Diplomarbeit

Implementation and quantitative analysis of a real-time sound architecture

Michael Voigt

16. April 2009

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Michael Roitzsch

Acknowledgments

First of all, I would like to thank Professor Dr. Hermann Härtig for the opportunity to write my diploma thesis about an intriguing topic like this and for creating a chair that has spawned all these innovative and promising projects. Thanks also to all the members of staff—I have never experienced a single one of you unhelpful. Exceptionally, I am grateful to my advisor, Michael Roitzsch, and to Martin Pohlack, who joined Michael Roitzsch for a while in advising me. Surely, I didn't make it always easy for you. I am thankful for your assistance in scientific questions, implementation problems, your patience—and for your colleagueship.

Furthermore I would like to thank the Linux Audio Developers community for creating such a wonderful sum of free software. Especially, I am grateful to Stéphane Letz for his helpfulness.

To the students in the office: We had a really great time, I deeply enjoyed the working atmosphere in the laboratory, and the interesting discussions during the coffee breaks.

My friends and my two brothers: Simply thanks you are there.
Last but not least I would like to thank my parents for their support.

Remit

The objective of this thesis is to design and implement a comprehensive sound architecture, allowing users to create a virtual sound studio. Linux features existing free solutions related to the task, but without guarantees on timing behavior of individual components or the entire virtual studio. The student should analyse the existing solutions and reuse them, if possible. The architecture must enable the wiring of various independent components like mixers and filters using a well-defined interface to form an audio streaming graph. To support real-time operation, the work shall use the DROPS research system as a foundation. The ALSA sound driver ported to DROPS by Mr. Voigt in earlier work should act as a source and sink of the graph.

In the course of this thesis, the student shall develop a methodology to quantitatively characterize single audio components. Based on that, the derivation of relevant timing properties for the entire streaming graph shall be possible (e.g., maximum latency, CPU time demand). The theory on Jitter Constrained Streams provides a potential starting point. The entire architecture and methodology shall be demonstrated with an example use case. The evaluation should verify the real-time properties of the running system and the correctness of computed streaming graph parameters.

Contents

1 Introduction

When performing music, correct timing is essential. Consequently, it is crucial as well for professional music production systems, which for the most part are digital systems nowadays. Musicians and sound engineers do not only want their tools to offer them a rich set of features, to be flexible, customizable, and intuitively and productively usable. They also expect from their tools to work reliably regarding their real-time behavior—just as they are used to it from the analog world: An analog mixer or multi-track recording system does not suddenly stop, click, or lag, because it has decided that it is time to check for system updates or the hard disk index needs to be rewritten. But audio software running on a general purpose desktop system does not provide the same level of reliability as analog equipment, when it comes to real-time behavior.

There are, however, digital solutions available on the market using a combination of specialized hardware and software that do meet the real-time needs of the professional user. But they are extremely expensive and do not offer the same degree of flexibility as a desktop computer. It is the goal of this thesis to bring these two worlds of digital audio recording and editing—the real-time reliability of the specialized solutions on the one side and the flexibility of a general purpose desktop system on the other side—one step closer together.

2 Requirements

It is the goal of this project to bring the two worlds of digital audio production described in the introduction one step closer together. To define what this means precisely, in the first section of this chapter an analysis of this long-term goal is given. As this is a fairly visionary aim, the project documented here can only make a small contribution to it. However, this definition should not be dismissed, because it acts as a guideline for this thesis. Section 2.2 enumerates the concrete, immediate results to be reached during the project.

2.1 Long-term vision

A professional audio workstation should give its user the opportunity to work pleasantly and reliably at runtime. To let the user work comfortably, the system has to react without a noticeable latency to his actions. Furthermore, the user wants the system to perform its jobs reliably: without interruption or clicking noises produced by buffer overruns or underruns. Therefore, professional audio processing can be considered to have real-time requirements.

Unfortunately, there is a conflict between these two goals, as visualized in Figure 2.1: Buffering can be used to compensate jitter in the arrival time of data packages [39, 36]. But the lower the desired maximum latency, the less data may be buffered, and consequently the harder it is to guarantee that all packages of audio samples can be delivered in time.

Hence, a balance between these conflicting requirements has to be found. The key to a good trade-off is high system predictability, because the better the predictability of the system and its components, the easier it is to fulfill both: Low latency and real-time requirements. Subcection 2.1.1 specifies the latency requirements, and the real-time requirements are clarified in Subsection 2.1.2.

2.1.1 Latency aspects relevant to professional audio systems

The information presented in this subsection is the outcome of miscellaneous studies, which are overviewed and cited in Paragraph two of [44].

The way the human nervous system perceives external events and reacts to them is very complex and shows nontrivial behavior in many situations. Thus, not

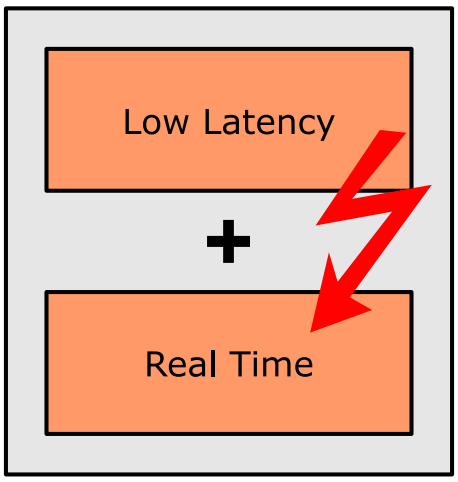

Figure 2.1: There is a conflict between low latency and real-time requirements: The lower the desired latency, the harder it is to guarantee timing constraints to be fulfilled.

every aspect of latency is equally important. There are three aspects of latency that are relevant to professional digital audio processing:

2.1.1.1 Processing latency

The processing latency is the time the system needs to respond to an event from the outside world, such as a keypress on a MIDI keyboard or the input audio samples entering the system at a constant rate.

The precise processing latency value still tolerable is highly dependent on the music instruments used in the recording, more specifically: the attack time of the instrument's sound. Even in professional chamber music deviations of up to 50 ms in onset time are not unusual. On the other hand, when it comes to rhythm, under certain circumstances human beings are able to detect timing discrepancies as low as 4 ms on a subconscious level.

Commercial digital all-in-one recording systems can achieve processing latencies below 2 ms. As a rule of thumb, a processing latency below 10 ms can be given as a desirable value.

2.1.1.2 Latency jitter

Human beings are able to adapt to processing latencies to a certain extent, as long as it is possible to anticipate them correctly. For example, the time between pressing a key of a piano and hearing the first wave cycles of that tone can take up to 100 ms—and still the piano player feels comfortable while placing the tones extremely accurately in time. Consequently, keeping latency jitter as low as

possible is much more important than a low processing latency itself. Latency jitters beyond 1 ms are not acceptable.

2.1.1.3 Inter-stream deviations

Since timing deviations as low as 20 μs between two channels of a stereo signal can be used by the human ear as cues to determine spatial positioning, and annoying comb filter effects can occur at any delay, anything else but sample-accurate synchronization between different audio streams is not tolerable.

2.1.2 Real-Time

There are competing definitions of the term *real-time*. Its colloquial usage differs from the accurate scientific definition. Many people, who refer to a task as beeing executed *in real-time* on their computer, mean it runs *at runtime*. Although both aspects are closely related, they are not equivalent: A modern standard desktop computer with a general purpose operating system like GNU/Linux, Mac OS X or Windows can decode and playback a video or soundfile without dropouts at runtime—in most of the cases. But still these systems are no real-time operating systems, because they cannot *guarantee* that every frame reaches the hardware buffer in time, or at least that the number of dropped frames per second does not exceed a certain limit. They solve this task with overprovisioning: Modern systems simply have so many CPU ressources available that, if the system load is low, it is likely that a sufficient number of frames is finished before their deadline. But if the user starts enough other tasks in parallel with the playback process, the situation immediately changes. While this is not harmful to a consumer listening to music or watching movies, it is not acceptable for a sound engineer or a professional musician working with the digital sound system—possibly even performing live on stage.

A real-time operating system (RTOS) in the scientific meaning can be characterized as a multitasking operating system that guarantees for a well-defined model of task sets to meet the timing constraints of the tasks in every case. A method called *admission* must be available for the system. The admission routine checks—either offline or online—for a specific set of tasks and their associated timing constraints whether the system is capable of fulfilling the timing constraints reliably or not. A typical example for a real-time task is a *periodic task* that consists of jobs occurring at a constant rate with fixed relative deadlines to be met. The timing constraints can be hard, that means all deadlines must be met, or soft—for example, only an assured minimum percentage of jobs has to be finished before their deadlines. For profound information about real-time systems I refer the reader to [50].

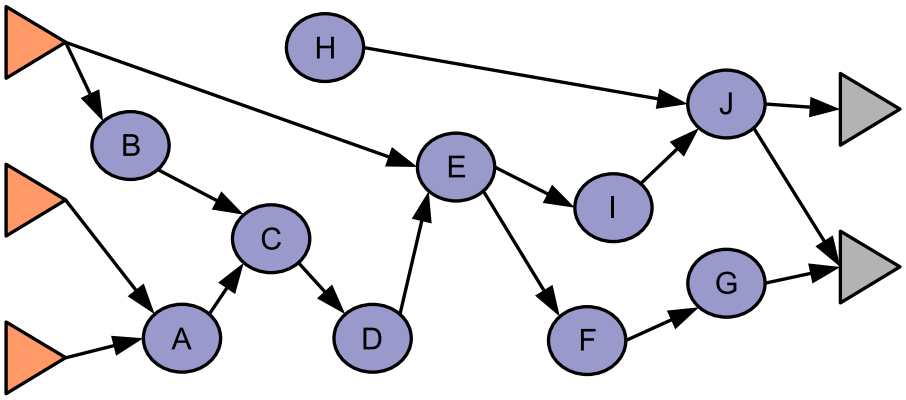

Figure 2.2: Streaming graph.

2.2 Objective of the thesis

As it can be seen from the remit, the objective of this thesis can be divided in two main parts:

2.2.1 Porting an open source solution to DROPS

Existing open source sound architectures are to be analyzed and the most appropriate one should be ported to DROPS (see Section 3.1). The chosen architecture should allow it to connect various components such as effect filters, mixers, virtual instruments or signal generators across address space boundaries to form a streaming graph like the one visualized in Figure 2.2.

This assignment is aligned with the long-term goal given in Section 2.1 in the following way: First of all, in contrast to Linux, DROPS does offer hard real-time scheduling as well as soft real-time scheduling algorithms. It has been shown that a microkernel-based design cannot only help to increase the system security, but that it is also suited well for real-time programming [54]. Secondly, the Linux kernel has been ported to DROPS to run as a userland task on top of its L4 microkernel Fiasco in a paravirtualized manner [42, 43, 40]. This server task is called L^{4}Linux and makes it possible to execute even unmodified Linux binaries side by side with native DROPS processes without compromising the real-time guarantees given by the system. This design promises to bring the two worlds described in the introduction together on one single DROPS system: The real-time critical audio processing parts can be extracted from conventional Linux audio applications such as Ardour and run as a native real-time tasks, while the L^{4}Linux side provides the usability and flexibility of a mature general purpose desktop operating system.

2.2.2 Development of a client characterization methodology

Part two of the task is to develop a methodology to quantitatively characterize the runtime behavior of the elements in the streaming architecture chosen in the first part. It should be possible to draw conclusions about the whole graph from the characterization of the single elements: Can this graph be executed reliably on a real-time system? What is the maximum latency? The methodology should be exemplified and validated with an example.

3 Foundations

This chapter introduces the different projects this project is based upon. It starts with the Dresden Real-Time Operating Systems Project in Section 3.1, since it is the environment the new sound architecture should be integrated in. The DROPS port of the Advanced Linux Sound Architecture (ALSA) should act as the driver of the sound server and is therefore presented in Section 3.2. The chapter ends with a comparison of the available free solutions that could be possibly reused in this project.

3.1 TUD:OS and DROPS

It is the goal of the Dresden Real-Time Operating Systems Project (DROPS) at the operating systems chair of the Technische Universität Dresden to continuously develop a microkernel-based multi-server research operating system. The system itself is called DROPS as well—or TUD:OS alternatively. The system serves as a playground for practical investigation of new ideas in operating system design. All developments at this chair are related to it. A more detailed overview about TUD:OS can be found for example in [53] or in [59].

