consumption and more.

Chapter 3

Architecture Design

In this chapter the whole architecture design will be discussed, as well as all the modifications done to the various software components and new software and libraries that has been written during this work, in particular the modifications needed to the JACK server and client libraries to make them support feedback scheduling using the AQuoSA API, but also all the new software, such as the `libgettid` library, the `dnl` JACK client used to load the JACK graph, the `rt-app` periodic application used as a disturb during simulations, and finally a port of the AQuoSA userspace library `qreslib` to the Hierarchical Multiprocessor CPU reservations described in section 2.5.3 on page 38.

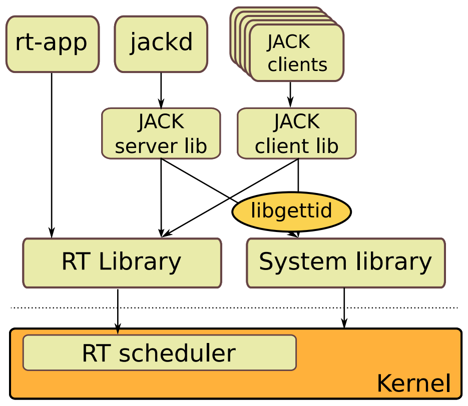

Figure 3.1: *An overview of the designed architecture.*

As said, one of the main goal while extending the JACK server to support resource reservations was to maintain both API and ABI compatibility. API, which stands for *Application Programming Interface* and which is of the interface provided by the library to applications that use it, was not extended, reduced, modified in any way, nor any semantic change was made to any of API-exported functions, resulting in complete API level compatibility with previous and official versions.

API (and ABI) compatibility was mandatory to this work: an API or ABI breakage would have meant that every single preexisting JACK client would have had to be modified in order to support these eventual changes, thus reducing the availability of this work for people using any standard Linux distribution and increasing all the efforts while developing (and hopefully distributing) this work. ABI, which stands for *Application Binary Interface*, covers details such as data type, size and alignment, as well as calling conventions, etc. Thus, in order to reduce chances of a possible API/ABI breakage, particular care has been taken in order to avoid, when feasible, any modification to exported data or functions.

All the modifications done to the JACK server and to the JACK client library are compatible with existing clients, so that the AQuoSA enabled JACK server and libraries can be used as a drop-in replacement for the official package, provided that its requirement (AQuoSA, `libgettid`) are installed in the system path.

3.1 libgettid

The problem arose from the need to convert POSIX threading `pthread` thread identifiers (i.e. of type `pthread_t`) to a Linux-specific thread identifier, which is a generalization of the POSIX process identifier. While the *Linux thread identifier* (`TID` for short) is a system-wide unique tag which unambiguously distinguishes the thread, the pthread library identifiers returned by the `pthread_create()` or `pthread_self()` library functions are only guaranteed to be unique within a process. Plus, thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is non-portable and can lead to unspecified results.

The JACK thread API, in file `jack/jack.h`, is specified using pthreads identifiers, i.e. every utility call that manipulates client threads takes a pthread_t pointer as the identifier for the thread to work on. As an example, the following function:

```
int jack_acquire_real_time_scheduling (pthread_t thread,
                                       int priority);
```

which changes the scheduling policy of the thread to match the current realtime scheduling policy the JACK server uses (if any), identifies the thread using the pthread_t identifier. Similarly, all the JACK server internals that handle threads and threads scheduling use the same identifiers. This is possible since all the threading related work is done in the server or in the client process space, as both client and server link the respective library.

The AQuoSA `qreslib` API, as said, uses instead the Linux TID when referring to threads. This is necessary as the API has been designed to handle threads in the system. As an example, the API definition of the `qres_attach_thread` function:

```
qos_rv qres_attach_thread(qres_sid_t server_id,
                          pid_t pid,
                          tid_t tid);
```

which is used to attach a thread to an AQuoSA server, identifies the thread to operate on by its PID and TID.

The problem was that there's no common way to convert the Linux specific TID to the pthread library pthread_t – and vice-versa –. The *GNU libc* (`glibc`) stores the Linux TID inside the pthread_t data, but, since pthread_t is to be considered opaque, nothing guarantees that it will be so in a different version of the same library. The only way to get the Linux tid of a thread is thus to call the `gettid()` Linux-specific `glibc` function *inside* the thread code. Since threads can be created and used from inside JACK clients code by using the JACK thread functions, and given the non-modifiability constraint for client code set at the beginning of this work, this API difference has been proven to be a problem.

Thus, to resolve this issue, a new, small, shared library has been written during this work. This library, named `libgettid`, wraps pthread calls so that it can intercept thread-related code in a way that is completely transparent to the process that the new thread is creating; then in the wrapped code it calls the `gettid()` system call and writes the TID in an internal private structure that maps it with the pthread_t. Then it exports a function to get the TID using the pthread_t as the key.

The wrapping is done using the `dlsym()` system call. The function `dlsym()` takes a "handle" of a dynamic library returned by `dlopen()` and the null-terminated symbol name, returning the address where that symbol is loaded into memory. `glibc` defines a special pseudo-handle, `RTLD_NEXT`, that searches the next occurrence of a function in the search order after the current library. This allows to wrapper around a function in another shared library, specifically the pthread library.

```

static void
libgettid_init(void)
{
    if (!real_pthread_create)
        real_pthread_create = dlsym(RTLD_NEXT,
                                    "pthread_create");
}

static void *
fake_start_routine(void *arg)
{
    /* [...] */
    /* get the needed information */
    tid = syscall(__NR_gettid);
    self = pthread_self();
    /* store them for later access */
    pthread_mutex_lock(&data_mtx);
    res = data_add(&threads_data, tid, self);
    pthread_mutex_unlock(&data_mtx);

    /* now call the thread original body */
    retval = args->routine(args->arg);

    /* following code is called when thread code
       returns */
    pthread_mutex_lock(&data_mtx);
    data_remove(&threads_data, self);
    pthread_mutex_unlock(&data_mtx);
    free(arg);

    return retval;

}

int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine) (void *),
               void *arg)
{
    struct create_args *args =
            calloc(1, sizeof(struct create_args));

    libgettid_init();

    /* call the wrapping routine, passing the
     * original in a field of the arg structure */
    args->routine    = start_routine;
    args->arg        = arg;
    error = real_pthread_create(thread,
                                attr,
                                fake_start_routine,
                                args);
    return error;
}

/* API function */
int pthread_get_tid(pthread_t thread, pid_t *tid)
{
    thread_id_t *info;
    if (threads_data == NULL) {
            return GETTID_E_NOTHREADS;
    }
    pthread_mutex_lock(&data_mtx);
    info = data_find(threads_data, thread);
    pthread_mutex_unlock(&data_mtx);
    if (!info) {
            return GETTID_E_THREADNOTFOUND;
    }
    *tid = info->tid;
    return 0;
}
```

For all this wrapping to work, the `libgettid` library should be loaded before the pthread library. This can be achieved by modifying the program build system to link the `libgettid` library, or, to avoid the need to patch and recompile clients, by setting the `LD_PRELOAD` environmental variable to the path where the `libgettid.so` file is installed on the filesystem, so that it gets loaded first even if the library was not linked during the compilation process.

3.2 dnl

`dnl` is a simple JACK client written to stress the JACK graph with a fictitious load, as to easily test modifications even on very high CPU usage. dnl takes in input from commandline a percent of the JACK loop it has to compute for and the previous and next client it has to connects its ports to.

During its process function it actively waits for that percent, translated in μsec by inspecting the sample rate and buffer size of the server, then it copies input port buffers to output port buffers, trying to simulate in this way a client computing for the same time. The active wait is simulated with the `clock_gettime()` syscall, using the Linux `CLOCK_THREAD_CPUTIME_ID` clock, which is an high resolution, per-process timer from the CPU (usually implemented using timers directly from the CPU itself[1]). This clock doesn't increment when the process is blocked, thus it's actually used to count the time spent by the process in the loop, and gives quite accurate timing for the purpose. This client is thus great in simulating the load of a JACK client like a filter with a almost constant execution time.

3.3 rt-app

rt-app is small test application, written as part of this work, that spawns N periodic threads, and simply runs the threads for the specified time. Similar to dnl, it makes heavy use of the `CLOCK_THREAD_CPUTIME_ID`, continuously checking it until the specified time have passed. Threads can be set to use the various scheduling classes supported by POSIX, and, if compiled with, by the AQuoSA framework, in which case it creates one server per thread.

During its execution it collects data (in memory, as not to block for I/O) and, after the execution ended, it dumps them to file.

This simple app can be used for both testing the performances of schedulers with respect to periodic applications or as a disturb application to simulate a realtime load in the system.

3.4 JACK AQuoSA controller

JACK internal has been modified to support resource reservations as they are supplied by the AQuoSA qreslib library. First, the build system was adapted to link the qreslib shared library, as well as the libgettid and qmgrlib shared

[1] Usually, on i386 platform, using TSC.

Table 3.1: *Statistics for the patch written to the JACK server, server library and client library to support AQuoSA and resource reservations.*

New lines of code	919
Deleted lines of code	359
Files changed	25
New files	2

objects.

Approaches to legacy feedback like those discussed and proposed in [5] could not be used or attempted, as the design of JACK imposes that only a thread of the application (being it client or server) application should be added to the reservation, thus making impossible to analyze JACK applications from the outside.

An alternative approach to the one implemented, described below, was to make all AQuoSA-related code be in a JACK internal client. This works to a certain extent while attaching and detaching clients to the AQuoSA server, but completely fail when trying to do feedback, as the internal client API does not provide any function or callback hook to synchronize with JACK cycle begin or cycle end. Directly modifying the JACK internal was then the chosen approach, with the aim of keeping changes as unobtrusive as possible, as to keep maintenances costs low when porting to a new JACK version.

All the code can be completely excluded at compile time, as every change is surrounded by `#ifdef`, so that the same codebase can be reduced to the distributed one. Moreover, the AQuoSA support can be completely disabled at run time, so that, when disabled, a single `if` statement is evaluated during JACK realtime critical paths.

By leveraging the `libgettid` support and shared memory already supported by JACK, almost no new code is inserted in the JACK client library, leaving clients almost untouched by these modifications.
Table 3.1 shows some statistics about the patch itself.

3.4.1 AQuoSA server creation

When configured to run with AQuoSA reservation by passing the `-A` switch to the jackd server commandline[2], all the AQuoSA code has been embedded in the `JackAquosaController` class. Reservation creation and deletion are handled, respectively, in this object's constructor and destructor. The `JackAquosaController` itself is created in the `JackServer::Open` function, right after the `JackEngine::Open` has successfully completed, but immediately before any driver creation, to be sure to catch the realtime threads the driver needs.

All the code needed for feedback scheduling and statistic account is thus self-contained in the class `JackAquosaController`. As stated above, when AQuoSA support is not enabled with the commandline parameter, the only change in execution with the respect to the distributed code is a statement evaluation, done in the `JackEngineControl::CycleBegin` function:

```
void CycleBegin( ... ) {
      #ifdef AQUOSA
        if (fAquosaController) {
          fAquosaController->CycleBegin( ... )
          ...
        }
      #endif
}
```

This code is the only code that "gets in the way"[3] when the AQuoSA runtime support is disabled, thus making possible to evaluate the performances of the *vanilla*[4]JACK version.

The `JackEngineControl::CycleBegin` function is called once per cycle in the realtime server thread. The `JackAquosaController::CycleBegin`

[2]All the commandline switch are also exported to the JACK2 new DBus control API, if enabled at compile time, although not used during this work.

[3]Obviously there is more code for thread creation and deletion, where there is additional code to attach and detach threads to the AQuoSA server, but we are referring to the code that get executed at every server cycle.

[4]*vanilla* is often used to refer to an unmodified version of some software, that is the code as it is distributed by the developers.

then takes care of setting all parameters using the previous cycle collected data, as the used budget and the others metrics, as it will be explained in depth in subsection 3.4.4 on page 54.

The `JackAquosaController` accepts, as parameters to its constructor, which are reflected to commandline switches, several options:

`-A` or `--aquosa` : enables the AQuoSA reservation within the JACK server and clients. It's disabled by default, and every other options presented below does not have any effect the server if `-A` is not present.

`-F` or `--fixed` : disables the feedback mechanism and sets a fixed budget

`-I <value>` or `--increment <value>` : the percentage, calculated as $incr = 100 + value$ to use as an increment with respect to values returned by the predictor, or, if `--fixed` was specified, the percentage of the total AQuoSA available bandwidth to reserve for JACK server and its clients. The relation $0 \leq value \leq 100$ must be true, with 0 accepted only with feedback enabled.

`-D <value>` : a multiplier to decouple the AQuoSA server period from the jackd period length, that is, if greater than 1, the AQuoSA server is created with a period which is D times the jackd period in μs. The same multiplier is applied to the budget to adjust it to the period being longer. The relation $value \geq 1$ must be true.

Once the AQuoSA serve has been created, its server id is placed in shared memory so that every processing can reference it to attach and detach threads or to query parameters.

As said, all modifications are surrounded by `#ifdef` statements, and are activated only if the `-A` parameter was present, so, from now on, it must be noticed that every feature presented can be removed at compile time or disabled at runtime reverting the original behaviour, even if not stated in the following explanations.