3.1.1 The microkernel approach

In contrast to an operating system architecture with a monolithic kernel, it is the goal of a microkernel-based design to keep the amount of code that runs in the privileged kernel mode of the CPU as small as possible. All the elements that do not necessarily require the kernel mode—such as device drivers or file systems—should be kept out of the kernel and run as server tasks in the user mode of the CPU on top of the microkernel. While with a monolithic design applications would access these system services by system calls directly, with a micro-kernel based architecture the communication between applications and system services is mapped to the inter-process communication (IPC) primitives of the microkernel. The difference between a monolithic and a microkernel-based multi-server operating system is illustrated in Figure 3.1. The German computer scientist Jochen Liedtke, who defined the L4 microkernel interface and developed its first implementation, puts the microkernel design paradigm as follows:

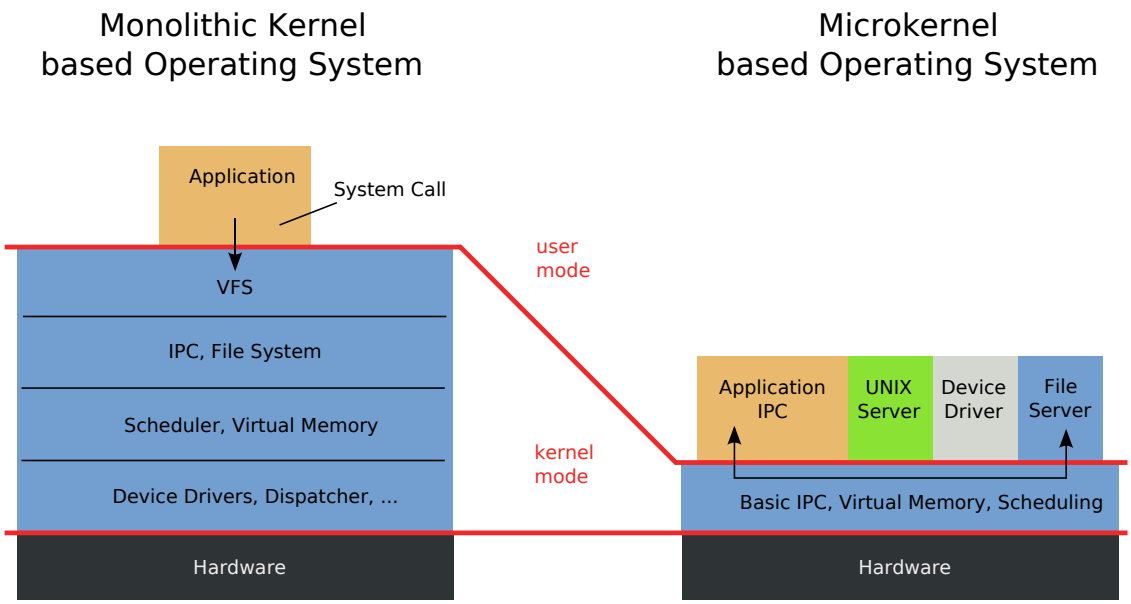

Figure 3.1: According to a microkernel-based design, the amount of code that runs in the privileged kernel mode of the processor should be kept as small as possible. Illustration taken from Wikimedia Commons. License: public domain.

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality. [48]

The original L4 microkernel interface was reimplemented many times in a various set of programming languages, and now the name L4 applies to this whole set of microkernel implementations, as well as to their different kernel interfaces. The microkernel of DROPS is called Fiasco [41] and it belongs to this L4 microkernel family. Fiasco has been developed entirely at TU Dresden's operating systems chair and is written in the C++ programming language.

3.1.2 L4Env

On the one hand, the microkernel approach permits a much more modular and flexible operating system architecture than a monolithic design. But on the other hand, it confronts the microkernel developer with the problem that a microkernel by design does not offer a large set of features.

For this reason L4Env [57], the L4 environment, was implemented. L4Env provides a common subset of standard servers and libraries offering a higher level abstraction and a richer set of features than the microkernel itself. In addition, it also contains development tools—such as the DROPS interface description language compiler DICE [31]. The L4Env servers that are required for this project are explained in Section 4.3.2.

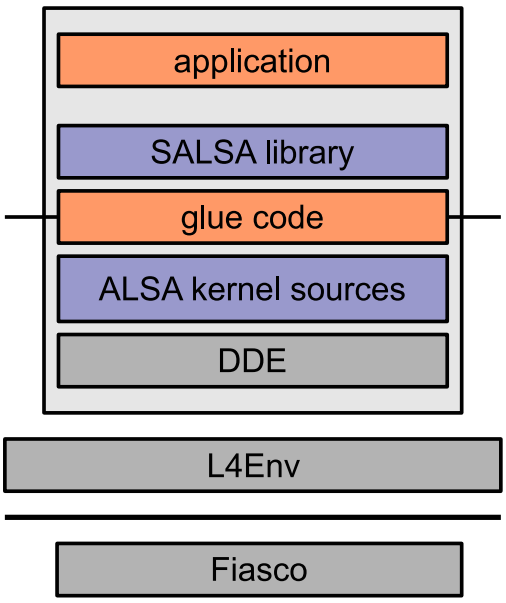

Figure 3.2: The ALSA port design, which finally was chosen: All components, including the application, are linked together to one single block.

3.2 ALSA port

Prior to this project, I did an internship at TU Dresden's operating systems chair. During this internship, I ported the the Advanced Linux Sound Architecture (ALSA, [2]) to DROPS (see Section 3.1). Because this port builds the foundation for this project, a short overview of the ALSA port design is given here. Implementation details can be found in appendix A.

The design finally chosen is visualized in Figure 3.2: All elements are linked together with the application to one single block in userland. The ALSA kernel sources make up the core component of the port. They could be taken over unchanged, because our device driver environment (DDE, see Section 2.2.3 in [58]) provides the usual linux kernel environment to them.

The ALSA architecture does not adhere to the standard UNIX file based `read()/write()` kernel-userland interface. Instead, besides of `open()` and `close()`, nearly all calls to the ALSA driver architecture are custom-defined `ioctl()` system calls. It is intended by the developers of ALSA not to access their driver architecture directly, but to use an ALSA userland library. Hence, a userland library had to be ported to DROPS as well. I decided against the common ALSA library in favor of the lightweight Small ALSA Library (SALSA-Lib, [19]), because it is less complex and therefore easier to understand, and it does not depend on a UNIX/Linux userland (configuration files, plug-ins in userspace) to work. A small file system emulation layer has been written to let SALSA-Lib be linked against the ALSA kernel part.

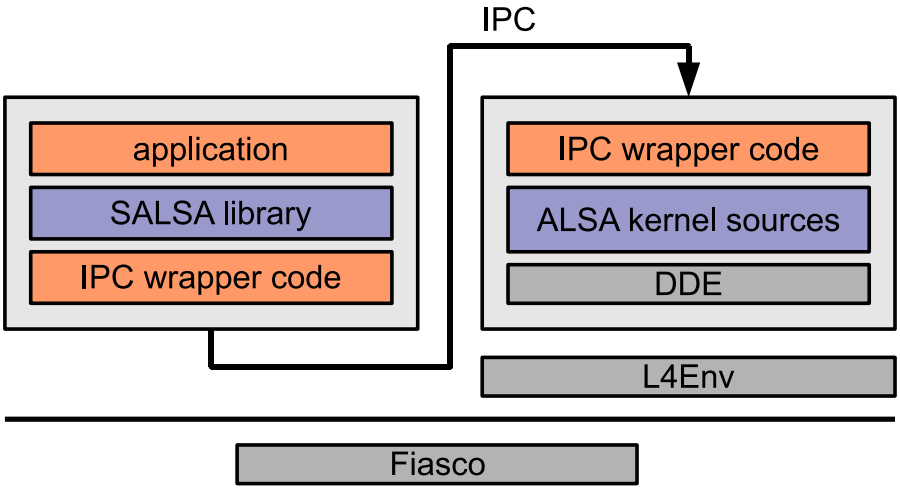

Figure 3.3: An alternative design: The application runs together with the Small ALSA library (SALSA-Lib) in its own address space. An emulation layer forwards the system calls from the SALSA library to the ALSA driver server via inter-process communication principles.

At first glance, it might look like a bad idea to link all these components together to one single big block. Creating a sound server and putting the address space boundary at the place where the former system call interface has been (see Figure 3.3) or using the ALSA API as IPC interface would be a more natural approach from a microkernel developer's viewpoint. There are three main reasons why the solution of Figure 3.2 was selected:

1. ALSA uses the `ioctl()` system call not only to transfer data directly between the kernel and the userland, but also passes pointers. The ALSA kernel part then either copies the data from or to user space with the `copy_from_user()` or `copy_to_user()` function—or the memory is shared in `mmap` mode. In any case it would be cumbersome to emulate this behavior across two distinct address spaces.

2. Neither the ALSA library API nor the kernel-userland interface of ALSA can be considered well-suited for multiplexing.

3. It is our goal to implement a comprehensive sound architecture with a more abstract API than the ALSA library. Seen from this angle, it is not useful to add this additional layer of indirection, as ALSA will only act as the driver of this sound system.

3.3 Comparison of existing free solutions

This section outlines the different available solutions, which could be used for this project, and concludes in Subsection 3.3.5 why Jackdmp has been chosen. For practical reasons, only those solutions are of interest, which can be ported to another platform without an extra permission and whose underlying knowledge is completely available. Therefore, this overview is restricted to free and open source projects.

3.3.1 The Jack Audio Connection Kit and Jackdmp

The Jack Audio Connection Kit (JACK, [9]) is an audio server designed for professional audio recording and production. It is available for many GNU/Linux distributions, for FreeBSD and Mac OS X. The project was started by the lead developer of the sequencer Ardour [4], Paul Davis, out of the need for a comfortable way of low-latency and sample-synchronized data exchange between audio applications on Linux. JACK has a server-based architecture and allows different audio applications to be tied together to a virtual sound studio across address space boundaries. These applications register themselves on the JACK server as *external clients*. In addition, Jack offers an interface for *internal clients*, which run inside of the server's context as shared objects.

Whereas JACK is written in the programming language C, under the direction of Stéphane Letz from the Centre national de création musicale Grame[1] a completely API compatible reimplementation of JACK—called Jackdmp—has been developed in C++. Besides its object oriented architecture, Jackdmp has removed limitations in the current JACK design: First and foremost, it enables JACK based streaming graphs to benefit from multiprocessor machines. Furthermore, it uses advanced lock-free techniques to access shared memory data. Jackdmp runs on GNU/Linux, Solaris, Mac OS X and Windows and is designated to become the official version 2.0 of JACK. A detailed description of JACK and Jackdmp follows in Chapter 4.

3.3.2 GStreamer

GStreamer [15] is a pipeline based multimedia framework and developed as a part of the freedesktop.org [7] project. It is similar to JACK in the way that it allows wiring many single elements to a multimedia pipeline. But unlike JACK, GStreamer is designed to be a tool set for the development of consumer multimedia applications rather than a complete audio architecture. If, for instance, a software media player should be developed, GStreamer can be used to synchronize the

[1] http://www.grame.fr/

audio and video output and to equip the player with an extensible set of already existing decoder plugins.

A key difference in the architecture of GStreamer and JACK is that GStreamer does not have a server process, with which applications register themselves. The core of GStreamer, which takes care of the scheduling and puts up the GStreamer streaming infrastructure, is directly linked as a library to the application. The pipeline components have to reside in a certain directory as shared library objects. At program start, this directory is crawled for such plugins, which then can be loaded into the program's address space.

The second difference between GStreamer and Jack is that the former allows connecting plugins only within a single address space. The purpose of GStreamer is more focused on the plugin part, while the primary goal of JACK is connecting tasks across address space borders.

3.3.3 User-oriented desktop sound server

The three most common open source desktop sound servers aRts, ESD and PulseAudio are portrayed in this subsection. One main reason for the existence of these desktop sound servers is that both the ALSA library and the Open Sound System [14] started to offer multiplexing of the audio hardware at a late state in their development.