3.4.2 Attaching and detaching threads

The JackPosixThread class has been modified to support attaching threads to server if AQuoSA support is compiled and enabled in the JACK server. Again, all the modifications are surrounded by the `#ifdef` and activate only when `-A` was specified. For threads directly created by the JACK server or client[5], a thread proxy has been defined so that it's possible, in a similar way of what happens in the libgettid code described in section 3.1 on page 42, to execute code in the thread context in order to get the Linux TID for the thread being created:

```
#ifdef AQUOSA
struct fake_routine_args {
    void* (*routine)(void*);
    void *arg;
    int sid;
};

void *
JackPosixThread::fake_routine(void *arg)
{
    struct fake_routine_args *fargs =
                              (struct fake_routine_args*) arg;
    void *retval = NULL;
    pid_t tid;
    pid_t pid;
    if (!fargs || !fargs->routine) {
        jack_error("fake_routine called with NULL args");
        exit(12);
    }
    tid = syscall(__NR_gettid);
    pid = getpid();
    qres_attach_thread(fargs->sid, pid, tid);
```

[5]This class is shared by both the server and library code, so that for certain functions it is difficult, if not impossible, to assert from which side the code was called, especially those function that has declared as `static`.

```
    retval = fargs->routine(fargs->arg);
    free(fargs);
    return retval;
}
#endif
```

This method covers the majority of cases in which is needed to attach the thread to the AQuoSA server. However there are cases in which it's needed to attach a preexisting thread to the AQuoSA server, as in the case with the following two functions:

```
  int jack_acquire_real_time_scheduling (pthread_t thread,
                                          int priority);
  int jack_drop_real_time_scheduling (pthread_t thread);
```

To solve the "pthread_t to thread id" issue, every client that uses these functions must be compiled against `libgettid` or must be started, using the `LD_PRELOAD` trick as described in section 3.1 on page 42. This way is then possible to get the TID starting from the `pthread_t` value:

```
int JackPosixThread::AcquireRealTimeImp(pthread_t thread,
                                        int priority)
{
    struct sched_param rtparam;
    int res;
    memset(&rtparam, 0, sizeof(rtparam));
    rtparam.sched_priority = priority;


#ifdef AQUOSA
    int *sid = get_sid();

    if (sid != NULL)
    {
        pid_t tid;
        pthread_get_tid(thread, &tid);
        jack_info("JackPosixThread::AcquireRealTimeImp"
                  " attaching %d to server %d", tid , *sid);
        qres_attach_thread(*sid, getpid(), tid );
        return 0;
    } else
#endif
    if ((res = pthread_setschedparam(thread,
                                     JACK_SCHED_POLICY,
                                     &rtparam)) != 0) {
        jack_error("Cannot use real-time scheduling (RR/%d)"
                   "(%d: %s)", rtparam.sched_priority, res,
                   strerror(res));
        return -1;
    }
    return 0;
}
```

The `get_sid()` function simply gets the `sid` from shared memory, returning a pointer to the address in which the AQuoSA server id is stored: if it is equal to the `NULL` value it means that, even if AQuoSA support is compiled in, it is disabled for this execution.

Detaching of a thread is done in a similar way, with the very same problematic for the `jack_drop_real_time_scheduling` function.

3.4.3 Resource usage accounting

A key aspect of using feedback scheduling is to keep track of the used budget to compute the next, expected value. Instead of relying only on the qres-lib `qres_get_exec_time` library function, JACK client and server libraries were modified to account for their used budget, using the system clock `CLOCK_THREAD_CPUTIME_ID`. At every JACK cycle, thus, for the server and for each client, in their respective real time threads, this clock is queried soon after waking up (that is, immediately after the `read` call on the FIFO returns) and immediately before sleeping (call `read` on the FIFO again).

The difference between the two values is the total CPU time taken from that particular thread, which, when summing values from all other threads (included the server thread, and thus all the work eventually done by the JackAquosaController), gives the total usage statistic, which is used both as the next value to fill in the predictor queue and for graphing usage (as described in section 4.2 on page 67).

3.4.4 CycleBegin and Feedback

As stated in section 3.4.1 on page 49, all the feedback handling mechanisms, such as instructing the predictor, setting budget, changing period if needed and accounting the budget used, are done inside the function `CycleBegin` of the `JackAquosaController` object.

This function comprises:

- getting the used budget of the previous cycle by calling the `qres_get_exec_time()` function, which accounts for all the used CPU time of all threads attached to the specified server.
- adding the value to the predictor[†]
- getting the next estimated usage from the predictor[†]
- if the predicted budget or the JACK period (or both) have been changed from the last cycle, for example as the effect of the feedback for the former or as effect of a buffer size change for the latter, then it sets the new server parameters[†]

Every time it is needed to change AQuoSA server parameters, being the budget or the period length, a number or checks is performed, in order to avoid unnecessary calls to qreslib functions[6]. Being P the requested AQuoSA

[†]These actions are performed only if feedback mechanism is enabled, and skipped if JACK is run with AQuoSA support but with fixed budget, i.e. the `-F` switch is specified, with the exception of that, if the period changed, new parameters are computed and set even for the fixed budget case.

[6]Remember, as noted in section 3.4.1 on page 49, that the `CycleBegin` function is called in the jackd realtime server thread.

server period, Q the requested budget, f the fragment multiplier, M and m respectively the maximum and minimum settable budgets, I the increment as passed to `-I` on commandline:

- if $(P * f) < 1333$, then $f = f + 1$
- let $P = P * f$
- if fixed , then let $M = \frac{M}{100} * I$
- let $m = (P/100) * 5$
- if fixed, let $Q = M$, else if not –fixed:
  - let $Q = Q * \frac{100+I}{100}$
  - if $Q > M$, then let $Q = M$
  - if $Q < m$, then let $Q = m$
- if Q or P changed (one or both), call `qres_set_params` with new values.

These checks are always performed when setting server parameters, even when adding a new client or reacting to an xrun.

3.4.5 New Client and XRun events handling

When a new client marks itself as ready to process audio data with the server using the `jack_activate` library function, the predicted value as returned from the predictor cannot be accurate anymore. That is because the predictor is obviously incapable to know the future and to know in advance how much computation time the client will take.

To overcome this situation, upon a new client activation the predictor queue is flushed and discarded, and the budget is bumped by a fixed percentage, which is 10% of the actual budget if this is to the minimum possible or 20% otherwise.

Due to the queue flushing, we have the side effect that the budget is kept for a small interval fixed to this new value, during immediately successive periods, while the queue is being reconstructed. This heuristics has proved to be sufficient to handle new client arrivals, as can also be seen in chapter 4 on page 62, in which the experimental results are presented and discussed.

The same strategy is used when the JACK server experiments an xrun event: the budget is artificially bumped up to reconstruct the predictor queue in order to adapt to eventual changes that led to the xrun event.

3.5 Porting qreslib to cgroup-enabled kernels

As part of this work the AQuoSA `qreslib` library has been modified to use Linux cgroup (as described in section 2.5 on page 33) to set scheduling parameters: the main reason for this port was to make possible for the modified JACK described in 3.4 on page 47 to run on new kernels that supports control groups, in particular the patchset to the Linux kernel described in section 2.5.3 on page 38.

Primary aim of this job was then to leave the qreslib API intact in order to maintain API compatibility in order to make JACK run on top of it.

This port depends on `libcgroup`, a library and a set of tools aimed to work with cgroups, and it makes uses of functions exported by that library when creating, deleting, and attaching / detaching of threads and processes.

3.5.1 Mapping AQuoSA servers on cgroup

As explained in section 2.5.3 on page 38, the kernel interface for the `rt-edfthrottling` patches exports for each cgroup 4 virtual files, the period and runtime of the group itself and the period and runtime of the task belonging to the group, then groups are scheduled using EDF, while tasks inside the group are scheduled using a Round-Robin like algorithm.

This scheme maps to the AQuoSA API quite well. In particular, a root group (which must called *aquosa*) has to be prepared by the system administrator. This way is possible to set a maximum bandwidth utilization for all the AQuoSA servers, if its required for some reason (either technical or commercial) to limit the system usage of real-time tasks. Moreover, the system administrator has to setup both the `cpu` controller and the `cpuacct` controller, create the “aquosa” root group in it and assigning it a correct owner and group so that users part of the chosen group can manipulate the subsystem (making it possible to reserve bandwidth for their tasks). Since the bandwidth is reserver for real-time tasks, those users still need permission for using SCHED_RR or SCHED_FIFO scheduling classes.

`libcgroup` can be used to ease this tasks, as it provide a init-time service which mounts the cgroup filesystem and create all the needed cgroups hierarchy as defined in the file `/etc/cgconfig.conf`. A sample `cgconfig.conf` file can be:

```
group / {
        cpu {
                cpu.rt_period_us = 1000000;
                cpu.rt_runtime_us = 950000;
                cpu.rt_task_period_us = 1000000;
                cpu.rt_task_runtime_us = 50000;
        }
        cpuacct { }
}

group aquosa {
        perm {
                task {
                        uid = root;
                        gid = realtime;
                }
                admin {
                        uid = root;
                        gid = realtime;
                }
        }
        cpu {
                cpu.rt_period_us = 1000000;
                cpu.rt_runtime_us = 900000;
                cpu.rt_task_period_us = 1000000;
                cpu.rt_task_runtime_us = 0;
        }
        cpuacct { }
}

mount {
        cpu = /cgroups/cpu;
        cpuacct = /cgroups/cpuacct;
}
```

As can be seen, a file like this reserve the 5% of the total bandwidth (which, in turn, is the 95% of the system bandwidth) to the root group (in which, by default, start all tasks) and leaves the 90% to the aquosa controller. It also instructs libcgroup to mount the cpu and cpuacct controllers. With this files users of the *realtime* group can manipulate aquosa groups.

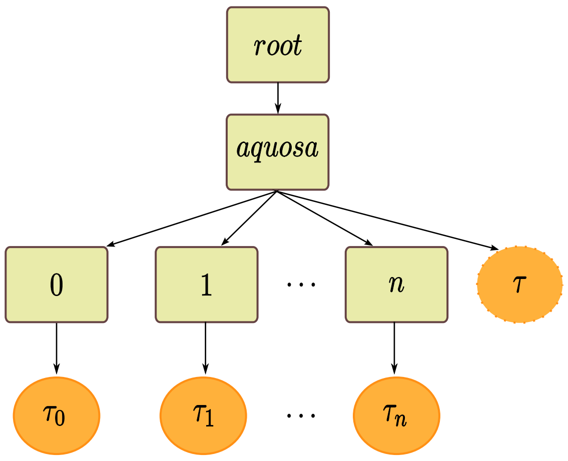

Figure 3.2: *Mapping of AQuoSA servers to cgroups*

Once the cgroup hierarchy is setup, when a tasks call the `qres_create_server` function a new cgroup is created under the aquosa root, and successive `qres_attach_thread` calls (targeting the returned server id) move threads to that group.

This way servers (which are mapped on groups) are scheduled with the EDF scheduler, while tasks inside a server are scheduled using RR. Figure 3.2

shows how cgroups are configured to support AQuoSA servers. Note that no tasks are allowed to belong directly to the aquosa cgroup, in fact its `cpu.rt_task_runtime_us` is set to 0.

3.5.2 Cgroup interface performance

During this porting, however, libcgroup has been found to be unusable for actually implementing feedback scheduling on top of it, and this only for a matter of performance.

Creation and deletion of servers, as well as attaching and detaching of threads, in fact, are not performance critical operations, as a client that need resource reservations usually calls this functions when it starts and when it stops. Clearly new threads can be created or removed on demand, but usually there is no need for high performance in this regard. Feedback scheduling, on the contrary, forces the application to read and adjust its values periodically, and this period can be very short, near to the millisecond range.

Early benchmarks showed that using libcgroup to adjust scheduling parameters took approx. 400 μs for writing and 300 μs for reading. In a cycle of "read accounting value", "write new parameters" this would lead to approx. 700 μs just for the feedback handling, leaving less than 30% of period for computations. This long times are due to the fact that libcgroup, being aimed for system administration tasks, does a lot of sanity checks prior to write or read values. This is surely desirable for normal operations, but when performance are critical a compromise can be found.

qreslib timings								
	ioctl				**cgroup**			
	Min	*Max*	*Avg.*	σ	*Min*	*Max*	*Avg.*	σ
set	0.3383	1.4730	0.3611	0.1099	1.7622	1.8560	1.7831	0.0131
get	0.1817	0.4528	0.1882	0.0323	0.9548	1.0260	0.9619	0.0097
time	0.2188	0.4227	0.2258	0.0284	0.5484	0.6151	0.5548	0.0075

Table 3.2: *`qreslib` timings on the three more common functions.*

Figure 3.3 on the next page report the same timing in a graph.