3.3.3.1 Enlightened Sound Daemon

The Enlightened Sound Daemon (ESD or EsounD, [6]) is the sound server of the GNOME desktop environment [26] and the Enlightenment window manager [23]. Network transparency is among its special features. ESD is in productive use on many desktop systems, but it misses a way to share audio data between applications as well as synchronization mechanisms suitable for professional audio recording.

3.3.3.2 aRts

The analog Real time synthesizer (aRts, [21]) was the standard sound server of the version 2 and version 3 series of the K Desktop Environment (KDE, [24]). It is one of the notable exceptions amongst desktop sound servers that does provide inter-application routing. But like ESD it does not offer a way to control, when a sound frame handed over to the server reaches the sound card. Besides being a sound server, aRts also has an integrated sound synthesizer, which is used for system sound generation. The analog Real time synthesizer is deemed not to be very stable and its development is discontinued.

3.3.3.3 PulseAudio

PulseAudio is a new, ambitious audio server, compared to aRts and ESD. It is designed as a drop-in replacement for ESD with the intention to solve the audio incompatibility problems on the Linux desktop that result from the massive diversity of audio APIs: Not only is PulseAudio fully compatible with ESD, it also provides direct connection libraries for ALSA, xine [30], MPlayer [13], XMMS [29], Audacious [5], GStreamer (see Subsection 3.3.2) and Libao [11]; as well as ALSA and OSS [14] wrapper drivers, which forward native calls to the driver APIs back to the PulseAudio server in userland. PulseAudio has a modular structure, which allows further adapters to be added easily. The specialities of it further include per-channel volume control, network transparency and Zeroconf [63] support.

Although it is the aim of PulseAudio to provide Linux with "a common solution that works on the desktop, in networked thin-client setups and in pro audio environments, scaling from mobile phones to desktop PCs and high-end audio hardware" [52], PulseAudio is not intended to be a competitor to JACK—at least not in the short run. To allow people use professional audio software while still being able to use desktop multimedia applications on the same system, the developers of PulseAudio rather aspire a tight integration with JACK. For this purpose a JACK sink and a JACK source PulseAudio module have been developed.

3.3.4 Audio APIs

In this subsection I refer to APIs (Application Programming Interfaces) as projects, which primarily define an interface, while audio architectures like JACK or GStreamer provide a complete implementation of the APIs they define.

There are three common plugin APIs in the Linux audio world: LADSPA, DSSI and LV^2, which are presented in the following subsections. Similar to the Virtual Studio Technology (VST, [62]) plugin API by Steinberg, these plugins are shared objects, which run in the context of the host application. The host application usually is a music sequencer and the plugins can be either audio filters or virtual instruments.

Subsection 3.3.4.4 introduces to the multimedia API Phonon. The cross-platform audio API PortAudio [33] is not regarded in this document, since its use and acceptance are limited so far.

3.3.4.1 LADSPA

The Linux Audio Developers Simple Plugin API (LADSPA, [1]) is the first professional plugin audio API of the free software scene. It features interfaces for the plugin to connect ports to the host application. Over these ports the plug-in can exchange audio data with the host program. In addition to audio ports

LADSPA contains control ports, which can be used by the client plugin to export parameters to the host application. The main program should provide a way to set these parameters—for example a graphical user interface. Because LADSPA lacks functions or data structures to send instructions such as MIDI commands to the plugin, only filter plugins but no virtual instruments can be implemented with LADSPA.

3.3.4.2 DSSI

The Disposable Soft Synth Interface (DSSI, pronounced "dizzy", [22]) is based on LADSPA and extends it with MIDI support. Indeed, one part of the data structure representing a DSSI plugin is a LADSPA plugin, which takes care of the audio data handling.

3.3.4.3 LV^2

LV^2 is a successor to both LADSPA and DSSI. Not only filter plugins but also virtual instruments can be written with it. But instead of adding MIDI support directly like DSSI, LV^2 is focused on extensibility: The static data about the plugin—such as the number and type of ports—is not located in the shared object's binary, but in a separate Resource Description Framework [17] file. New port types can be defined in this file easily and there are already standardized extensions—for example one that adds MIDI support.

3.3.4.4 Phonon

Phonon [25] is the new multimedia API of the KDE [24] version 4 series. Native KDE 4 applications may only use this API to interact with multimedia hardware. This convention—together with others—is intended to assure source code compatibility of KDE 4 applications across operating systems. Amongst others there are Phonon backends based on xine [30], GStreamer, VLC [28] and MPlayer [28] for UNIX-like systems. For Mac OS X exists a special backend for Quicktime [3], for Windows exists a backend, which uses DirectX. To enable cross platform audio and video playback Phonon is also part of the Qt framework since the release of version 4.4.

3.3.5 Conclusion

To draw a conclusion from the preceding subsections, the reasons why the JACK architecture and particularly Jackdmp was chosen as the solution to be used in this project are summarized here:

1. The sound architecture of JACK allows the wiring of independent elements to a virtual recording studio across address space borders. No other option offers a similarly convincing solution.

2. The JACK sound server is the de facto standard in the world of professional free software audio recording and editing.

3. JACK has a generic and abstract client API, which does not bother the programmer unnecessarily with details about the underlying hardware.

4. The Jack Audio Connection Kit project does not only define an API, but also comes up with a reference implementation.

5. JACK synchronizes its clients' channels on sample accuracy—the best possible precision that can be reached on the software part in a digital audio system.

6. Jackdmp was chosen in favor of the first JACK implementation, because first of all it has a well-structured object-oriented software architecture that makes porting much more comfortable than it would be with the original implementation of the JACK API. Secondly, it removes limitations of the first version in C: It is capable of benefiting from multiprocessor machines, it uses lock-free techniques for shared memory access and it separates the real-time and the non real-time parts of the clients into two threads. Separating these two parts has the benefit that the non-real-time part cannot disturb the timing behavior of the real-time part.

4 Design

Section 3.3.5 points out, why the JACK sound architecture—and particularly Jackdmp—were selected for this project. As usual when porting applications, a huge part of the design is transferred from the original implementation. For this reason, first the JACK sound architecture is explained in general and afterwards the design of Jackdmp on DROPS is presented in Section 4.3. The principles of the JACK sound architecture are more obvious in the original design, since it is simpler than the design of Jackdmp. Therefore, the design of the C version is explained in Section 4.1, followed by the differences between JACK and Jackdmp in Section 4.2.

4.1 The JACK design

This section overviews the JACK design. It starts in Subsection 4.1.1 with the basic design paradigms of JACK and continues in Subsection 4.1.2 with a detailed description of the JACK engine cycle. It ends in Subsection 4.1.3 with a characterization of the JACK design from an external client's viewpoint.

4.1.1 Design paradigms

As already described in Section 3.3.1, JACK makes it possible to tie various internal and external clients together to a virtual sound studio. The JACK sound architecture is based on the following four design axioms:

Unique frame format: In the context of digital audio technology, a *frame* of an audio stream with n channels is the n-tuple of the channels' samples at the same moment in the stream. JACK supports one single audio frame format: 32 bit floating point numbers with an absolute value equal to or less than one. Moreover, all frames have to be monaural. Thus, a sample and a frame is equivalent in JACK. All applications need to convert their audio data to this format in order to exchange it with other JACK clients or the driver. Allowing mono frames only is not a restriction, since multi-channel streams can be mapped to multiple mono streams.

Using a single frame format has the big advantage that format negotiations cannot occur. Choosing normalized real numbers additionally makes it possible to increase the frame's accuracy—for example from 32 bit to 64 bit—without breaking source code compatibility.

Shared memory data exchange: JACK uses shared memory buffers as a zero-copy mechanism to transfer audio data (or other streaming data like MIDI commands) amongst the involved clients. These data connections are called *ports* in JACK.

Callback-based API: Common UNIX sound servers like ESD (cf. Section 3.3.3.1) or aRts (cf. Section 3.3.3.2) are typically based on an active data delivery model, in which the client can call the server whenever it wants to deliver as much data as favored. JACK on the contrary has a callback-based API: A client has to register a `process(int n)` function on the server. With this function, the server can instruct the client to read `n` frames from each of its input ports, process them, and write the results to the output ports.

Block-structured engine cycle: JACK uses a fixed sample rate across the entire streaming graph. In addition, the sample processing in JACK is performed in blocks of a fixed size. In one *engine cycle*, such a block of samples passes through the whole graph before a new engine cycle begins. The number of handled samples per cycle is equal to the minimum buffer size needed for every port.

The sample rate and the buffer size in frames are requested by the user when the server is started. A client may request a different buffer size at runtime. In both cases, the JACK audio interface backend (also called *driver* in this context) has to confirm whether it is able to run with this set of parameters.

4.1.2 Engine cycle

The engine cycle of JACK is executed sequentially. Obviously a client cannot be processed before all the clients feeding it with data have finished. Therefore, JACK needs to perform a topological sort to find a total order that is a superset of the partial order given by the streaming graph. The sort is conducted every time the graph state changes—for instance when a port is connected or disconnected or a client gets activated or deactivated.

The following describes the procedure of an engine cycle on the basis of Figure 4.1. For the purpose of explanation, the Linux version of JACK is assumed for the rest of this and the following subsections. Please consider the streaming graph in Figure 2.2: Apparently, the alphabetical order depicted in Figure 4.1 happens to be a sequence that satisfies the partial order of the given graph. A new engine cycle is triggered by the audio interface, when it signals the JACK driver that the output data from the previous cycle has been delivered to the sound card and new input data is available. After reading the new data, the engine then iterates over its sequentialized list of clients. If it comes across an internal client, it simply calls the `process()` function of the client. When the engine encounters an external

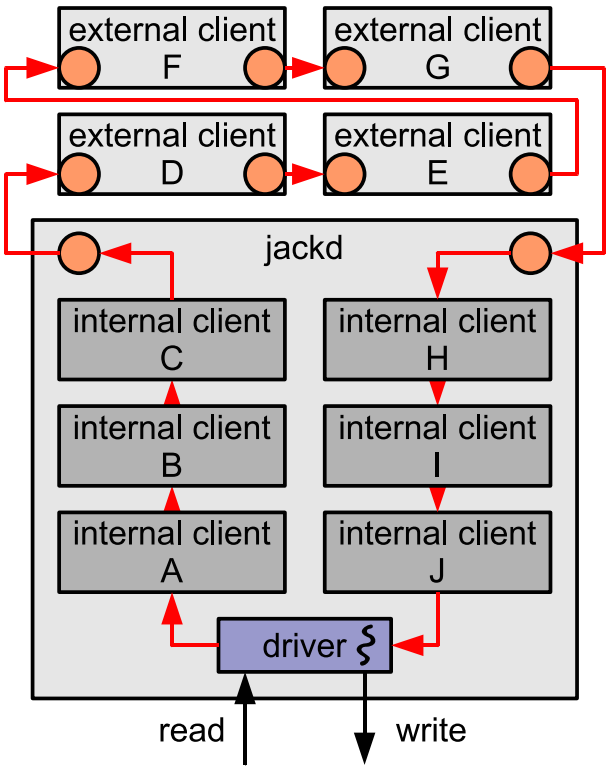

Figure 4.1: The JACK engine cycle. The light gray boxes represent the address space delimiters of the processes depicted: the JACK server daemon and the different external clients. The orange circles stand for the mechanism for external client signaling—on Linux named pipes (FIFOs) are used. For the sake of simplicity the communication channels between the clients and the server have been omitted in the illustration. Please also note that in the terminology of JACK the term *driver* does not refer to a hardware driver of an operating system, but to a JACK backend, which abstracts from the used audio interface.

client, it wakes up the client and blocks—to get woken up in turn by the last member of the external sub-graph it just signaled. The client signaling on Linux is implemented by writing and reading meaningless characters to and from named pipes—also called FIFOs resulting from their behavior (First In–First Out).

Coming back to the the example from Figure 4.1 and applying the procedure explained in the preceding paragraph to it: The only external sub-graph in this example consists of the clients D, E, F, and G. That means, after calling the `process()` function of client C, the engine writes a character to the *wait FIFO* of client D, then reads from the *next FIFO* of client G, and thus gets blocked until G wakes up the engine again. The main developer of JACK calls this method of sequential execution across address spaces *user-space cooperative scheduling* [34].

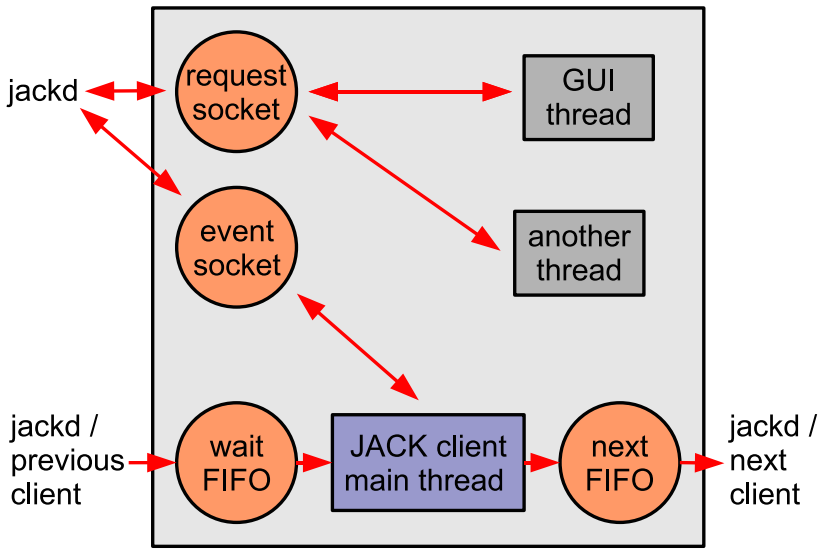

Figure 4.2: An external JACK client.

4.1.3 External client

In this subsection the JACK architecture is described from the viewpoint of an external client. Figure 4.2 illustrates the internals of such a client in more detail than Figure 4.1. At the bottom of the picture, the *wait FIFO* and the *next FIFO* can be seen, which are also depicted in Figure 4.1. As described in the preceding subsection, they are used to synchronize the execution of the server and the external clients.

In addition to the external client signaling mechanism, the JACK architecture provides two channels between each external client and the server: A request channel and an event channel. The former is used by the client library to send requests to the server, for example to register, deregister, connect or disconnect a port, to activate or deactivate the client or to set a new buffer size. When an event has occurred—such as a buffer overrun or underrun, a client (de-)registration or a port (dis-)connection—the server process notifies its clients about it via the event channel. These channels need to be bidirectional for allowing the remote procedure calls to have a return value. On Linux the channels are implemented with UNIX domain sockets. The sockets are symbolized by the orange circles in the upper left corner of Figure 4.2.

Between the *wait FIFO* and the *next FIFO* the JACK client's main thread (also called *audio thread*) can be seen. It runs the client's main loop. Besides the audio thread, there are probably other threads running in the client's address space, for example a graphical user interface handler thread (GUI thread). The main thread is created for the client by the JACK library when the client calls `jack_activate()`. The main loop it runs consists of blocking on the *wait FIFO*, calling the clients

`process()` function and writing a character to the *next FIFO*. But moreover, the audio thread is also responsible for receiving and handling the notifications coming from the JACK server over the event socket. The following lines of pseudo code explain how the event handling is woven into the main loop:

```
while (true)
{
          wait(event socket || wait FIFO);
          // wait for a notification
          // to arrive at the event
          // socket or to get woken up
          // by the previous client
          // in the execution order
          // or the jack server daemon

          handle(events);
          // if no event has occurred
          // and the client has been
          // woken up from the wait FIFO
          // this function simply returns

          read(wait FIFO);
          // this function blocks if the
          // client has been woken up
          // by an event only

          // this function returns immediately
          // if a character had been written
          // to the wait FIFO before and
          // not been read yet

          process(int nframes);

          write(next FIFO);
          // wake up the next client
          // in the execution order
          // or the jack server daemon
}
```

4.2 Jackdmp

Apart from Jackdmp being written in C++, it differs from the original implementation in the following aspects:

Multiprocessor support: If JACK runs on a system with more than one processor, still all the clients are executed in sequence, even if clients could be processed in parallel according to their data flow dependencies.

Jackdmp therefore uses another client signaling model: Instead of performing a topological sort on the streaming graph every time the graph state changes, the client signaling of Jackdmp is directly based on the graph. For each client there is an object of the type `JackActivationCount` located in shared memory. Whenever a client finishes its `process()` function, it calls the `Signal()` function of every activation counter, whose associated client is fed with data by it. As the name suggests, the class `JackActivationCount` contains a counter. At the beginning of every cycle, each client's activation counter value is set to the number of clients it depends upon. The `Signal()` function atomically decrements the counter's value and tests whether the value drops to zero. If this is the case, the function signals the client. On Linux FIFOs are again used for that purpose.

Two JACK threads on client-side: The JACK client main thread explained in Subsection 4.1.3 is split into two parts in Jackdmp: A "real-time thread" running the process loop and a notification thread receiving events coming from the server. Where in JACK the `process()` function of internal clients is called by the server's driver thread, in Jackdmp internal clients too have their own "real-time thread", which gets signaled the same way as for external clients.

Asynchronous driver cycle: In addition to the synchronous driver cycle, in which the driver reads the audio buffer from the sound card, triggers the engine cycle and writes the output data as soon as the graph processing is finished, Jackdmp introduces an asynchronous mode. In this mode, the driver no longer synchronizes with the end of the graph execution. Instead it reads the input data, writes the buffer from the previous cycle, initiates the graph processing and sleeps until it gets woken up by the audio hardware again. The drawback of this approach is that it adds another buffer of latency (cf. Section 6.1). The advantage is a more robust server: If the graph could not be completed in time, in asynchronous mode the driver still has the opportunity to react to this circumstance, while in synchronous mode it is already too late when the driver recognizes the problem.

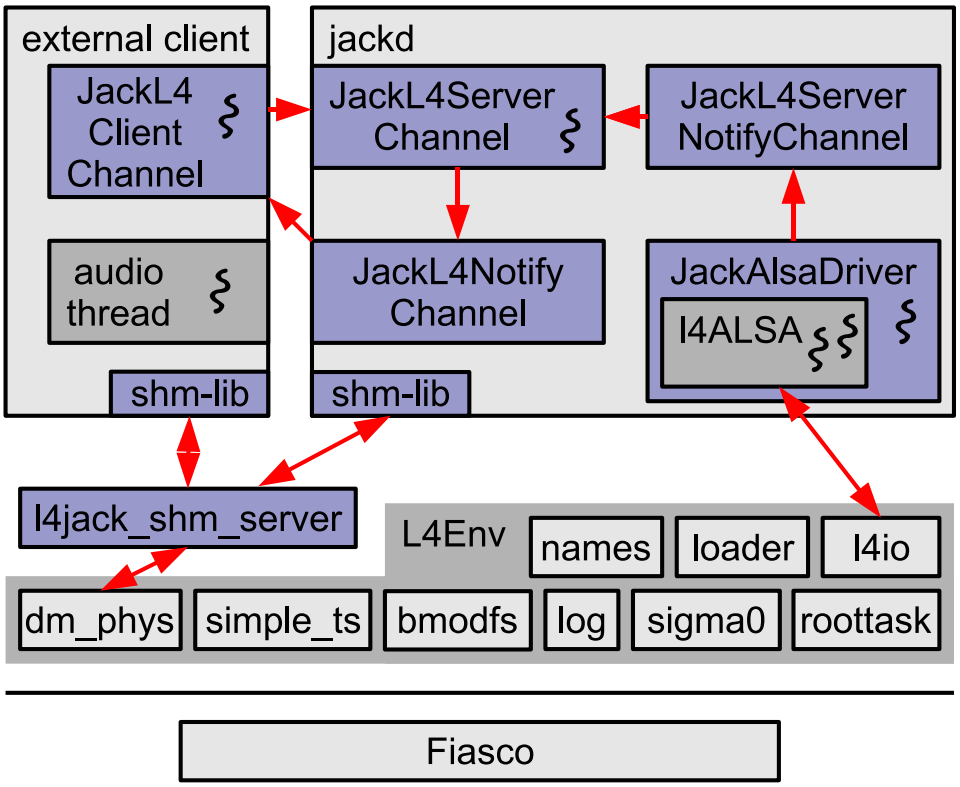

Figure 4.3: Design of the Jackdmp DROPS port. The elements that have been implemented during this project are marked violet in the picture. For simplicity's sake the method for client signaling, possible internal clients, the JackMessageBuffer (run in its own thread), and the JackFreewheelDriver are not visualized in this figure.

The original version of JACK could not easily be adapted to run in asynchronous mode, as it executes the internal clients in the context of the driver.

More extensive design documentation about Jackdmp can be found in [45, 47, 46].

4.3 Jackdmp on L4

The developed design of the Jackdmp DROPS port is visualized in Figure 4.3. The Jackdmp server daemon is depicted in the upper right corner of the picture. It contains the Jackdmp ALSA backend, an instance of JackAlsaDriver that uses the ported ALSA library (see Section 3.2) to access the audio hardware, as well as instances of the three classes for client–server communication—see the following subsection for a detailed description. The driver and the request handler both are executed by their own thread, the ALSA library—or more precisely the Device

Driver Environment (DDE) it uses—starts up additional threads, which are required to emulate the Linux driver environment correctly.

To the left of the server you can see an external example client with its audio thread, which runs the `process()` loop, and its JackL4ClientChannel thread, which runs the event loop—as explained in Section 4.2. Below the external client the shared memory server, l4jack_shm_server, can be found. Next to it, directly above the kernel-userspace border, the subset of L4Env servers that are required to run Jackdmp on DROPS are situated. Finally, at the bottom of the picture you find the Fiasco microkernel (see Section 3.1.1).

The remainder of this section is organized as follows: The following subsection clarifies the functionality of the four classes that implement client–server communication on Jackdmp. Subsection 4.3.2 then introduces the required L4Env servers and reasons in what sense they are needed for the port. Subsection 4.3.3 is devoted to the shared memory server, l4jack_shm_server, and its client library, shm-lib.

4.3.1 Client-server communication

The DROPS port conforms to the class structure of Jackdmp, including the four communication classes. The first three of the classes implement the request channel and the event channel explained in Section 4.1.3. In the following a description of these four classes is given. The alias class names (cf. Section 5.1), as they appear in the platform independent code, are given in parenthesis.

These classes are implemented with the help of the Dresden IDL Compiler DICE [31] on DROPS.

JackL4ServerChannel (JackServerChannel) This class implements the server-side endpoint of the request channel. A handler running in its own thread waits for requests from clients coming over the request channel, and calls the according function in the Jackdmp engine (not depicted), if a request arrives.

JackL4NotifyChannel (JackNotifyChannel) JackNotifyChannel implements the server-side endpoint of the event channel. It provides an abstraction of the system-specific mechanism for sending event notifications to clients to the engine. While the server holds only one instance of JackServerChannel, there is one JackNotifyChannel object per external client.

JackL4ClientChannel (JackClientChannel) The endpoints of the two channels (request channel and event channel) are represented by two separate classes on server-side. By contrast, on client-side both channel endpoints are combined in one class: JackClientChannel contains the thread that receives and handles the event notifications coming from the server; and it provides an abstraction of the system-specific mechanism for sending requests to the server to the external client.

JackL4ServerNotifyChannel (JackServerNotifyChannel) JackServerNotifyChannel enables the driver to send event notifications to clients without being threatened of getting blocked indefinitely: JackL4ServerNotifyChannel requests JackL4ServerChannel in a nonblocking manner to forward the notification. Either the request can be delivered successfully and JackL4ServerChannel notifies the clients via JackL4NotifyChannel later on or the notification gets lost. This mechanism is based on the idea that a lost client notification is less harmful than the driver being blocked indefinitely.

4.3.2 L4Env servers

The following L4Env servers are required to run Jackdmp on DROPS:

names is the L4Env name server. It manages a mapping from the set of names (strings) it has registered to the set of system-wide available thread identifiers (thread IDs). An L4 thread, which wants to offer a service, can register a string value on the name server. If now another thread wants to use this service, it queries the name server, whether the service is available and under which thread ID it can be reached—L4 uses thread IDs to identify communication partners.