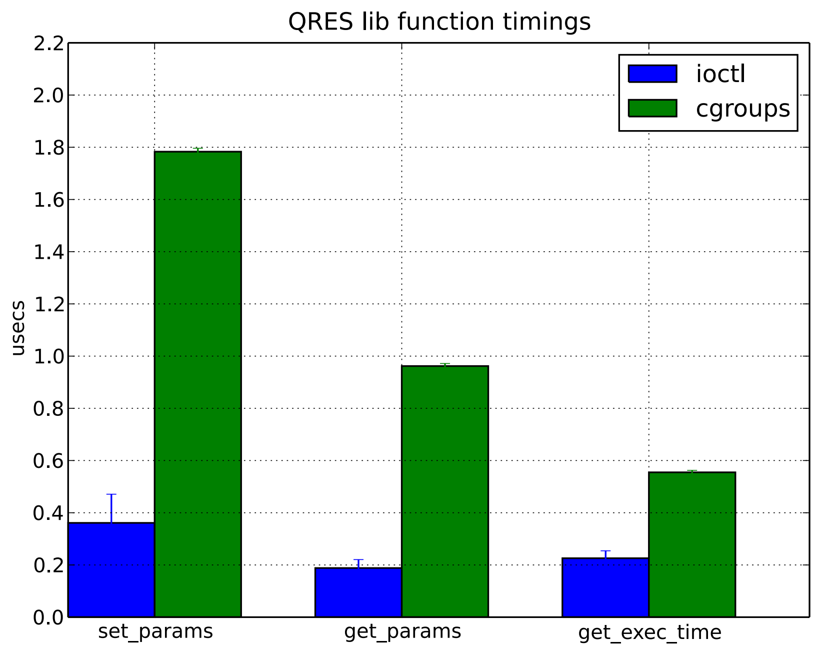

Figure 3.3: `qreslib` *timings on the three more common functions.*

Thus, too solve this issue when the `qreslib` opens or creates a cgroup to represent a scheduling server, caches all the file descriptors into the library, so that successive call to the same cgroup (i.e. using the same server id from an API point of view) reuses the open descriptors without the need to have a cycle like open/read/close. This could lead to potential problems if the cgroup is removed from the system between successive calls.

Table 3.2 on the preceding page show time taken by the three most used function of the API when adjusting the feedback, for each of the two implementations. Results of this quick benchmarks show clearly how the `ioctl`-based implementation in the AQuoSA `rresmod` kernel module is faster compared to the cgroup pseudo-filesystem, which is thus better suited for continuously adapting the scheduling parameters.

Another disadvantage of the cgroup-based interface is that scheduler parameters have to be set in a particular order, that is if both period and runtime have to be change at once care has to be taken on the order of which the four parameters are set in the respective four files on the virtual file system. For example, if shrinking both period and budget, the budget has to be shrieked first, as doing the contrary may temporary increase the requested bandwidth, possibly making the change to fail by the means of the admission control. This has multiple cases in which the order matters, and it's the replicated in the kernel itself, as when the kernel react to changed parameters in cgroup files then it has to follow the correct order again, leading to having twice the same code, once in userspace and once in kernel space.

Chapter 4

Experimental results

The real-time performance of the proposed modified JACK server has been evaluated by extensive experimental results, which are presented in this chapter. First, a few results about the real-time performance of the adopted real-time schedulers and kernels are presented, as compared to the behavior of the *vanilla* kernel, for the sake of completeness. These results have been gathered on sample scenarios built by using a synthetic real-time application developed solely for this purpose. Then, results gathered from running the modified version of JACK so as to take advantage of the real-time scheduling of the underlying OS kernel are presented and discussed in detail.

4.1 Test platform

For experiments and tests a quite common consumer PC configuration was used, with the Gentoo GNU/Linux distribution installed as OS with the following technical characteristic.

- Processor type and feature:

vendor_id	:	GenuineIntel
cpu family	:	6
model	:	23
model name	:	Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz
stepping	:	6
cpu MHz	:	2997.000
cache size	:	6144 KB

- Sound Cards:

Multimedia audio controller	:	VIA Technologies Inc. ICE1712 [Envy24] PCI Multi-Channel I/O Controller (rev 02)
Subsystem	:	TERRATEC Electronic GmbH EWX 24/96
Latency[†]	:	64
Kernel driver in use	:	ICE1712
Kernel modules	:	snd-ice1712
Audio device	:	Intel Corporation 82801I (ICH9 Family) HD Audio Controller (rev 02)
Subsystem	:	ASUSTeK Computer Inc. Device 829f
Latency[†]	:	0
Kernel driver in use	:	HDA Intel
Kernel modules	:	snd-hda-intel

- Main Memory:

MemTotal	:	2058944 kB
SwapTotal	:	2097144 kB

[†]This is the PCI bus latency, an integer between 0 (immediate release of the bus by the device) and 248 (the device is allowed to own the bus for the maximum amount of time, depending on the PCI bus clock). The higher this number is the more bandwidth the device has, but limits concurrent accesses to the bus by other devices.

- Kernel versions:
  - Linux 2.6.29-aquosa #5 PREEMPT x86_64 GNU/Linux
  - Linux 2.6.33_rc3-edfthrottling #1 PREEMP x86_64 GNU/Linux
  - Linux 2.6.31.12-rt20 #3 PREEMPT RT x86_64 GNU/Linux
- AQuoSA framework and kernel patches versions:

Generic Scheduler Patch version	:	3.2
qreslib version	:	1.3.0-cvs
EDF throttling version	:	2.6.33-git-20100221
qreslib-cgroup version	:	0.0.1-bzr-64

- System libraries and software:
  - GCC C compiler:

Target	:	x86_64-pc-linux-gnu
Thread model	:	posix
gcc version	:	4.4.2 (Gentoo 4.4.2 p1.0)

  - GNU C Library: 2.11
  - JACK and clients:

JACK audio connection kit	:	1.9.5-svn-3881

4.2 Measures and metrics

JACK version 2 (1.9.x) integrates a powerful profiler that records various timings to help understanding the behavior of the server and connected clients.

This profiler, enabled by a compile time switch, allocates memory for all its metrics on server startup, then starts filling up the in-memory data structure, avoiding saving timings to file while the server operates: as the profiling is done in the realtime server thread, blocking on I/O would affect negatively performances, poisoning the collected data. All the saving is thus done upon server shutdown. When the available memory for profiling is exhausted, the server restarts writing from the beginning, using the available space as a circular buffer.

Various metrics are profiled by the distributed version of JACK. Moreover, some other metrics were added to the profiler as they are interesting for this work:

audio driver timing allows to track, for each server cycle, the audio driver timings, that is the duration between consecutive interrupts. The corresponding plot is supposed, for normal operations, to be a regular curve, as flat as possible; theoretically the best situation is achieved by a flat horizontal line, meaning that every server cycle duration has been equal to the computer latency from the buffer size and sample rate using the formula:

$$latency = \frac{frames\ per\ period}{samplerate} * 10^6\ (\mu s)$$

When the server period is regular, measured without any clients active, then the server *asynchronous mode* can be used safely (remembering that it adds an additional buffer latency for the sake of reliability). On the contrary, a non regular interrupt timing forces the *synchronous* mode to be chosen, as the server could lack time to finish its execution if duration between two consecutive interrupts is too short. Figure 4.1 on the next page displays an example graph from this timing. One thing that should be noted in Figure4.1 is that spikes are always present in pairs: this is normal because the server, using double buffer, tries to re-sync shortening or enlarging a cycle if the previous one was longer or shorter than what it should be.

JACK CPU time tracks the usage, in percent, of the JACK period used to do DSP computation. Its computed by the server at every cycle using the collected timings. It offers no more information, as it is basically depending from the client end timing.

driver end date displays the driver[1] relative end time, that is the time from the cycle start, for each server cycle, at which the driver finished

[1]With *driver*, as discussed in section 2.2.2 on page 8, we refer to the JACK code that interfaces with the ALSA driver, exporting physical input/output ports to jack clients.

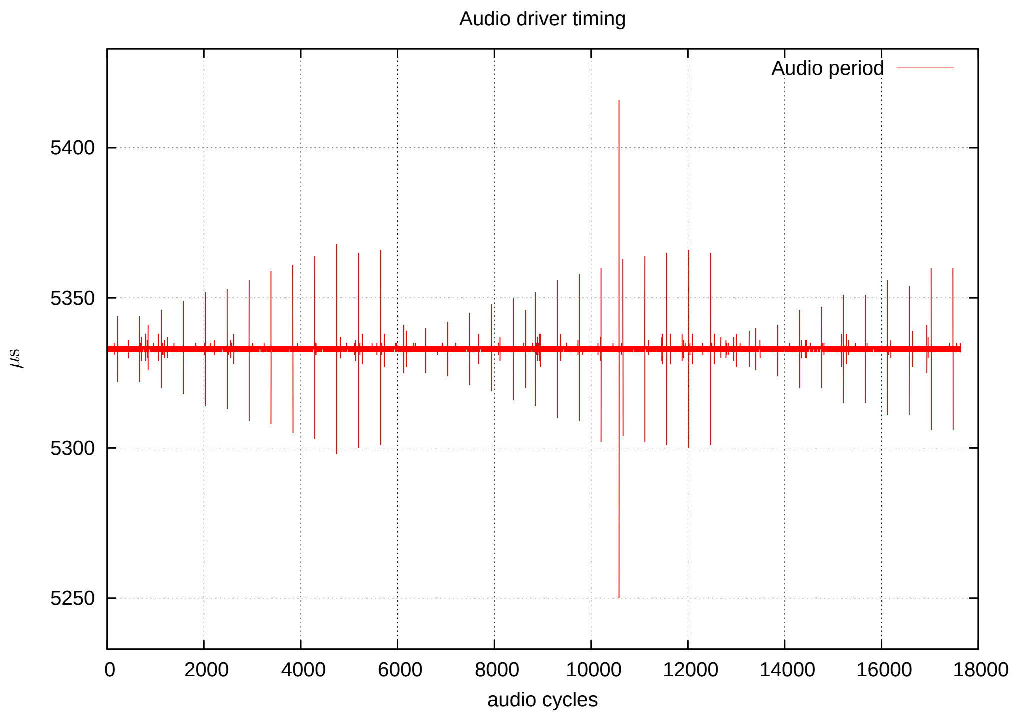

Figure 4.1: *Audio driver timing with the server configured to run with a sample rate of 48000 Hz and buffer size set to 256 frames per period, resulting in a latency of 5333 μs. The server is running with no clients attached. The measured period is quite stable, diverging very little from the theoretical value.*

to write audio data and started to sleep till the next driver interrupts. For each cycle this quantity should obviously be lower than the actual audio period for that cycle, as otherwise this would mean that an *xrun* happened (the server was not able to write data before the ALSA driver needed it). The corresponding plot is interesting while evaluating the difference of the two server operational modes: when the server is running in asynchronous mode, in fact, the driver does not wait the graph end, but returns immediately after the write step, resulting in a very different curve from those generated by a synchronous mode run.

clients end date takes, for each active client, all its end times (relative to the period start). The generated curve provides an overview of the DSP usage of each client, as well as the audio period timing used as a reference. Here, as with the driver end date timings, the server is working correctly if the last client end time is less than the audio period.

client scheduling latency measures the difference between client activation, that is the time when a client has been signalled to start by previous clients, and the actual wake-up time. When the client real-time audio thread becomes runnable, the global scheduling latency depends on the processor to be available and on the actual OS's scheduling latency. These values thus depend on various external factors, such as the topology of the JACK graph, the scheduler in use and the number of processors the PC has.

clients duration measures the difference between client end date and client actual wake up time, resulting in the actual duration of the client at each cycle. This values includes interferences due to other processes possibly scheduled by the OS while the client was executing.

AQuoSA budget was added as part of this work, and it takes three timing measures regarding the AQuoSA CBS server associated with the JACK server instance running. The three metrics are the budget set at the beginning of the period, the predicted value that the feedback mechanism computed to be set, and, at the end of the cycle, the actually used budget. If this last measure is larger than the budget that was set it means an *xrun* has occurred.

Sometimes the above introduces timings measures are shown using a *cumulative distribution function*, or **CDF** for short, which completely describes the probability distribution of the quantity plotted. Taking period duration ρ as an example, its CDF is given by:

$$x \mapsto F_\rho(x) = P(\rho \leq x)$$

where the right side represent the probability that the variable ρ takes is lower than or equal to x. The CDF can be defined in terms of the probability density function $f_\rho(\cdot)$:

$$F_\rho(x) = \int_{-\infty}^{x} f_\rho(t)\mathrm{d}t$$

Plot of a cumulative distribution often has an S-like shape, where the graph of the ideal distribution of measured periods would be a step-like curve, meaning that the probability of a point to be both greater than or lower than the ideal value would be zero.

4.3 Basic timings

Some experiments were done by running the JACK server alone to measure audio driver interrupt timing under different server parameters, scheduling policies and sound hardware. The purpose of these tests is to compare asynchronous and synchronous modes of operation, to compare different Linux kernel versions that have been tested and, finally, to compare the two hardware sound cards used during this work (to see if a consumer, built-in audio card can offer similar performance to a, despite being quite old, *pro-sumer* card such as the `ICE1712` card is). In all these tests the feedback mechanism implemented inside JACK as part of this work was disabled, and a bandwidth of 94% was reserved for JACK operations, to limit disturb factors during these tests.

4.3.1 Jack Async vs Sync Mode

These tests aim to show the basic timing of the audio driver interrupt, to see, with no other realtime processes nor clients running, how *regular* the interrupts are - and thus the JACK server period. Another purpose of such tests is to see how stable is the period when using the asynchronous mode for the JACK server.

As noted in section 4.2 on page 65 while describing the audio driver timing, the spikes come in couples. On Figure4.2 this simple test is run on the `ICE1712` card using a sample rate of 96000 Hz, both in synchronous and asynchronous mode of operation for the JACK server. Figure4.2 shows a

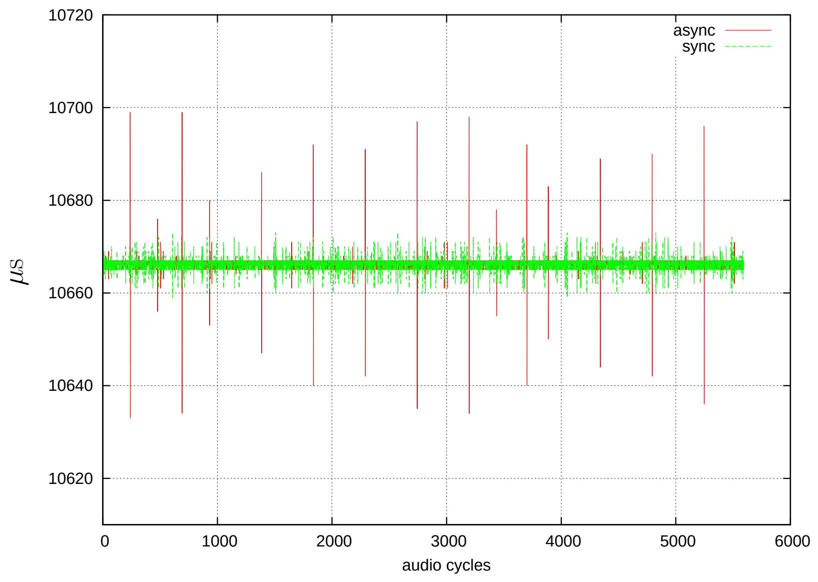

Figure 4.2: *Audio driver timings, sync and async mode at 96000 samples per period with a latency of 10666 μs on ICE1712 card using the AQuoSA scheduler with a 95% fixed bandwidth.*

latency of 10.7 ms, while Figure4.3 shows a very low latency of 1.3 ms. To be remarked the fact that asynchronous mode adds an extra buffer latency to the server, so when using that mode actual latencies are, respectively, 21.3 ms and 2.6 ms, the same as if the buffer size is doubled when in sync mode.

Figure 4.4 on page 71 and Figure 4.5 on page 72 represent the very same experiment run with a sample rate of 48000 Hz instead. Lower sample rate means, as to leave the output latency fixed, larger buffer sizes; thus, to be consistent with experiments at 96 kHz, buffer sizes were changed accordingly.

These simple experiments show that, while the period is quite regular in both operational modes, the sync mode is the one that has more regularity. This was expected, as the server operations are more strict and tightened to the audio driver interrupts. It should be noted, again, that this operational mode is also the one that provides better performance with respect to latency, while degrading the reliability of the server itself, as it tolerates less disturb and could possibly generate more ALSA xruns. That said, from now on, all other tests are run in sync mode, since using Resource Reservations re-adds the lost of robustness.

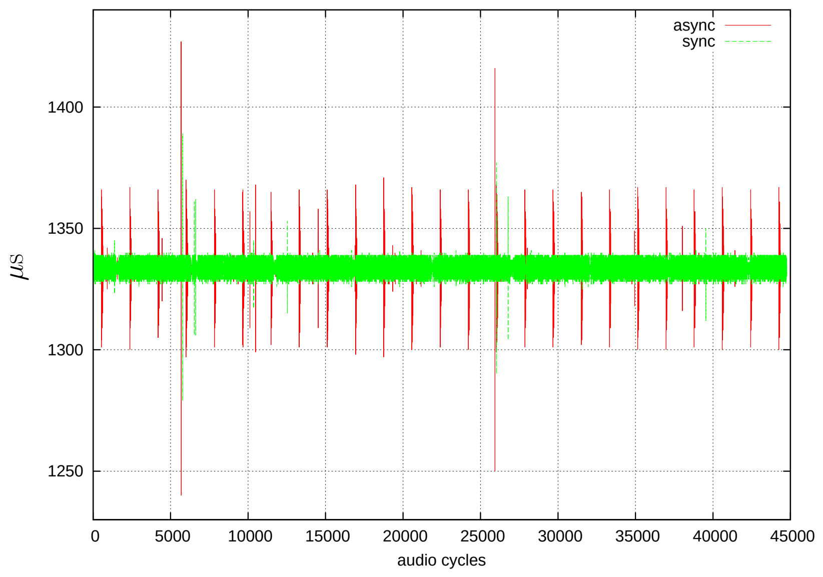

Figure 4.3: *Audio driver timings, sync and async mode at 96000 samples per period with a latency of 1333 μs on ICE1712 card using the AQuoSA scheduler with a 95% fixed bandwidth.*

From these experiments it can also be seen that the differences between the two audio cards are minimal with respect to the interrupt timing, meaning that both cards perform the same from the JACK point of view. Since that, in following experiments the `ICE1712` card has been preferred, as it supports more frequencies (11025Hz, 22050Hz, 44100Hz, 48000Hz and 96000Hz) and more buffer sizes (from 64 up to 2048 samples) thus ranging from 0.7 ms latency with 64 frames as buffer size at 96kHz sample rate to 185.8 ms with 2048 frames at 11,025 kHz. While the latter is definitely uninteresting for this work, the former is the corner case, as a latency under the millisecond can be very challenging for the system to support, while it ensures the best responsiveness the system can offer with the hardware in use.

It should be noted, however, that as latency increases reliability does as well, and with 0.7 it ms is very likely that xruns will show more often than at larger latencies, so it should be used in situations where some losses on the generated output could be tolerated.

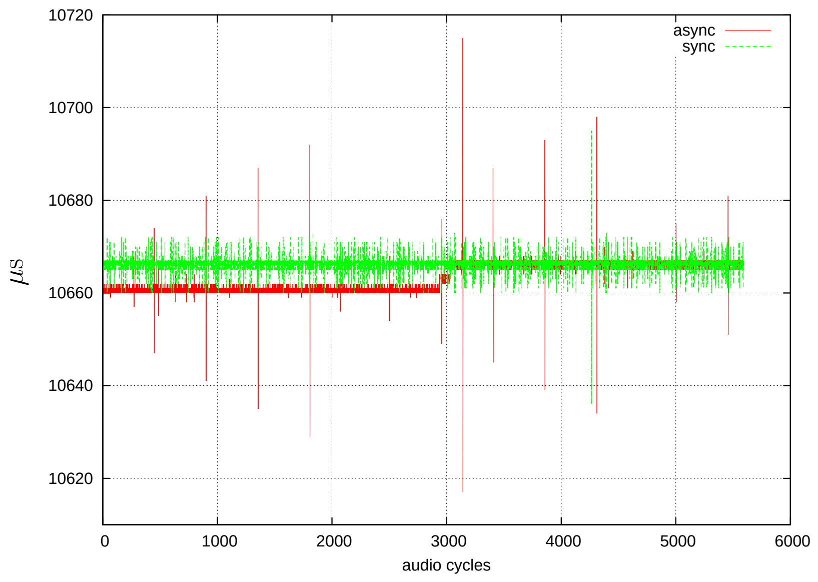

Figure 4.4: *Audio driver timings, sync and async mode at 48000 samples per period with latency = 10666μs on ICE1712 card using the AQuoSA scheduler with fixed budget.*

4.3.2 Kernel and scheduling timing

With this experiment we want to show how JACK behaves with the two main used schedulers and with the PREEMPT-RT Linux kernel patches. In this experiment the jackd server is still run without any client attached to it, as very basic timing measures are being evaluated.

Within this experiment this only client was setup to be directly connected to system ports which abstract ALSA physical audio card outputs. Figures 4.6 on page 73 through 4.9 on page 76 show some measurements taken during this experiment. The jackd server was configured to run with a buffer size of 128 samples per period, with a sample rate of 96 kHz, using the ICE1712-based audio card as in section 4.3.1 on page 68: these parameters force a minimum latency of 1333 μs introduced by the JACK system itself.

Figure 4.6 on page 73 shows the period length, in μs, measured for audio cycle 32159 to 43958 of the run, for the 3 different schedulers we wanted to explore. The figure was zoomed-in to have better understanding of the behavior of the three schedulers. Timings are still quite precise with all of

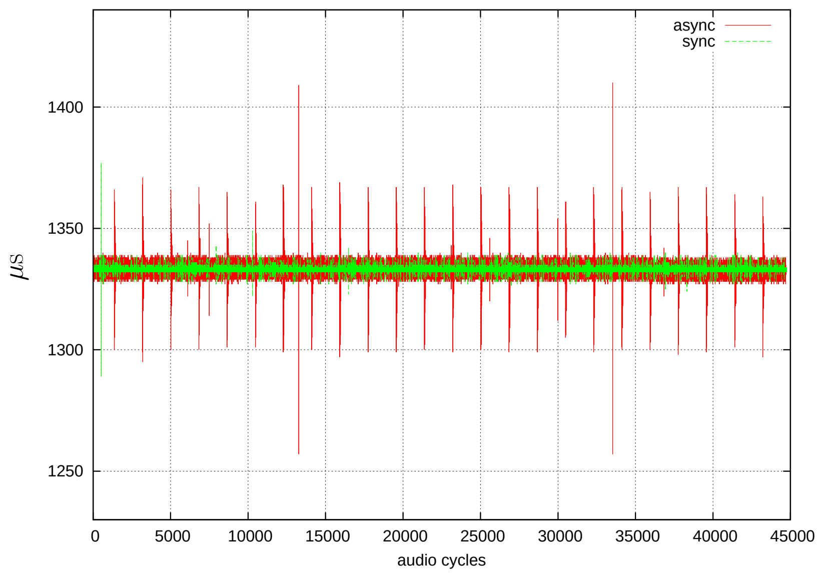

Figure 4.5: *Audio driver timings, sync and async mode at 48000 samples per period with latency = 1333µs on Intel HDA card using the AQuoSA scheduler with fixed budget.*

	Min	Max	Average	Std. Dev
Linux	1243	1423	1333.268	2.421
Linux-rt	1308	1357	1332.431	1.583
AQuoSA	1279	1389	1333.268	2.704

Table 4.1: *Audio driver timings of JACK running with 1333 µs latency at 96 kHz.*

them, shown in table 4.1, and shows that all three kernels can be used to work at the selected latency.

The PREEMPT_RT patchset is the one that gives better results in this test, followed by the AQuoSA scheduler and plain Linux. Should be noted as AQuoSA, while having closer minimum and maximum with respect to plain Linux, has a bigger standard deviation: this is what can be observed in Figure 4.6 on the following page, and in section 4.3.1 on page 68 experiments, that is the AQuoSA scheduler makes the JACK period jump frequently in close interval of the mean (which is equal in all of the tests to the expected latency). This is not a problem for JACK to act correctly, and it is probably due to the CBS server itself.

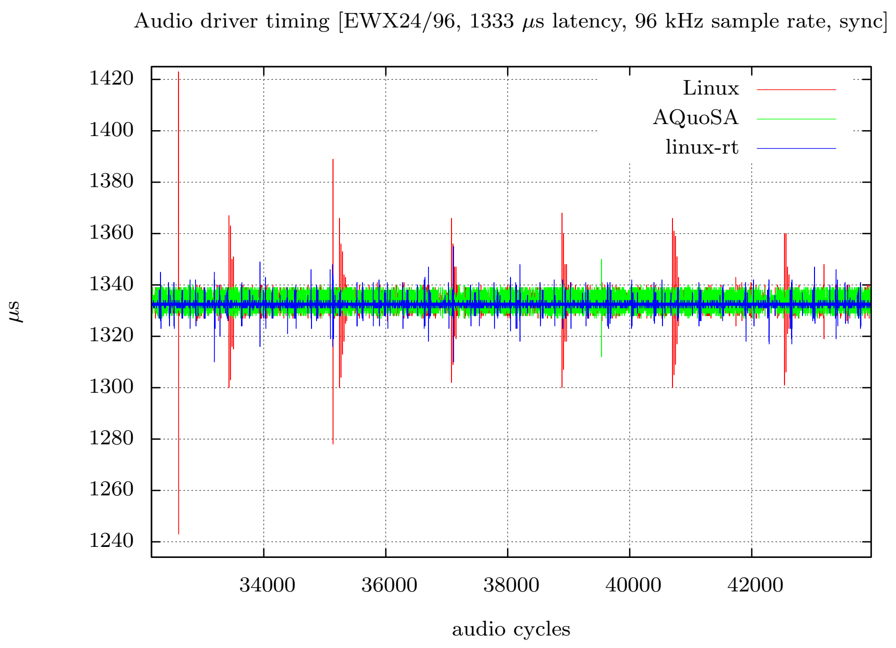

Figure 4.6: *Audio driver timings of JACK running with 1333 μs latency at 96 kHz, using SCHED_FIFO, linux-rt SCHED_FIFO and AQuoSA.*

Figure 4.7 on the following page shows the cumulative distribution function of the all measured period data as an alternate view.

Figure 4.8 on page 75 and Figure 4.9 on page 76 shows the driver end time, for each audio cycle, of the same experiment, while Table 4.2 on the next page shows the driver end timing statistics. In these graphs and tables can be seen the very same behavior found while looking at period timings. Again, the linux-rt scheduler is the one which has better results, with faster response and lower jitter, while AQuoSA sits in the middle, even if it has the single worst response time. "Normal" Linux scheduler perform well in average and minimum response time, but periodically reports spikes up to 40-44 μs, which are in line with the results seen so far.