The name server is used in the class JackL4ClientChannel by JACK clients to find out the JackL4ServerChannel thread ID they need to know to send an registration request to the JACK server. Then the client registers its JackL4ClientChannel thread ID on the name server to enable the JACK server to find out this ID in order to send notifications to the client. The string, with which the client registers its thread on the name server, is composed of the client name transmitted to the JACK server on registration. Therefore the JACK server can generate this string himself.

l4io is the L4Env input–output resource manager. It administrates I/O port regions and the PCI configuration space. In addition, it also receives all the interrupts the kernel does not handle himself and acts as a userland interrupt multiplexer. It implements the Omega0 protocol [51].

The input–output resource manager l4io is needed by DDE (see Section 2.2.3 in [58]), which the Jackdmp–ALSA backend uses on DROPS to access the audio hardware.

bmodfs is a simple read-only file server.

In Jackdmp drivers and internal clients are compiled as shared object files, which are loaded at runtime and linked dynamically to the server. On startup, the bmodfs file server is equipped with JackAlsaDriver, JackDummyDriver

and potentially internal clients as well, which it hands over to the server daemon on request.

sigma0 In accordance with the microkernel design paradigm by Jochen Liedtke (cf. Section 3.1.1), memory is managed in userland on TUD:OS. To make userspace memory management possible—changing page table entries is allowed in kernel mode only—Fiasco equips the userland with three memory management primitives. It enables L4 tasks to map or grant parts of their address space to other tasks and to flush mapped pages.

The sigma0 task is the initial pager of the system. During the boot process, the kernel transfers the entire available physical memory, including I/O ports and I/O memory, to it. A detailed description of the L4 memory organisation concept can be found in [49].

roottask Comparable to *init* on Linux, roottask is a setup task that needs to be started before the other tasks (roottask must be started directly after sigma0). It acts as a simple system resource manager and substitutes basic parts of several services that have not been started so far—notably the loader, l4io, and simple_ts.

dm_phys is the L4Env physical memory dataspace manager. At start time dm_phys takes the whole physical main memory from sigma0 to offer it to applications in the form of dataspaces. A dataspace is a high-level representation of a piece of memory. The concept of dataspace management [32] is built on top of address spaces and the L4 memory management primitives (map, grant, flush).

The physical memory manager dm_phys is required where main memory should be allocated or freed, for instance in the shared memory manager l4jack_shm_server.

simple_ts is a simple task server and is responsible for creating, configuring and destroying L4 tasks at runtime.

loader The loader is needed if program binaries should be loaded and started at another time than boot time. It depends on the simple task server simple_ts. In the current configuration I have set up the loader to receive the binaries from bmodfs. The loader is necessary to make dynamic linking possible on DROPS.

log The L4Env logserver serializes the output from different running tasks and tags it with the correct source identifier.

4.3.3 Shared memory server

Shared memory plays a prominent role in JACK and Jackdmp. The audio data streaming is based on shared memory, and large parts of the engine data is shared between clients and the server. But the UNIX-like shared memory handling scheme of Jackdmp—a process can create a shared memory segment with a certain name or identifier and make it publicly available, so that another process may attach it at any later instant if it knows the name—does not fit well with the way DROPS handles memory. Although the physical memory handler of DROPS, dm_phys, does allow naming the dataspaces it manages, it does not offer a way to request an existing dataspace by its name. Furthermore, dm_phys attaches a dataspace to a third process only if the owner of the dataspace has explicitly granted access rights to that third process before.

For these two reasons, I introduce an extra intermediate shared memory thread. Another thread can instruct this thread to create a new dataspace on dm_phys for it and associate the dataspace with a given identifier. If a third thread is aware of this identifier, it can now request the shared memory thread to grant the access rights for that dataspace to it. The shared memory thread may do this, because it is the owner of the dataspace. Since the shared memory thread intrinsically has nothing to do with Jackdmp, I decided to place it in its own address space.

A small emulation library, shm-lib, catches the native UNIX shared memory calls and translates them into calls to the shared memory server, l4jack_shm_server. Implementation notes about the shared memory server are given in Subsection 5.2.2.

5 Implementation

This chapter is dedicated to the implementation aspects of the Jackdmp port to TUD:OS. Section 5.1 presents the code cleanup, which has been performed to improve the portability of Jackdmp. Section 5.2 then describes the strategy that was applied to the port. Finally, in Section 5.3 two aspects are highlighted that have caused problems during implementation.

5.1 Portability improvements

The codebase of Jackdmp can be characterized to have a sophisticated object-oriented structure. To substantiate this statement the following example how Jackdmp uses polymorphism to reduce redundancy in the code is given: The method `NotifyClients()` of `JackEngine` walks through the list of registered clients, an array of `JackClientInterface` objects, and calls their virtual method `ClientNotify()`. If the client is internal, the implementation of the class `JackClient` is called, because `JackInternalClient` is derived from `JackClient`—which in turn is derived from `JackClientInterface`—and `JackInternalClient` does not provide an own implementation of that function. If the client is external, the implementation in the class `JackExternalClient`—the representation of an external client on server side—is called, because this class does provide an implementation. The function `ClientNotify()` of `JackExternalClient` initiates a remote procedure call of `ClientNotify()` of the associated `JackLibClient` object—the representation of an external client in the clients address space. Like `JackInternalClient`, the class `JackLibClient` is derived from `JackClient` and does not provide an implementation of the member function `ClientNotify()`.

However, the combination of optional code parts is also done by using `#ifdef` constructions in some places—notably, where a platform specific treatment is required. This causes portability problems. For example: To reuse as much of the existing code as possible, in some cases, we want to pick the solution of Linux, in some that one for Mac OS X, sometimes the Windows version and in some cases a custom solution needs to be developed. Such combinations, of course, cannot be accomplished by defining the `__APPLE__`, `__linux__` or `WIN32` configuration flag. A more elaborate analysis about the damages caused by the excessive use of the `#ifdef` preprocessor statement can be found in [55].

If, on the other hand, I simply copied all the files that need changes for porting to another directory and made the—possibly only small—changes there, I would have ended up with a nearly empty `contrib` [1] directory and a codebase, that is hard to maintain—even during porting. Therefore, the first step of porting was to conduct a code cleanup and get the patches committed to the project's code repository. The refinement process led to several commits[2] to the Jackdmp tree. Its basic elements are overviewed in the remainder of this section.

The guideline of the cleanup was to eliminate as many `#ifdef` statements as possible, to achieve a better separation between platform dependent and platform independent code in general and to make more use of object oriented features for implementing platform differences. As for the separation between platform independent and platform dependent code in header files: For every header named `A.h` that needs platform specific adaptations now a file `A_os.h` can be found in each of the corresponding platform subdirectories. `A_os.h` is included by `A.h` and it contains the needed platform specific parts. A more common way to reach the same result is to use the `#include_next` GNU compiler extension. We cannot use it, because it should be possible to compile Jackdmp with C++ compilers other than GCC. Furthermore, the two new header files `JackSystemDeps.h` and `JackCompilerDeps.h` were created, which include their operating system or compiler dependent counterparts `JackSystemDeps_os.h` and `JackCompilerDeps_os.h`. Often used macros, compiler instructions, type definitions and system header inclusions have been collected in these two files. Although it is not always possible to clearly distinct between operating system and compiler dependence, it makes sense to have these two separate files anyway: It enables using different compilers on the same operating system on the one hand, and sharing the same compiler dependent macros and definitions across platforms on the other.

Since as little code as possible should be moved from source to header files, the same principle could only partly be applied to the source files. As to the C++ files, the focus thus was put on using object orientation for improving portability. For this purpose, a new header `JackPlatformPlug.h` was introduced, which includes the header `JackPlatformPlug_os.h`. This file is used to plug in the correct class for the specific operating system with the `typedef` directive and again is located in each of the corresponding platform subdirectories. Caused by this `typedef` mechanism, a platform dependent class has two names: The name it is declared with and the interface name, with which it appears in the platform independent code. Each the other name is called *alias name* from now

[1] Like in many projects, it is a coding convention at this chair to put files, which are completely unchanged, in a distinct `contrib` directory, when reusing code. This makes it easy to see, which parts of the code could be taken over unchanged and which parts had to be adapted.

[2] See 2008-08-28, 2008-08-31, 2008-09-01, 2008-09-04, 2008-09-05, 2008-09-19, 2008-09-20 in `http://subversion.jackaudio.org/jack/jack2/trunk/jackmp/ChangeLog`.

on. Even though treating the C files was a harder task, a few of them—such as `JackTime.c`—could be split and moved to the matching directories.

Finally an example shall be given to demonstrate how this cleanup improved the constitution of the Jackdmp source code: While before the cleanup, the code of the function `jack_drop_real_time_scheduling(pthread_t thread)` in `JackAPI.cpp` was

```
#ifdef __APPLE__
    return JackMachThread::DropRealTimeImp(thread);
#elif WIN32
    return JackWinThread::DropRealTimeImp(thread);
#else
    return JackPosixThread::DropRealTimeImp(thread);
#endif
```

after the cleanup it simply reads

```
    return JackThread::DropRealTimeImp(thread);
```

5.2 Porting strategy

The code cleanup performed during this project helped to make the porting process easier and more structured, as exemplified in the preceding section. In any case, an all-embracing cleanup—introducing abstractions everywhere changes are needed for porting—is out of this thesis' scope. Consequently, although the desired solution is a modular implementation, a hybrid porting strategy had to be employed. It consists of the following three approaches:

5.2.1 Modular implementation

This section lists the classes that could be implemented in a modular manner. Their alias names (cf. Section 5.1) are given in parenthesis.

The client-server communication classes: The two communication channels between the server and a client are designed with four different classes. Their functionality is explained in Section 4.3. On DROPS they are implemented with the help of the Dresden IDL Compiler (DICE, [31]).

- JackL4ServerChannel (JackServerChannel)
- JackL4NotifyChannel (JackNotifyChannel)
- JackL4ClientChannel (JackClientChannel)
- JackL4ServerNotifyChannel (JackServerNotifyChannel)

JackL4Thread (JackThread): The thread abstraction class is implemented with the L4 thread library (l4thread) of the L4 environment (see Section 3.1.2).

JackL4Mutex (JackMutex): A lock abstraction. It uses the L4 lock library, also a part of L4Env.

JackL4ProcessSync (JackProcessSync): This class provides a synchronization primitive for threads within an address space. On Linux, it is implemented using pthread (POSIX Threads) condition variables. The L4 version uses the condition variables that are supplied by DDEKit, a part of DROPS' Device Driver Environment (see Section 2.2.3 in [58] for more information).

JackL4IPCSynchro (JackSynchro): JackSynchro encapsulates the client signaling method used on the particular platform. The client signaling on DROPS is achieved by sending and waiting for so-called register messages. Section 5.3.2 gives more details.

Furthermore, the Jackdmp time abstraction—located in the C file `JackL4Time.c`—could also be implemented modularly. Its function `GetMicroSeconds()`, which is used in Jackdmp to measure time intervals, is realized with the time stamp counter [61] of the CPU. Because the timer resolution of Fiasco is currently about 10 ms, the function `JackSleep(long usec)` performs busy waiting if the number of microseconds to be waited is less than 9000, and only otherwise it calls `l4_usleep(int usec)`.

5.2.2 Reimplementation of an interface

Another method commonly used for porting is reimplementing a well-defined interface. This method was, for instance, adopted for the ALSA port (cf. Section 3.2). It has the advantage that it enables a clean separation of reused and self-provided code. On the other hand, system level interfaces can be complex, and their semantics might be based on assumptions that are not always evident.

The shared memory handling of Jackdmp is taken without much change from the JACK codebase. It is located in the C source files `shm.h` and `shm.c`. A small class interface encapsulates the code to fit in the object-oriented structure of Jackdmp. Implementing either the interface defined by `shm.h` or the object-oriented one of Jackdmp would have been possible approaches for bringing JACK's shared memory handling to DROPS. But as a lot of the functionality of `shm.c` is needed for the DROPS implementation anyway, I preferred to leave `shm.h` and `shm.c` as they are and to emulate one of the interfaces they use. The System V was chosen in favor of the POSIX version, because it is less complex and therefore easier to reimplement.