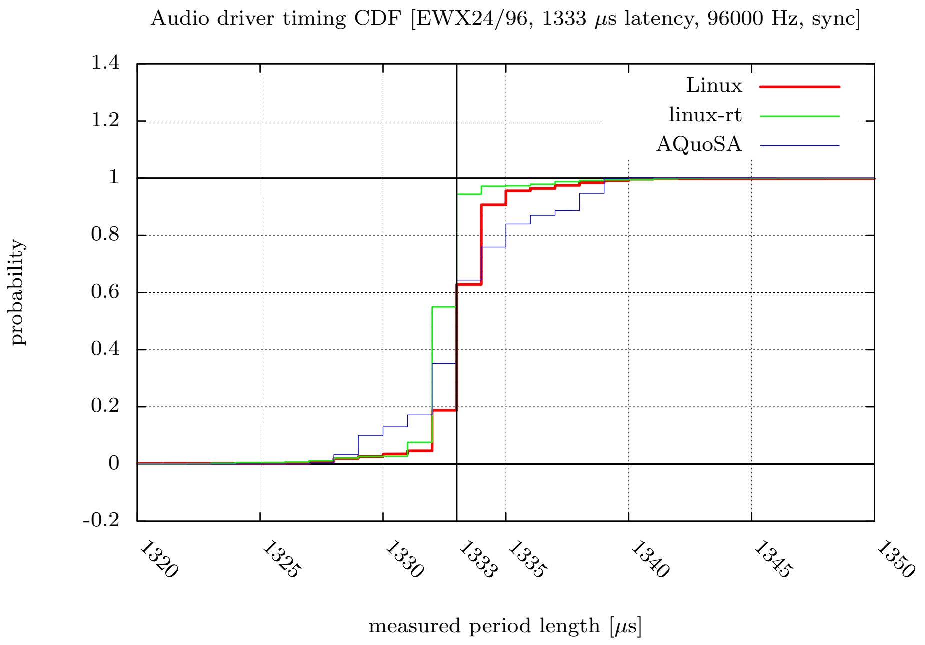

Figure 4.7: *JACK audio driver timing CDF with different schedulers.*

	Min	Max	Average	Std. Dev
Linux	4.0	47.0	7.418	1.384
linux-rt	4.0	33.0	7.233	0.757
AQuoSA	6.0	55.0	9.529	1.528

Table 4.2: *Driver end of JACK running with 1333 μs latency at 96 kHz with Linux, linux-rt and AQuoSA schedulers.*

4.4 JACK and other real-time applications

For this experiment jackd and two `dnl` clients were setup to run. Each dnl client, as described in section 3.2, has two input ports and two output ports, and in its process callback it simply busy-wait for the amount of time it is configured to and then copies audio data from input to output. The JACK clients and server operations are perturbed with two `rt-app` threads, as described in section 3.3.

The first `dnl` client is started a $t = 15s$, while the second `dnl` client starts at $t = 30s$. The two rt-app threads are started at the same time of the two dnl clients, and they both stop after $35s$ (that is at $t = 45s$ and $t = 60s$

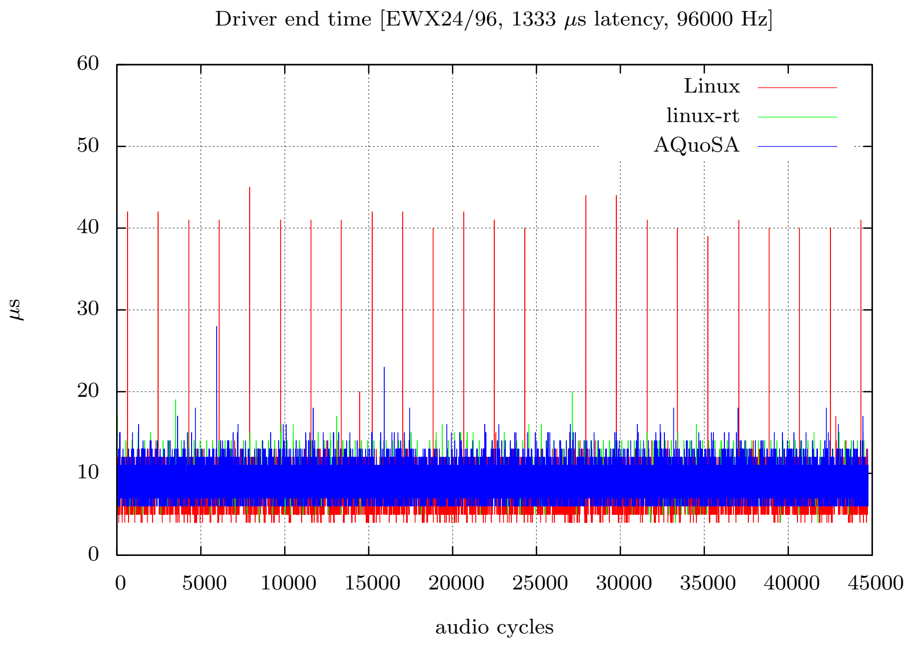

Figure 4.8: *JACK audio driver end time with different schedulers with 1333 μs latency at 96 kHz.*

respectively), while both dnl clients are being shutdown together at $t = 60s$. The total time length of the experiment is $90s$. Both dnl clients are set up to consume 10% of the audio cycle time, so, as the latency is set to 1333 μs with 128 buffer size and 96 kHz sample rate, as in previous experiments, they both are configured to run for approx 133 μs at each cycle[2]. rt-app threads have a period of 10 ms and a computation time of 3 ms.

This scenario has been repeated four times, and in each run the scheduling class (or the priority, as explained in depth below) of rt-app processes and JACK clients/server was changed . The aim of this experiment is, in fact, to test how well the JACK server can operate with other tasks (represented by the two rt-app threads) running real-time in the system. This is the field in which the CBS scheduler can bring big improvements, much more than in the basic timing experiments presented in sections 4.3.2 and 4.3.1, as the resources asked by the JACK server as reserved for it and its clients.

[2]Remember that JACK calls the clients *process* callback once per audio cycle.

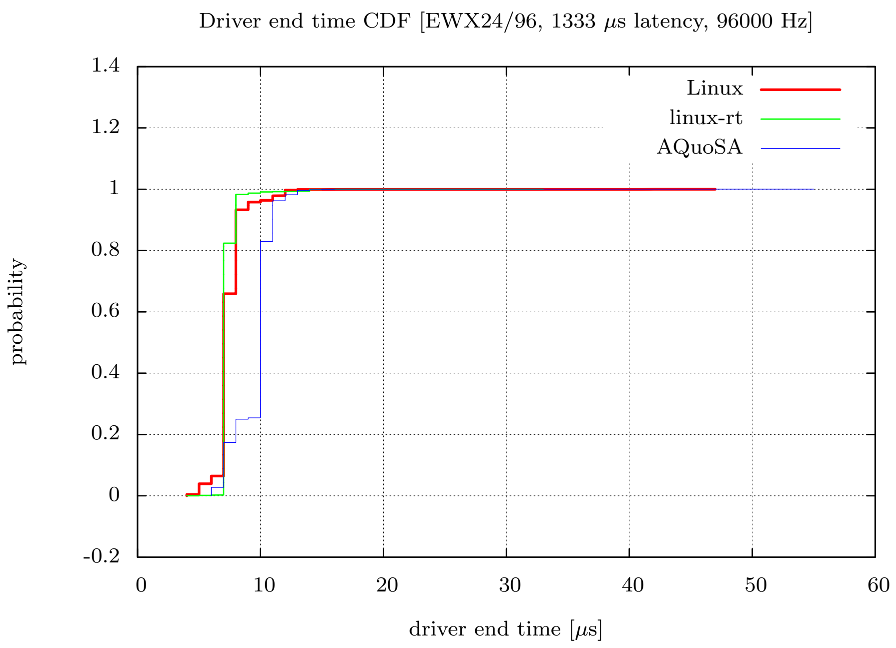

Figure 4.9: *JACK audio driver end time with different schedulers with 1333 μs latency at 96 kHz.*

During these tests, when jackd was using AQuoSA reservations, the feedback mechanism was enabled, so as to limit to the safe minimum the bandwidth requested by the JACK server and clients, and to make possible to reserve bandwidth for the rt-app tasks as well.

Four runs where done, using the following combination of schedulers:

	JACK	rt-app tasks	JACK prio.	rt-app tasks prio.
1	SCHED_FIFO	SCHED_FIFO	10	10
2	AQuoSA	SCHED_FIFO	—	10
3	AQuoSA	AQuoSA	—	—
4	SCHED_FIFO	SCHED_FIFO	11	10

The fourth run was using SCHED_FIFO for both JACK and rt-app, but priorities, in contrast with the first run in which all processes have the same priorities, were manually assigned as a rate monotonic scheduler would have assigned them.

Figure 4.10 on the following page shows the driver end timing of the four

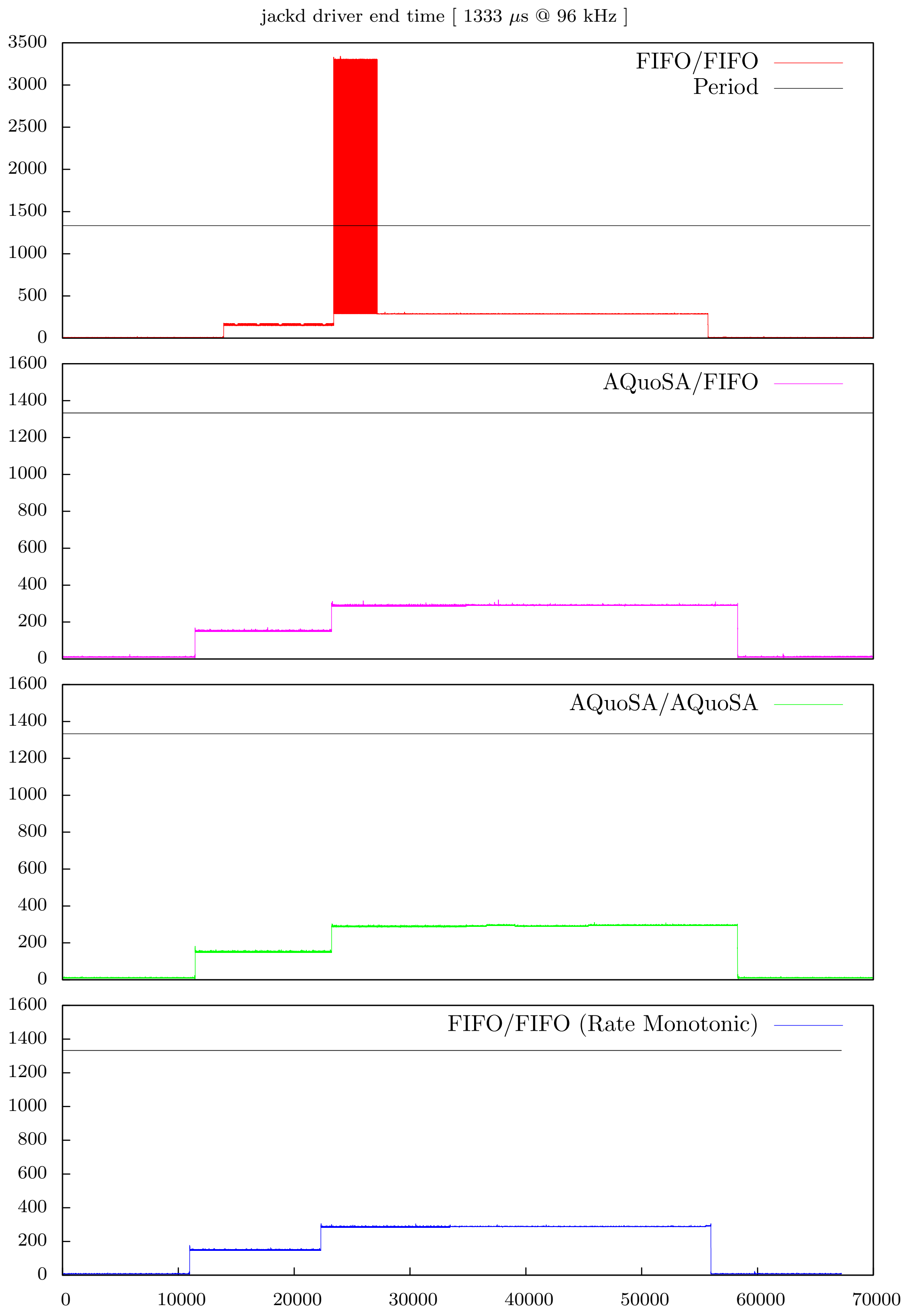

Figure 4.10: *JACK disturbed driver end time.*

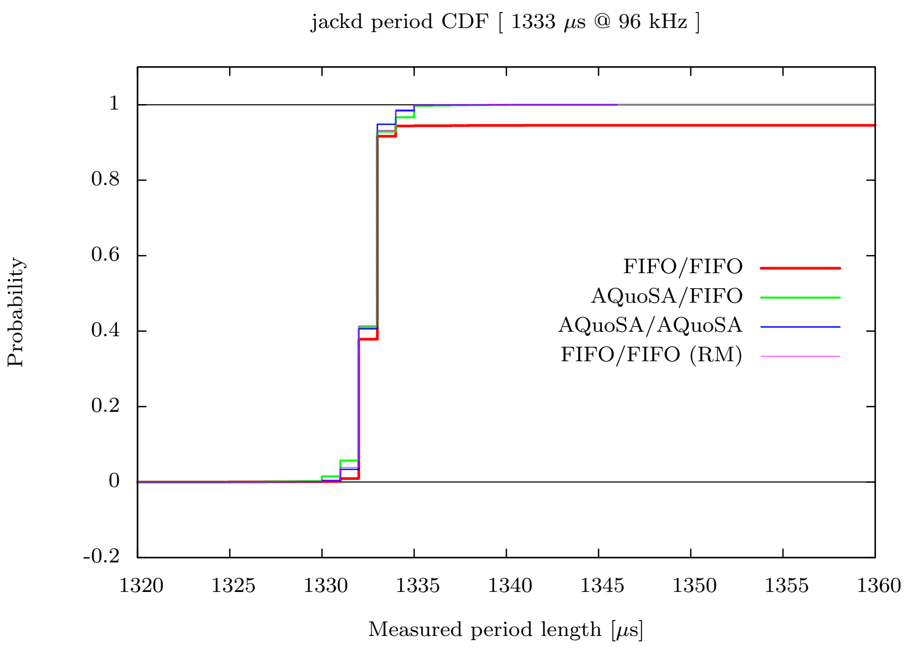

Figure 4.11: *JACK disturbed driver timing CDF.*

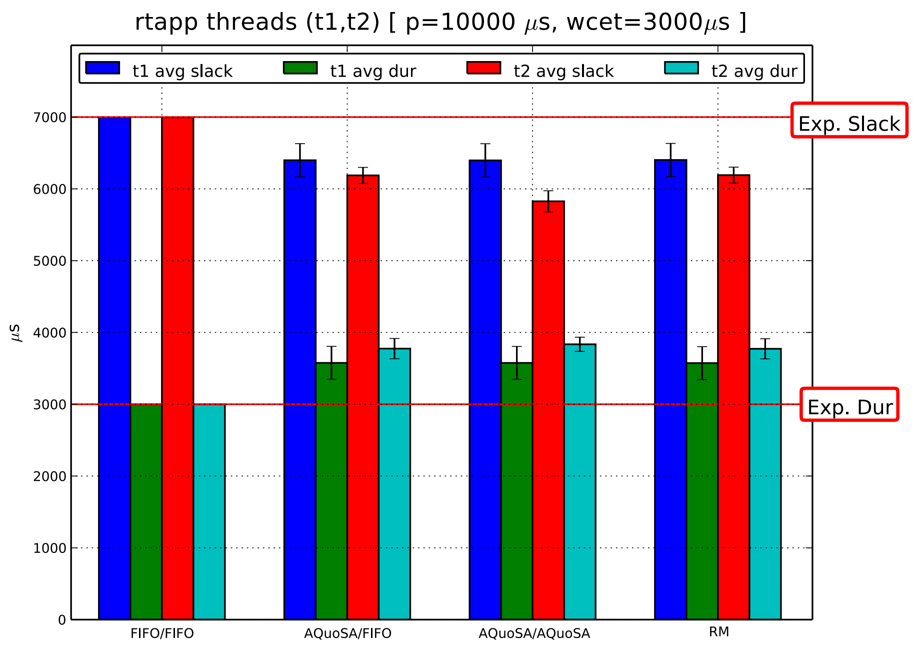

Figure 4.12: *rt-app disturb threads with different scheduler.*

runs. It can be clearly seen that in the case of both system running in SCHED_FIFO without using rate monotonic the JACK server is greatly penalized, as it cannot preempt long running rt-app tasks, thus leading to a non-working system with end time of 3000 μs and more.

Figure4.11 shows the cumulative distribution of the period timing for the four runs, zoomed in to see the behavior near 1333 μs.

Finally Figure4.12 shows the rt-app threads slack and real duration averages for all the four runs. In this graph can be seen how in the equal priorities SCHED_FIFO run, while the JACK server was not able to complete its work, the rt-app tasks have precise timings, as they manage to get enough CPU time to complete their work.

This experiment thus shows how having resource reservations helps in getting more than an application to run in respect of its deadlines, without requiring the user to have knowledge on how to set priorities in order to get expected results. The budgets for JACK and rt-app are, in fact, auto discovered by applications themselves, the don't require the user to set them explicitly.

4.5 Complex scenarios

In this section, different scenarios are going to be discussed in which multiple clients were started with the JACK server, to show how the server behaves using AQuoSA and high cycle usage (up to the 75% of the total available time per cycle), that is to try to push the usage to a corner-case limit. AQuoSA scheduler was configured to run with SHRUB enabled and disabled, and finally an other experiment had an instance of rt-app running.

4.5.1 AQuoSA with or without SHRUB enabled

SHRUB, as described in section 2.3.6, is a mechanism that extends the CBS server found in AQuoSA to reassign the spare time in the system to those active servers. In particular, it has been found very useful with the JACK server: one of the major problems with feedback scheduling applied to the

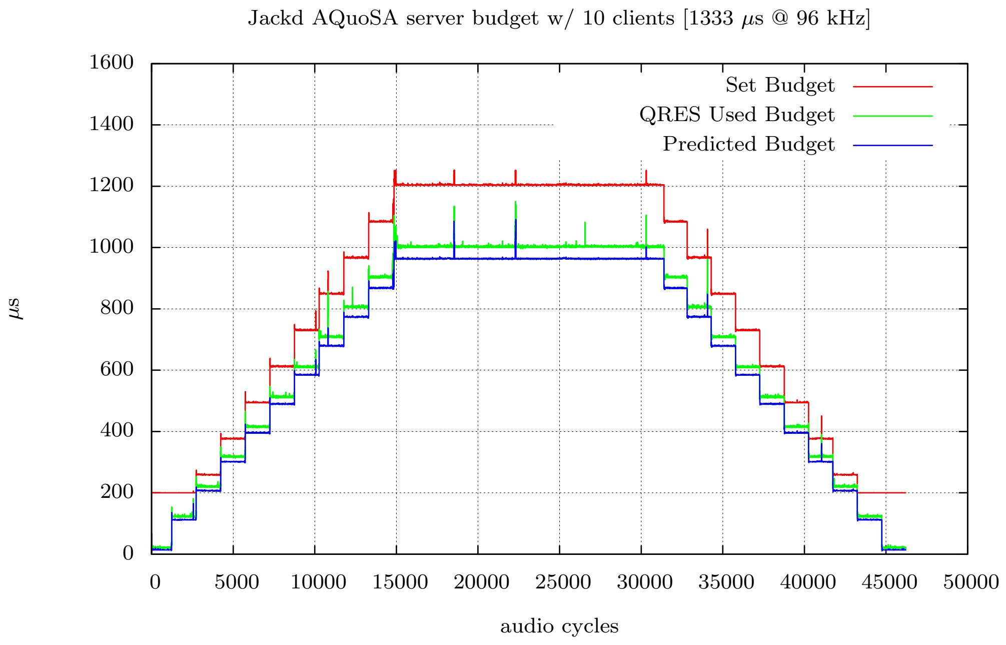

Figure 4.13: *AQuoSA budget trend with 10 clients. To be noticed the low bound under which "set budget" (red line) remains fixed, while otherwise it is an overbooking of the "predicted" one (blue line).*

JACK architecture is that if the CBS server exhausts the budget for JACK's tasks, then, being an hard reservation, it delays it till the recharge, causing a long and audible audio xrun.

A mechanism like SHRUB, thus, alleviates the problem that otherwise can be only resolved by overbooking the budget. Even strict overbooking is not enough, as a new client can completely change the total duration and thus the needed budget for JACK to complete its work meeting the deadline constraint. The spare time thus can be used to handle this particular situations, avoiding unnecessary overbooking.

In this experiment the AQuoSA framework is compared with and without SHRUB being enabled. A total of 10 dnl clients where setup to start at regular intervals, resulting in a staircase like utilization pattern up to the 75% of the cycle time. Each client is connected to the previous one, in a chain from the input ALSA ports abstracted by the JACK server, to the output ports. The resulting graph is then serialized and every client unblocks the next one,

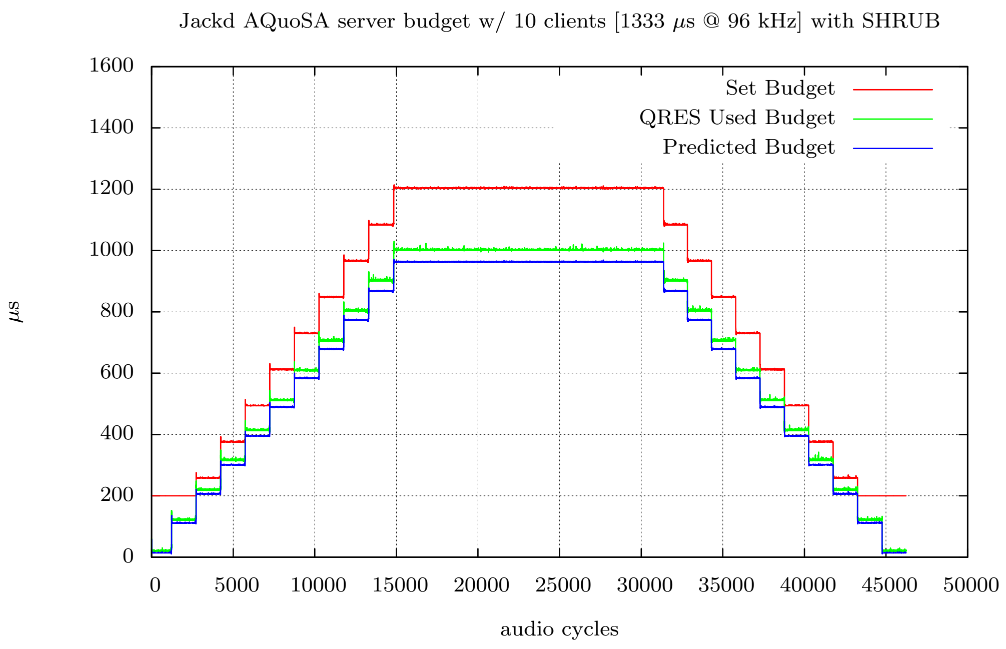

Figure 4.14: *AQuoSA budget trend with 10 clients and SHRUB enabled.*

respecting the data-flow model discussed in section 2.2.5 on page 11.

Figure4.13 shows the set and used budgets for AQuoSA without SHRUB, while Figure4.14 show the same metrics (described in section 4.2 on page 64). Figure4.15 is still the same scenario with SHRUB active, but using a reservation which has the period and the budget doubled with respect to the JACK period and utilizations. Finally, Figure4.16, 4.17 and 4.18 show the driver end time and period plots for the same situation as above.

Both solutions can handle this configuration of clients, but the timings for the SHRUB enabled tests are much more regular. It should be noted, in the budget graphs, with and without SHRUB, the heuristic implemented to handle the "new client" situation. Since the predictor cannot known in advance how much CPU time a new client will take to complete, it simply bumps the budget up of a certain configurable percent of the so-far used budget. This is causing the spikes on each "step" of the staircase. This is much more evident when SHRUB is disabled.

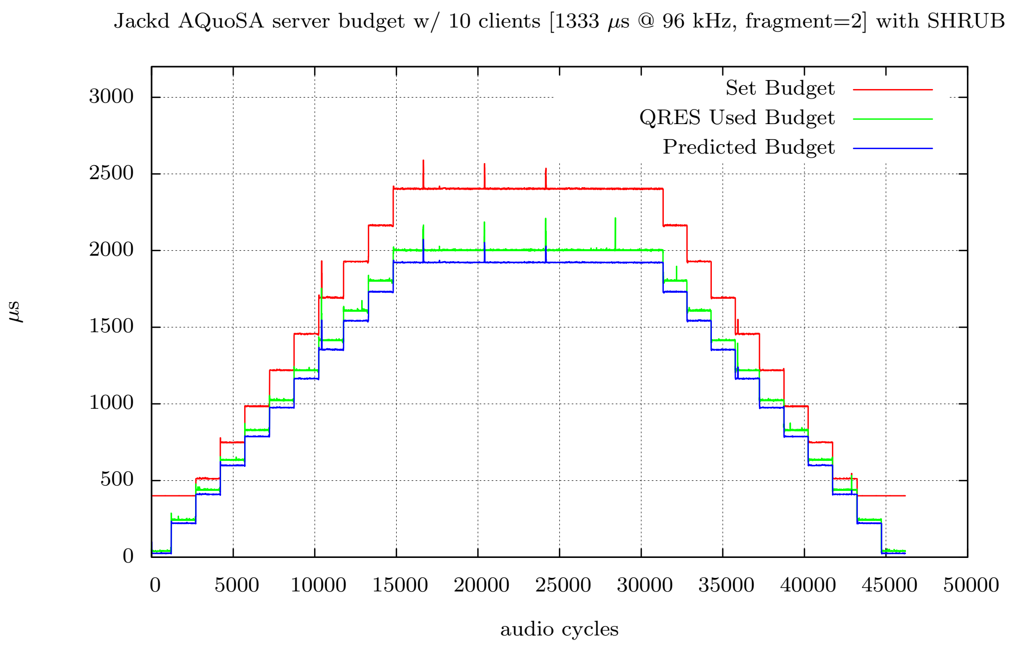

Figure 4.15: *AQuoSA budgets trend, using the double of JACK period as AQuoSA period (thus resulting in budgets being doubled as well. This mechanism of fragments is automatically enabled if the period is below 1333 μs.*

4.5.2 Multi-application environments

This experiment is configured as the one described in section 4.5.1 on page 79, with the difference of an added background rt-app that perturbw the JACK operations. This replicates the tests done in section 4.4 on page 74, but using a loaded JACK server with 10 clients connected for a total utilization of 75%. The rt-app thread are using a very long period and low computation time, in order permit to have its bandwidth reservation while asking for the ∼83% of the total bandwidth for the JACK itself. The rt-app thread is thus configured to use the 0.05% (a period of 10ms and a computation time of 500 μs) of the total available system bandwidth, and runs for all the experiment time.