5.2.3 Brute force

If neither a modular implementation nor an emulation are feasible, a method that is always possible is directly changing the code where needed. A slightly more elegant way to achieve the same result is copying the files that need changes to another directory, making the changes on the copy, and setting up the include directory order or the build system in way that these changed copies overlay the original files. The following files needed changes that could only be carried out with the brute force method explained here:

- `JackServerGlobals.cpp`
- `JackControlAPI.cpp`
- `JackDriverLoader.cpp`
- `JackTools.cpp`

5.3 Implementation peculiarities

5.3.1 Exception handling and run-time type information

Jackdmp makes use of C++ exceptions, notably to handle errors that may occur during shared memory allocation. Unfortunately, L4Env (cf. Section 3.1.2), does not provide the infrastructure needed for C++ exception support: If exceptions are utilized in a C++ project, the GNU compiler—which is used for the whole DROPS system—wraps certain wind and unwind code around every function. This code is situated in the compiler's support libraries `libsupc++.a` and `libgcc_eh.a`. In the GNU compiler collection version four series these support libraries are implemented with thread-local storage via the *gs* register. This mechanism is not available on DROPS, and consequently linking an application with the libraries results in broken binaries.

Norman Feske, co-author of the Genode framework [8], kindly advised me on how it is nevertheless possible to equip an L4Env application with C++ exception support: by linking it against `libsupc++.a` and `libgcc_eh.a` from a GCC version three, while still compiling it with a version four compiler. The support libraries from a version three GCC do not depend on thread-local storage. Before exceptions can be thrown and caught, the exception handling must be initialized by calling the function `__register_frame()` of `libgcc_eh.a`. As a positive side effect the two support libraries do not only bring exception handling to DROPS, but also the C++ run-time type information (RTTI) system.

5.3.2 Client signaling

While the Linux version of the client signaling mechanism uses named pipes (FIFOs), on DROPS it is implemented with a L4 feature called *short IPC* or *register message*. The register message is the fastest available L4 IPC mechanism, and—like all kernel-supported IPC types on L4—it works in synchronous fashion. The short IPC send system call (`l4_ipc_send(..., L4_IPC_SHORT_MSG, ...)`) transmits the content of the registers *EBX* and *EDX*, provided the destination thread has invoked the short IPC receive system call before (`l4_ipc_wait(..., L4_IPC_SHORT_MSG, ...)`) and consequently is already waiting. Otherwise, the sender thread gets blocked and waits for the receiver to call `l4_ipc_wait()`. Timeouts ranging from zero to infinity can be set for receive and send IPC system calls.

When performing a classic short IPC, the Fiasco microkernel switches from the source thread to the destination thread, but leaves the registers *EBX* and *EDX* untouched for the destination thread to read the message stored there by the source thread. The time slice[3] of the source thread gets donated to the destination thread, which means that the destination thread runs when the short IPC is finished. This behavior is not desired for our case, as it causes unnecessary context switch overhead: The signaling thread, which was going to call `l4_ipc_wait()` and sleep directly after the short IPC anyway, is preemted from the CPU and scheduled later on again only to call `l4_ipc_wait()` and get preemted again. But the time slice donation when sending short IPCs can be suppressed on Fiasco by setting the *deceit bit*. Then the sender thread keeps the CPU after sending a short IPC, provided it has a priority that is higher than or equal to the priority of the receiver. More information about the *deceit bit* can be found in Section 5.2.1 of [35].

The class encapsulating the client signaling implementation is called JackSynchro in the platform independent code parts, the DROPS alias is named JackL4IPCSynchro. JackSynchro contains—amongst others—the following functions:

- `Signal()`
- `Wait()`
- `TimedWait(long usec)`
- `Allocate(const char * name, const char * server_name, int value)`
- `Connect(const char * name, const char * server_name)`

[3] Fiasco distinguishes between *execution contexts* and *scheduling contexts* that can be handled independently. See [53] for more information.

`Allocate()` is called by the Jackdmp server during client registration, `Connect()` must be called by the client that is represented by the JackSynchro object and by every client that possibly wants to signal the corresponding client. The desired behavior of JackSynchro can be characterized as a cross-process condition variable. Please note that the instances of that class are not located in shared memory. External clients and the server each have their own local array of that type.

One problem that occurred during the implementation of JackL4IPCSynchro is that L4 does not provide separate objects to identify IPC end points, such as Mach ports—it simply uses thread IDs. But when allocating the JackSynchro object of a client, the audio thread has not been created yet and consequently its thread ID is unknown so far. Hence, the L4Env name server (cf. Section 4.3.2) could not be used to map the client name to the ID of its associated audio thread, because the name server does provide delayed registration.

A possible solution would have been to only save the client name during `Allocate()` and `Connect()` and let the audio thread register itself the first time it calls `Wait()` or `TimedWait()`. But being forced to call the name server in the real-time path is not a good solution.

I circumvent the problem in the following way: I introduce a new shared memory array that assigns to every reference number (a unique number assigned to every client by Jackdmp) the ID of the associated audio thread—if no thread ID is known, the value is equal to `L4_INVALID_ID`. `Allocate()` and `Connect()` instruct the shared memory server (cf. Section 4.3.3) to attach this memory region and to create it if it does not exist yet. The first time the audio thread calls `Wait()` or `TimedWait()` it can store its thread ID in the shared memory array.

But this solution raises another problem: `Allocate()` and `Connect()` have the client name as a parameter, but not the client's reference number. Fortunately, there is a global function called `GetSynchroTable()` both on client-side and on server-side, which returns the array of JackSynchro objects. The array is always indexed by the reference number that the server assigns to every client on registration. Now, JackL4Synchro can perform a reverse lookup of itself (*this* pointer) in the table to obtain its own reference number.

6 Client characterization methodology

The most important aspect to be aware of about execution time and latency in JACK is that by the architecture of JACK the latency of the graph is independent of its execution time—except for the unavoidable influence of the client's execution times on the minimum possible graph latency: If the execution time of a client scatters too much, it may happen that the execution of the graph is not finished at the end of all cycles. Furthermore, a too large minimum execution time of a client or a combination of clients may prevent the graph from being executable in time at any latency.

Hence, the client characterization methodology proposed in this chapter is limited to real-time aspects. Section 6.1 describes how the overall latency of a JACK graph can be computed and why it is not influenced by the clients' execution times. In the subsequent section I propose a scheme to quantitatively characterize the timing behavior of JACK clients and how these introduced characteristics can be used in a real-time system to make timing guarantees. In the remit of this thesis the theory on jitter-constrained streams is suggested as a potential source of help for the development of this client parametrization methodology. Section 6.3 argues why this suggestion was rejected.

6.1 Latency

To discuss how the latency of a JACK graph results, I first explain the basic functionality of computer's audio interface in general and how it inherently constrains the possible minimum throughput latency. Afterwards, the computation of the JACK graph latency is derived from that.

The typical audio interface of a computer has an input buffer, where it stores the data coming from the analog–to–digital converter, and an output buffer, from which it reads the data that it feeds into its digital–to–analog converter. At a constant rate the sound hardware sends an interrupt to the CPU, signaling that new data is available to be read on the input buffer and that there is free space on the output buffer for data to be written. Only a rare number of sound cards does not synchronize input and output, and sends interrupts for playback and recording separately. Both the output buffer and the input buffer must be partitioned into at

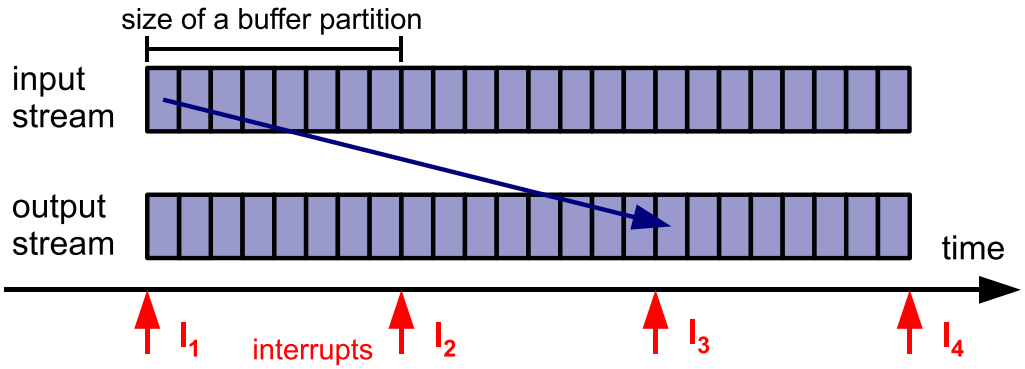

Figure 6.1: A chart showing the frame of the input stream and the frame of the output stream currently being processed by the audio hardware at a moment in time.

least two parts: One part that is accessed by the sound card and one part that can be accessed by the CPU. When the interrupt occurs these roles of the buffer partitions are switched. This technique is called *double buffering*, and in contrast to graphics cards, sound cards cannot operate in single buffering mode, in which the CPU and the I/O hardware access the same single buffer part concurrently. ALSA supports it to divide the buffers into more than two parts, provided the sound card can operate in this mode. The I/O hardware then accesses these parts in a round-robbin manner. The interrupt frequency is equal to the sample rate divided by the size of a buffer partition in frames.

With the knowledge from the preceding paragraph, we can now calculate the minimum possible throughput latency, that is the delay the system adds by piping the input into the output as fast as possible. Figure 6.1 visualizes, for a certain moment in time, the frame of the input stream that is currently written to the input buffer by the audio hardware, and under it the frame of the output stream that is currently played back. Double-buffering is assumed in the figure. As it can be concluded from the chart, the occurrence of interrupt I_2 is the earliest instant the CPU can access the block of frames the sound card has been writing since interrupt I_1. Consequently, interrupt I_3 is the earliest instant the sound card can start to playback this data. It is therefore easy to see that the minimum throughput latency $\mathrm{Lat}_{\mathrm{min}}$ is calculated by

$$\mathrm{Lat}_{\mathrm{min}} = \frac{n \cdot p}{f}, \tag{6.1}$$

where p is the size of one buffer partition in frames, f the sample rate, and n the number of partitions per buffer.

Now coming back to JACK: The JACK sound architecture is based on the principle of fixed block size audio processing, which means that the buffer size and sample rate of the entire JACK graph are also the same parameters used by the driver and

finally also by the sound card—where the JACK buffer size corresponds to the size of one buffer partition on the sound card. Hence, the JACK sound architecture offers its client the minimum latency possible for these two parameters—or in other words, JACK does not add any latency. The same applies to Jackdmp, except for one difference: When Jackdmp operates in asynchronous mode (see Section 4.2), it does add a buffer of latency and its graph latency then computes as

$$\mathrm{Lat}_{\min} = \frac{(n+1) \cdot p}{f}. \tag{6.2}$$

While JACK provides the minimum possible latency to its clients, the clients themselves may add additional latency to the signal path. The simplest example is a client that buffers the input for a certain amount of time and outputs it with delay. An appropriate scheme to determine the latency that is added to the signal path is currently being discussed in the JACK community. The present proposal by Paul Davis[1] plans to let clients compute the overall latency of their output ports autonomously. An attribute is attached to every port indicating the latency that has been added to the signal path so far. A client that adds further latency has to retrieve these values from its input ports, sum it up with the latency caused by itself, and store the results in the attributes of its output ports. The most important fact for us about this possible additional latency is that it again does not depend on the time it takes to compute the client's `process()` function, but only on the specific signal processing algorithm of the function.

6.2 Methodology proposal

The scheme I propose in this section is visualized in Figure 6.2. As elaborated in the preceding section, the latency of a JACK graph does not directly depend on the clients' execution times. Moreover, the JACK design does not provide mechanisms for real-time admission or related tasks. Therefore, the scheme is designed orthogonally to the JACK server. First the structure of the scheme is described in principle, afterwards possible types of the exchange parameters are discussed in more detail.