For the AQuoSA-enabled JACK this is of no problem, and the experiment "ends smooth" resulting in no xrun reported by the JACK ALSA driver. For the SCHED_FIFO test, on the contrary, we can clearly see how it starts

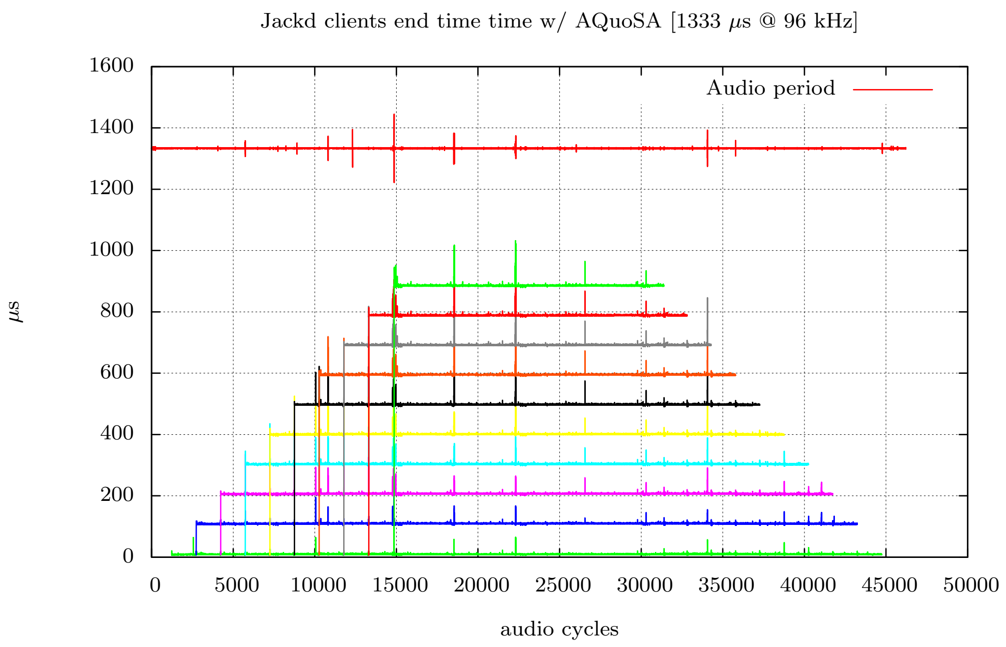

Figure 4.16: *Clients end time with AQuoSA scheduler and 10 clients. To be noted that when a client takes longer to complete, this is reflected in successive clients (that start later) and then on the audio period length. Clients are numbered from bottom to top, so Client0 (connected to the inputs) is the on the bottom, and Client9 (connected to outputs) is the topmost one.*

having a jitter of the total rt-app thread's computation time (which is equal to 500 μs), then completely misbehaving with more than 7 clients.

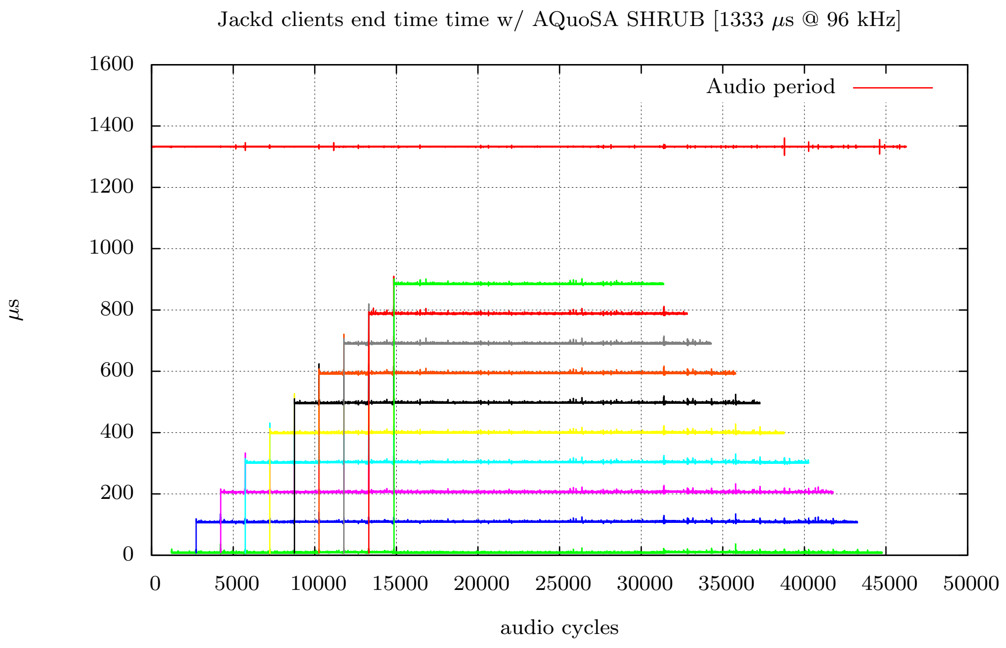

Figure 4.17: *Clients end time with AQuoSA scheduler, 10 clients and SHRUB enabled.*

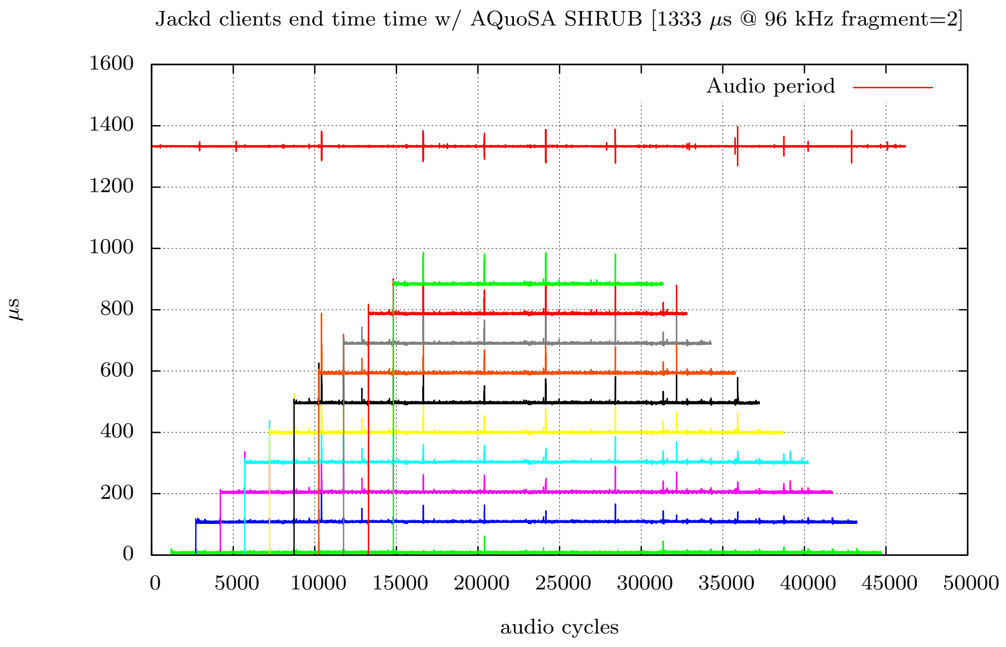

Figure 4.18: *Clients end time with AQuoSA/SHRUB and 10 clients, using the AQuoSA period doubled with respect to JACK period length.*

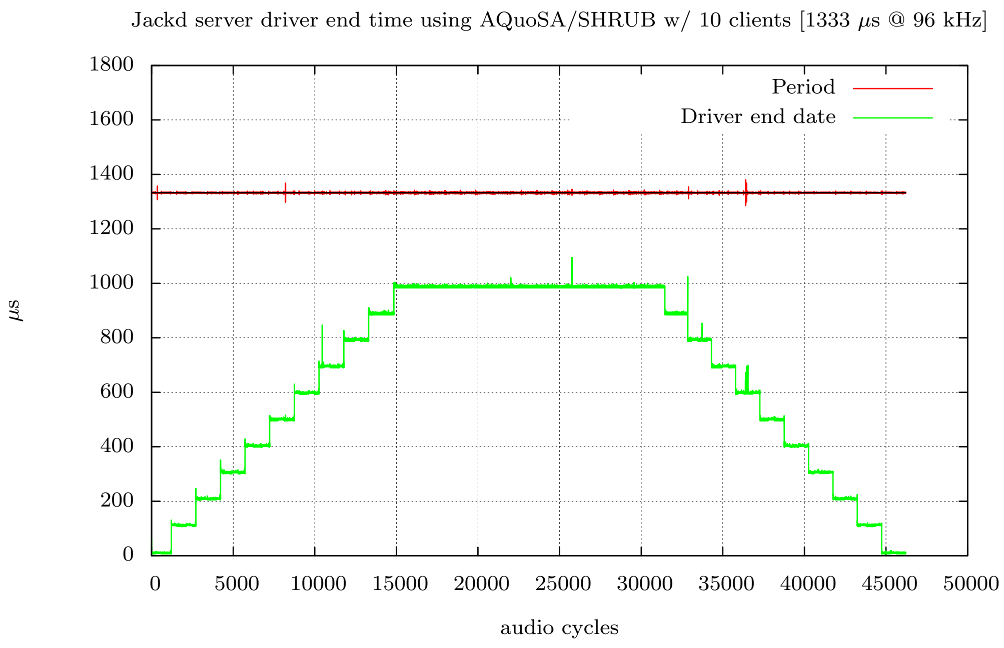

Figure 4.19: *JACK perturbed driver end and period with 10 clients, with 2 rt-app tasks running in background using AQuoSA with SHRUB for both JACK and rt-app.*

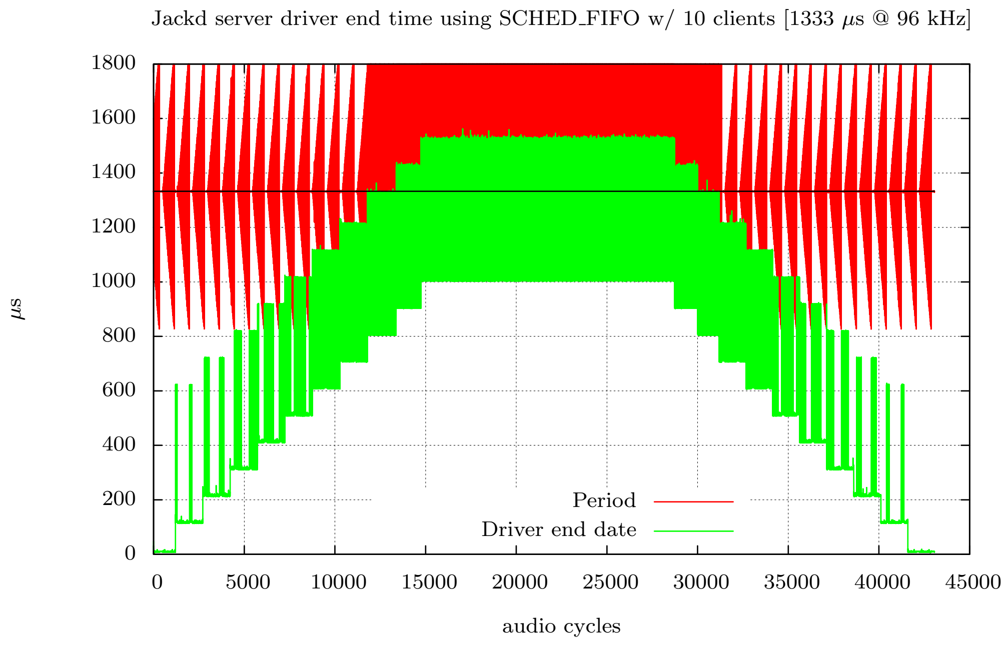

Figure 4.20: *JACK perturbed driver end and period with 10 clients, with 2 rt-app tasks running in background using SCHED_FIFO for both JACK and rt-app.*

	Min	Max	Average	Std. Dev	Drv.End Min	Drv.End Max
AQuoSA	650.0	683.0	666.645	0.626	6.0	552.0
linux-rt	629.0	711.0	666.263	1.747	6.0	602.0
Linux	621.0	1369.0	666.652	2.696	5.0	663.0