Although it may be already obvious, I should clarify at this point that DROPS is an event-driven real-time system. For the scheme, I introduce a JACK admission server, on which the JACK clients must register themselves additionally. The JACK admission server may be a JACK client himself, for example to obtain information about the graph structure or to request the driver to adjust its settings as suggested in Section 6.3. After registration, a client i has to transmit its current execution time characteristic $c(i)$ to the JACK admission server. Depending on the target

[1] `http://article.gmane.org/gmane.comp.audio.jackit/18654`

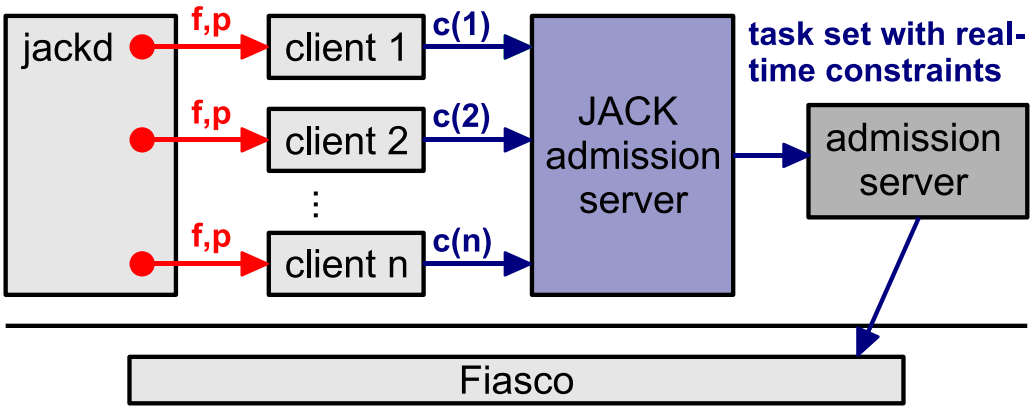

Figure 6.2: Design of the proposed scheme. As the execution times of the clients do not influence the system latency directly, the scheme is designed orthogonally. The red arrows refer to the JACK client–server protocol, the blue arrows stand for the protocol of the new scheme.

admission server and scheduling algorithm, these characteristics can be, for instance, the probability density function of the `process()` function's execution time or its worst case value. The characteristics are transmitted at runtime, since they may depend on the internal state of the client (internal parameters), the frame rate or the used buffer size—as verified by the measurements presented in Section 7.3. The possible dependencies result from the particular algorithm of the client's `process()` function and therefore only the client itself can know about them. Hence, it is the task of the client to take care about these dependencies and provide a correct value for a certain state.

After receiving all the execution characteristics from the clients, $c(1)$, $c(2)$, ..., $c(n)$, the JACK admission server maps these characteristics to the real-time task model requested by the admission server of the operating system (from now on called *OS admission server*). It would also be possible to join the JACK admission server and the OS admission server to one single server. The JACK admission server is informed by the OS admission server about the outcome of the admission and notifies its clients about it in turn. If the admission was successful, the OS admission server maps the task model it uses to the scheduling primitives of the Fiasco microkernel and requests it to reserve the required CPU demand. If the characteristic of a client changes, it has to notify the JACK admission server about it and the whole admission process has to be repeated. The scheduling primitives offered by Fiasco are based on fixed priorities and enforceable time slices [56]. A large set of scheduling and admission algorithms—for example Rate Monotonic Static scheduling (RMS)—can be mapped to these primitives.

In the following I will preset two possible types of client characteristics. Although a deadline miss in an audio production system does not cause miserable harm—

like a deadline miss in an aircraft control system, the real-time requirements of a professional audio production system are principally hard: No buffer overruns or underruns should occur. But the construction of true hard real-time systems causes huge engineering effort. For instance, the determination of the worst case execution time cannot be based on naive measurements, because it requires a method that considers all possible cases. Furthermore, basing the CPU demand reservations on worst case execution times leads to a bad CPU utilization, since the average-case execution time of a job is normally significantly lower than its worst case value (the measurements presented in Section 7.3 show that). Finally, because of features like the system management mode [60], true hard real-time guarantees cannot be given on a desktop computer anyway, and extremely few buffer overruns or underruns may be tolerable even on a professional audio workstation. Consequently, at least two sets of parameter types are reasonable:

Hard real-time approach In this approach the client characteristics $c(1)$, $c(2)$, ..., $c(n)$ are the worst case execution times of their `process()` function. If the used scheduling algorithm is capable of treating task dependencies properly, the JACK admission server can directly map the graph structure to the description of task dependencies requested by the OS admission server. Otherwise, the signal time of the clients may be modeled as their release time and the absence of task dependencies can be assumed, as the clients are blocked by the operating system until they are signaled.

Soft real-time approach In this approach, the client characteristic $c(i)$ is the probability density function (possibly approximated with normalized histograms) of the execution time of client i. The JACK admission server models the whole graph as one single task, whose execution time has a probability function obtained by convolving all client characteristics. With this probability density function, the JACK admission then can determine the CPU demand required for a certain percentage of graph cycles to be finished in time.

For this approach, it must be assumed for all clients with at least one connected input port that the execution time of their `process()` function does not depend on their input signal. Otherwise, the execution times of the clients cannot be modeled as independent random variables. Only if two random variables X and Y are independent from each other, the probability density function of the random variable X + Y can be obtained by convolving the probability density functions of X and Y.

Since the JACK client model does not provide a notion of job parts that can be executed optionally or a quality level that can be reduced at runtime resulting in a lower execution time[2], unfortunately the advanced soft real-time scheduling algorithms available for DROPS [37, 38] cannot be used here.

[2] In my judgement it is not possible to find such a model that applies in general.

A promising intermediate solution between the soft-real time and the hard-real time approach is treating the JACK clients and the JACK driver as tasks with hard real-time constraints, but obtaining their "worst case" execution times with a well-considered method of extensive measurement.

6.3 Jitter-constrained streams

The theory of jitter-constrained streams [39, 36] presents a generalized reference model to describe data streams of packets with an equidistant arrival time that may vary within bounded limits. It makes statements about how buffers have to be dimensioned not to loose packets, provided its model is applicable. Because the JACK architecture is based on the principle of block-structured streaming with a fixed frame rate and buffer size (see Section 4.1.1 and Section 4.1.2 for more information), the theory of jitter-constrained streams is helpful for this project only in one single sense:

If the minimum and maximum execution times of all clients are known, the minimum and maximum graph execution time can be derived from these values easily. In this case the finish time of the graph can be modeled as the arrival time of a jitter-constrained stream's data packet. Then the theory can be used—for example by the JACK admission server introduced in the preceding section—to obtain the minimum number of buffer partitions required if no overrun or underrun should happen in the sound card buffer. The theory can be applied without adaptions for this purpose and therefore no further discussion is needed here.

7 Evaluation

This chapter compares the results achieved in this project with its objective defined in Section 2.2. The objective is divided into two main parts: The first part of the objective was analyzing existing open source sound architectures and porting the most appropriate one to DROPS. It is evaluated in Section 7.2. The second part was the development of a client characterization methodology and is evaluated in Section 7.3. The general test setup used for the timing measurements presented in this chapter is given in Section 7.1.

7.1 Measurement setup

All the measurements have been made on the same test machine, which is equipped with an AMD Duron CPU (clock frequency: 1303.058 MHz) and 512 MB of main memory. The time was measured with the time stamp counter [61] of the CPU. The values from the time stamp counter yield correct results in this case, as the test machine has only one CPU, which does not support dynamic frequency scaling. The used GNU/Linux distribution was *Ubuntu Studio, Hardy Heron* with a low latency kernel [16] of the version 2.6.24-23. Fiasco and L4Env were taken from the internal code repository (Fiasco: revision 33602, L4Env: revision 30084). Jackdmp and its clients were executed with real-time priorities on Linux; on DROPS all tasks had the same priority. In all cases Jackdmp was operating in asynchronous mode. If not marked otherwise, 5000 values were taken per measurement.

The following clients have been subject to measurement:

jack_metro A signal generator repeating sections of a sine signals interrupted by intervals of silence at a constant rate. It ships with Jackdmp as an example client.

jack_thru Also an example client from Jackdmp. It has two input ports and two output ports. The client simply copies the content of its input buffers to the output buffers.

GVerb (gverb_1216) A commonly used reverb emulator.

japa The JACK and ALSA Perceptual Analyser (japa, [12]) is an audio spectrum analyzer that also includes a white noise and a pink noise signal generator.

Dyson compressor (dyson_compress_1403) A dynamic range compressor.

DJ Flanger (dj_flanger_1438) A flanging audio effect filter. Flanging belongs to the class of phase-shifting effects.

Multiband EQ (mbeq_1197) A fixed band equalizer.

DJ Flanger, GVerb, the Dyson compressor, and the multiband equalizer are LADSPA plugins from the *SWH Plugins* package [20]. They were turned into JACK clients with JACK Rack [10].

7.2 Porting an open source solution to DROPS

7.2.1 Architecture selection

The JACK sound architecture makes it possible to wire various external and internal clients together to form a virtual sound studio, as it is required in Section 2.2.1.

As regards the long-term goals defined in Section 2.1, JACK faciliates synchronization at sample-level precision, which is the most accurate inter-stream synchronization that can be achieved on a digital sound system. Furthermore, with JACK's block-structured engine cycle any software caused latency jitter can be avoided. As elaborated in Section 6.1, JACK enables its clients to run at the lowest throuput latency that is possible for the chosen set of driver parameters.

7.2.2 Jackdmp port

The version of Jackdmp this port is based upon contains 71,955 lines of code (numbers generated using David A. Wheeler's SLOCCount [18]). The part of the DROPS port that was implemented modularly (see Section 5.2.1) or by reimplementation (see Section 5.2.2) counts 1,958 lines of code, the part adapted with the brute force method (see Section 5.2.3) counts 786 lines of code. The latter part mainly consisted of commenting out UNIX specific function calls that are not implemented on DROPS, but which are not necessarily required to run Jackdmp. Thus, a code structure suited well for porting can be attested to Jackdmp, especially after the cleanup performed during this project.

The functionality of the streaming architecture was validated by extending the dummy driver with a monitor function that sums up all the samples it receives as input during one cycle, and then prints the summed values on the console. I compared this console output from the DROPS version with the Linux version for different graph configurations and signal generators and I noticed matching results in every case. Hence, it can be concluded that the streaming works correctly in the port. Furtheremore, I performed unit tests with serveral single components,

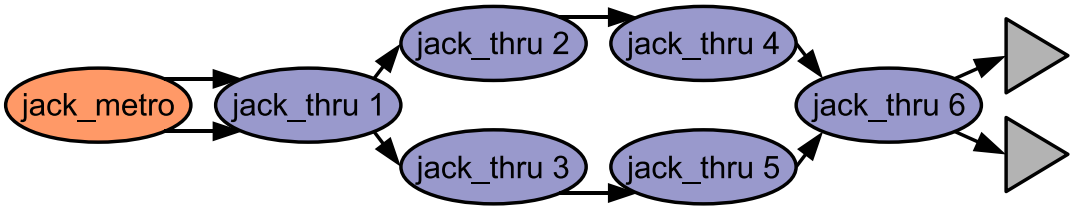

Figure 7.1: Test setup used to analyze the real-time performance of Jackdmp on DROPS. The setup contains six instances of jack_thru and one instance of jack_metro. The stereo output of jack_metro is fed into jack_thru 1, then it is split and led to the two parallel jack_thru client chains. Instance six of jack_thru joins the signal again and outputs it to the dummy driver. Because jack_metro does not have an input, it is signaled by the Jackdmp engine.

such as the client signaling mechanism, the client–server communication classes, the semaphore abstraction, the condition variable implementation and the time abstraction.

Unfortunately, I was not able to get the Jackdmp–ALSA backend working on DROPS for the following reason: The ALSA backend shipped with Jackdmp works in `mmap` mode and consequently uses the system calls `mmap()` and `poll()`. These calls depend on an advanced infrastructure in the kernel and cannot be implemented by simply redirecting them to the corresponding function in the file operations structure of the file they are invoked on. DDE does not provide this infrastructure at the moment, and a Jackdmp–ALSA backend that can operate in `read()/write()` fashion is not available. After putting much effort in extending DDE with the required infrastructure without results, I decided to focus on other tasks—such as the measurements—rather than finishing the driver.

7.2.3 Real-time performance

To analyze the real-time perfomance of the port, the timing of the JACK graph depicted in Figure 7.1 was traced while beeing executed on DROPS. To have a set of reference values, with which the results can be compared, the same experiment was performed on Linux again. The following four parameters have been measured for each of the clients in the setup:

Signaling latency The time interval between the instant the client is signaled and the instant it wakes up.

Sleep latency The time interval between the instant the client has finished its `process()` function (which is equal to the instant before the client starts signaling its successors) and the instant before it starts sleeping. This value is helpful to unveil the difference in timing caused by setting the *deceit bit* (see Section 5.3.2).