Table 4.3: *Period and driver end measures with 667μs latency and 10 clients*

4.6 Very low latencies

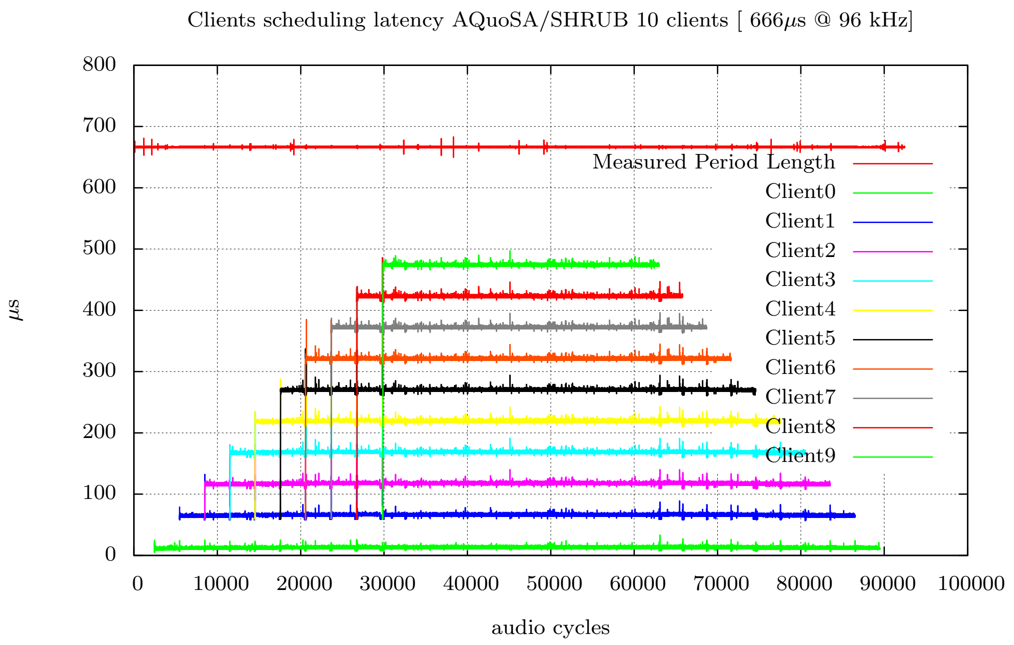

Figure 4.21: JACK running with AQuoSA and SHRUB with a buffer of 64 frames per periods at 96 kHz, resulting in a latency of 667 μs. The AQuoSA server period is set to 2001 μs, that is three times the JACK period.

This last test was performed to explore a loaded scenario with very low latencies. The JACK latency is at the minimum reachable with the hardware used, setting 64 periods per buffer at 96 kHz sample rate, resulting in 667 μs total latency introduced by the JACK server. 10 client are connected to the JACK server in a similar scenario of the one described in section 4.5 on page 79, but the total utilization is lowered to as none of the kernel and schedulers were able to reach 75% utilization under this condition.

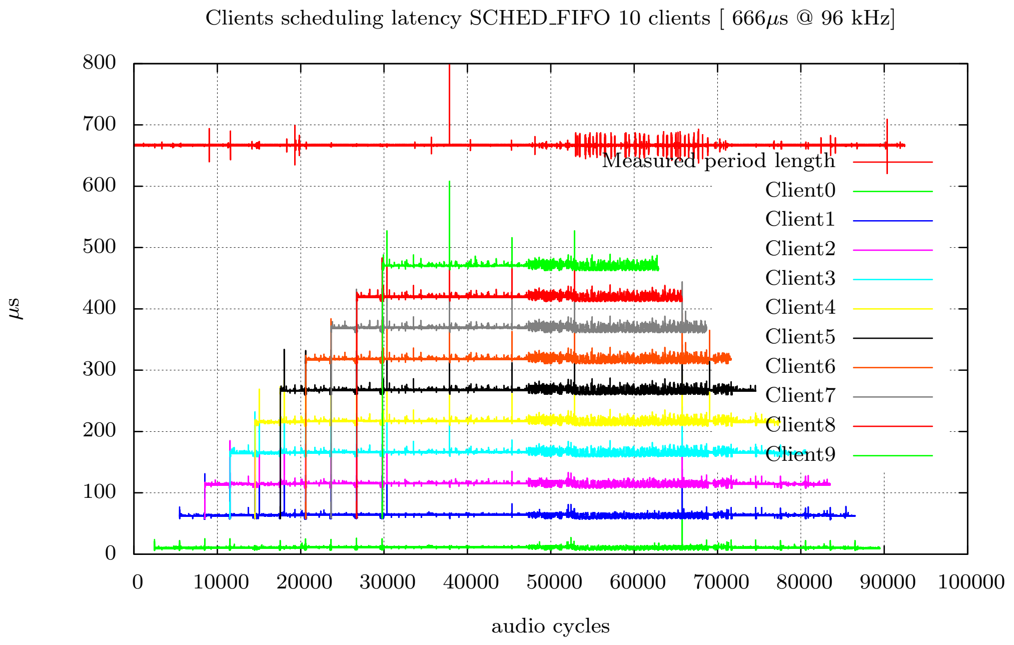

Figure 4.22: JACK running with SCHED_FIFO with a buffer of 64 frame per periods at 96 kHz, resulting in a latency of 667 μs.

Both linux-rt and AQuoSA managed to avoid any xrun, while on Linux SCHED_FIFO some were reported. Table 4.3 on the preceding page shows the period and driver end time statistics about all three runs, while the graphs represented in Figure 4.21 on the previous page, 4.22 and 4.23 on the following page show the clients end date and audio driver trend.

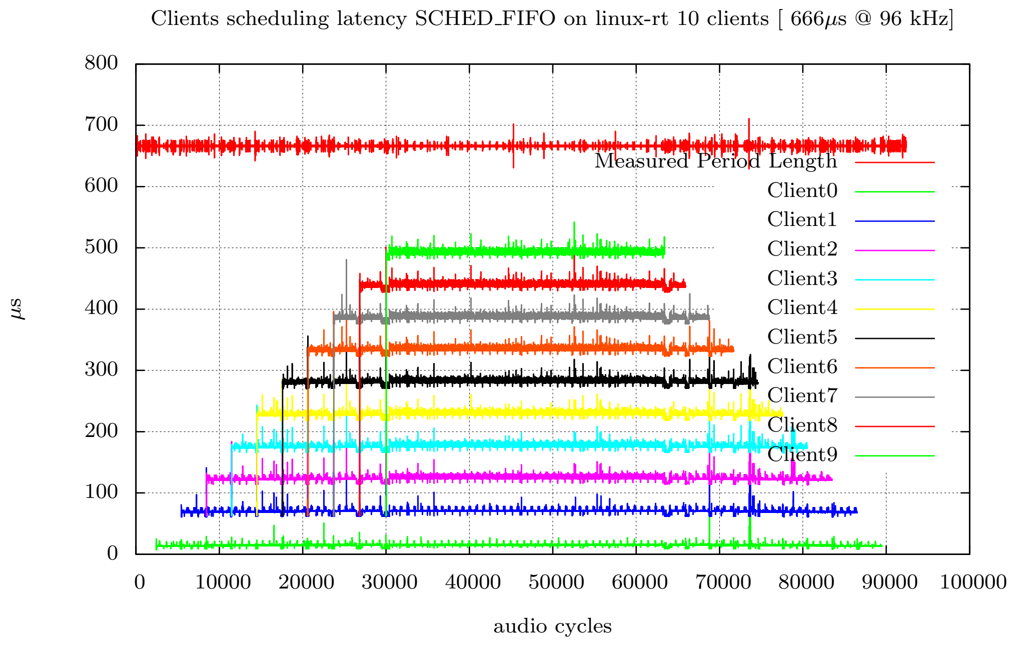

Figure 4.23: JACK running with SCHED_FIFO on linux-rt kernel with a buffer of 64 frame per periods at 96 kHz, resulting in a latency of 667 μs.

Chapter 5

Conclusions and Future Work

In this work JACK has been modified to leverage the Resource Reservations provided by the AQuoSA framework. Library and utilities where also developed to test its behaviour on standard linux kernel using POSIX realtime extensions, on linux-rt patchset and on the AQuoSA real-time scheduler.

Various approaches to were evaluated and considered, from a theoretical point of view and from a practical point of view in implementing them. The final chosen approach was to directly modify JACK server and client libraries to provide support for feedback scheduling and resource reservations: this allowed the JACK clients to run unmodified with the patched server and libraries. The `libgettid` library was written to solve some issue raised by differences on how threads are identified in the Linux kernel and in the POSIX thread library . Finally a jack client application and a real-time periodic application were written to load the system to evaluate performances of tested schedulers.

Various comparisons were done between various mode of operation (synchronous and asynchronous), various hardware (`HDA` and `ICE1712`), various kernel ("vanilla" linux, AQuoSA and linux-rt), and finally with multiple clients with or without other realtime loads.

In these tests the AQuoSA scheduler and, in general, the whole resource reservation mechanism has proved to make JACK capable of very low latencies even with low latencies and to be capable, as well as to make JACK coexists with other real time loads. Since linux-rt has better and more stable timings, could be of interest to have a EDF/CBS implementation within this patchset, as this setup may probably be the best of the two world.

Another modification that is interesting for future works is to make one reservation per client, instead of having a single reservation for all jack client and server threads. This can possibly open the scenario of completely remove the FIFO-based machinery in favour of inter-process semaphores and access protocols such as the Bandwidth Inheritance Protocol. Moreover, having one reservation per client could allow to explore the recent SCHED_DEADLINE patchset, which adds a new scheduling class that use EDF to schedule processes.

Code listings

5.1 A simple JACK client

```c
#include <jack/jack.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>

jack_port_t *in, *out;
jack_client_t *client;
int end;

/* the audio process function: this simply copies input data
   to output buffer. The size of buffers can change between
   calls, and it's passed as the nframes parameter.
*/
int process(jack_nframes_t nframes, void *arg)
{
    jack_default_audio_sample_t *in_buf, *out_buf;
    /* get input and output buffers */
    in_buf = jack_port_get_buffer(in, nframes);
    out_buf = jack_port_get_buffer(out, nframes);
    memcpy (in_buf, out_buf, nframes * sizeof(
              jack_default_audio_sample_t));
    return 0;
}

```

```
void shutdown(jack_status_t code, const char* reason, void *arg)
{
    printf("Server closing, exiting...");
    end = 1;
}
void terminate(int sig)
{
    printf("Signal received, exiting ... \n");
    /* deactivate the client, unregister ports and close */
    jack_deactivate(client);
    jack_port_unregister(client, in);
    jack_port_unregister(client, out);
    jack_client_close(client);
    end = 1;
}

int main(int argc, char** argv)
{
    /* register the client with the JACK server: if it's not
       running, it will be autostarted */
    jack_status_t status;
    const char *server_name = NULL;
    char *client_name = "simple_client";
    client = jack_client_open(client_name, JackNullOption,
                              &status, server_name);
    /* set the audio processing callback. It will run in
       its own realtime thread */
    jack_set_process_callback(client, process, NULL);
    in = jack_port_register(client, "in",
                            JACK_DEFAULT_AUDIO_TYPE,
                            JackPortIsInput, 0);
    out = jack_port_register(client, "out",
                             JACK_DEFAULT_AUDIO_TYPE,
                             JackPortIsOutput,0);
    if ( (in == NULL) || (out == NULL))
    /* handle error, as no more ports are available */
    {}

    /* install the server shutdown callback */
    jack_on_info_shutdown (client , shutdown, 0);

    /* install signal handlers */
    signal(SIGQUIT, terminate);
    signal(SIGTERM, terminate);
    signal(SIGINT, terminate);

    /* tell the server we are ready to process data */
    jack_activate(client);

    /* loop until shutdown() is called by the server or a
       signal is received */
    end = 0;
    while (end == 0);
         sleep(1);
    return 0;
}
```

Bibliography

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 4, Washington, DC, USA, 1998. IEEE Computer Society.

[2] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Systems*, 29, 2005.

[3] ALSA. (advanced linux sound architecture) homepage. `http://www.alsa-project.org`.

[4] Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, and Scuola Superiore S. Anna. Hierarchical multiprocessor cpu reservations for the linux kernel .

[5] Tommaso Cucinotta, Luca Abeni, Luigi Palopoli, and Fabio Checconi. The wizard of os: a heartbeat for legacy multimedia applications. In *Proceedings of the 7th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, 2009.

[6] Takashi Iwai. Sound systems on linux: From the past to the future. In *UKUUG Linux 2003*, 2003.

[7] JACK. Jack website. `http://www.jackaudio.org`.

[8] Giuseppe Lipari. Greedy reclamation of unused bandwidth in constant-bandwidth servers, 2000.

[9] C.L. Liu and James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment, 1973.

[10] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. Aquosa—adaptive quality of service architecture. *Softw. Pract. Exper.*, 39(1):1–31, 2009.

[11] Luigi Palopoli, Luca Abeni, Tommaso Cucinotta, and Sanjoy K. Baruah. Weighted feedback reclaiming for multimedia applications. In *In ESTI-media*, pages 121–126, 2008.

[12] RTWiki. Rtwiki homepage. `http://rt.wiki.kernel.org`.

[13] D. Fober S. Letz, Y. Orlarey. Jack audio server for multi-processor machines. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 1–4, 2005.

[14] R.Moret S.Letz, N.Arnaudov. What's new in jack2? In LAC, editor, *Proceedings of Linux Audio Conference 2009*, 2009.