Sleep time The time a client sleeps until it gets signaled again.

Duration The time between the instant a client starts processing and the instant it has finished its `process()` function.

The variance and the mean value of the numbers measured on DROPS are listed in Table 7.1 for the case the *deceit bit* (cf. Section 5.3.2) is set when signaling clients, and in Table 7.2 for the case it is not set. The results of the test on Linux are listed in Table 7.3.

The measurements show that the signaling latency on DROPS is significantly lower on the average for all the clients—with the exception of jack_thru 4 running with the version where the deceit bit is set. Furthermore, the tremendously lower variance in the signaling latency and the duration in all the cases indicates that DROPS provides a much better predictability in scheduling than the low latency patched Linux kernel. A lower average value of the duration can be interpreted as less overhead caused by the operating system, for example with inefficient scheduling decisions. Thus, the assumption made in Section 5.3.2 that setting the deceit bit for client signaling reduces scheduling overhead has to be questioned. The high variance of the sleep time on DROPS results from a transient phenomenon that can be seen in the raw data, and which lasts for about the first 50 cycles. Unfortunately, I was not able to find out what causes this phenomenon.

7.3 Development of a client characterization methodology

The methodology proposed in Section 6.2 is based on the assumption that the execution time of a client's `process()` function can depend on the current internal client state (internal parameters), the frame rate or the used buffer size. Moreover, the soft real-time approach suggested in the same section assumes that the execution time of the handled clients is not influenced by their input signal. To analyze whether these assumptions are reasonable and to gain further knowledge about their timing behavior in general, the execution time of typical JACK clients was measured during this project. Except for the frame rate, the influence of these parameters (buffer size, input signal and internal state) on the execution time has been tested for several clients. The results of these measurements are presented in following subsections.

These measurements have been done on Linux, because the connection of the Jackdmp port with L^4Linux has not been accomplished yet, and therefore for most of the available clients it would be a hard task to port them to DROPS.

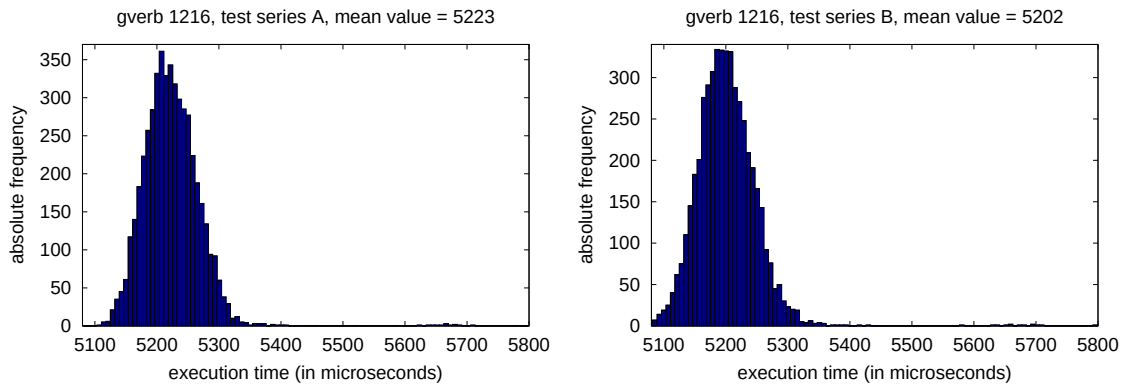

Figure 7.2: The histograms of two different measure series with the same setup.

As a quantitative parameter of the deviation between two measure series A and B, I define the *difference* D_{AB} of their harmonized histograms as

$$D_{AB} := \sum_{i=1}^{N} |a_i - b_i| \,, \tag{7.1}$$

where N is the number of bins in the histograms, and a_i or b_i is the value in bin i of measure series A or B. *Harmonized* means in this context that the histograms of A and B are adjusted to consist of the the same number of bins N, with each pair of bins with the same index representing the same interval of the measured value. A value of D_{AB} close to zero indicates good compliance between A and B, a value close to 2K indicates a bad compliance, where K is the number of measure points taken. For example, the value D_{AB} of the histograms depicted in Figure 7.2 is equal to 2116, where $K = 5000$ and $N = 100$. This parameter provides a rough estimation of the deviation between two histograms, and is used here to check whether a certain circumstance changes the measured distribution of a random variable or not. But it should be treated with caution, since it converges to 0 for $N \to 1$ and to 2K for $N \to \infty$ for any two measure series. For a more trustworthy statement about the independence of two statistically obtained random variables from a parameter or certain circumstances, much more data must be acquired and a well-accepted statistical method has to be used.

7.3.1 Buffer size

The execution time of gverb was measured at a sample rate of 48 kHz for the buffer sizes 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096. During these measurements, the white noise signal generator of japa was connected to the input of gverb, and the output of gverb was connected to the driver. The resulting histograms are given in Figure 7.3 and Figure 7.4. A different vizualization of the results is

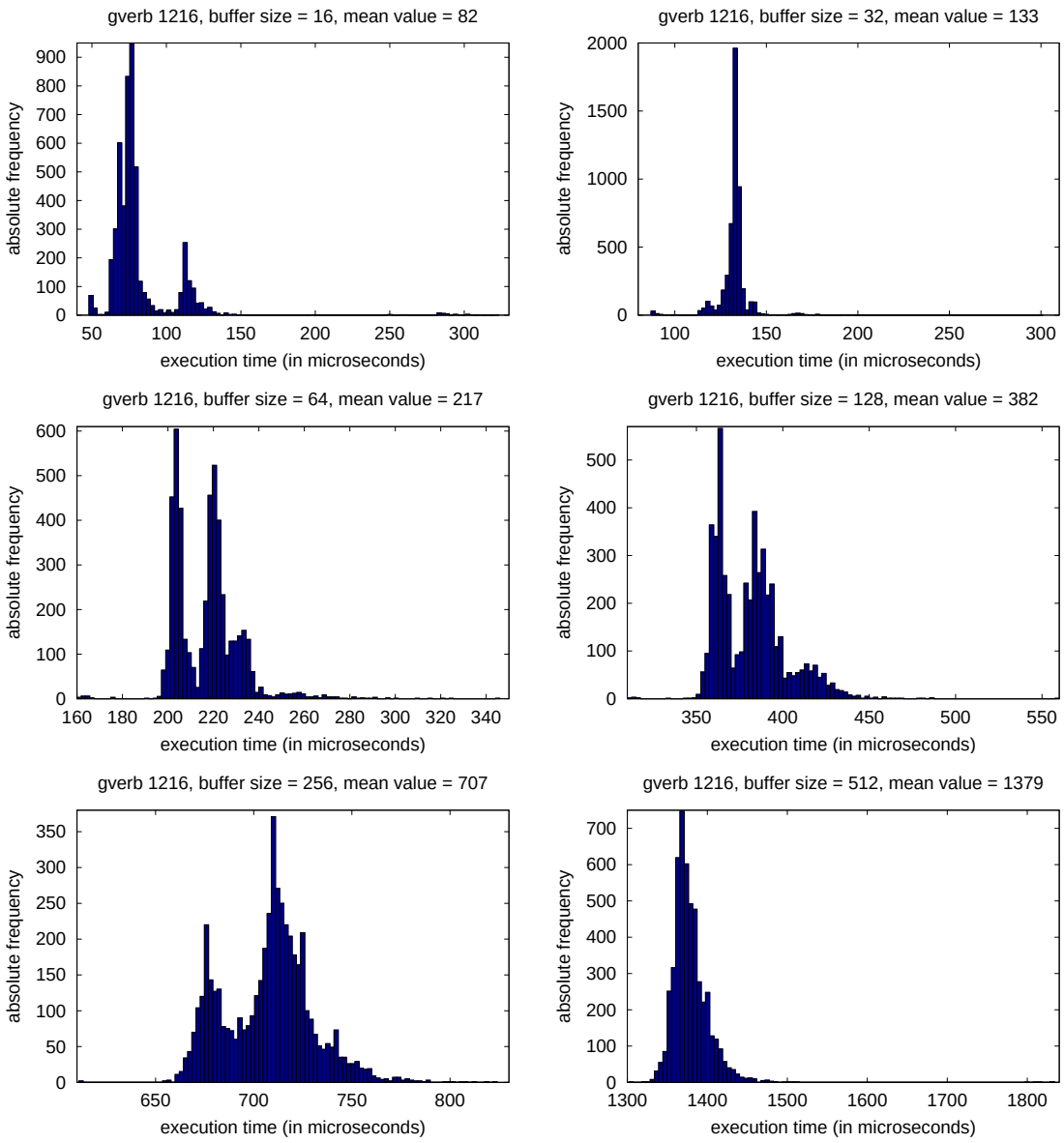

Figure 7.3: Histograms of the measured execution time of gverb for different buffer sizes, part one (buffer size 16–512).

presented in Figure 7.5: A *quantile plot*, which shows for all the tested buffer sizes the measured mean values, and in addition the range between the 0.1-quantile and the 0.9-quantile as a green error bar, as well as the range between the maximum and the minimum of the measure series as a red error bar.

The same experiment was performed with japa and with jack_metro. The quantile plots of their results are given in Figure 7.6 and Figure 7.7.

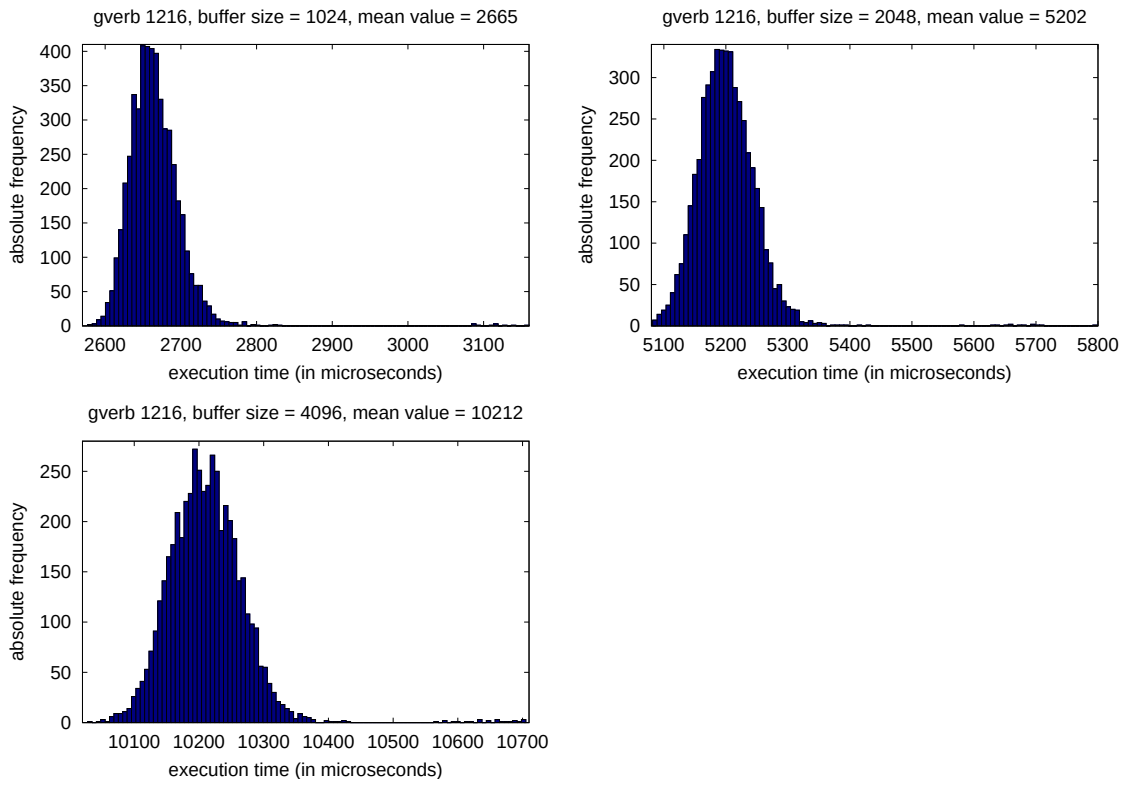

Figure 7.4: Histograms of the measured execution time of gverb for different buffer sizes, part two (buffer size 1024–4096).

7.3.2 Input signal

The following signals were used to test whether the execution time of gverb depends on its input:

White noise A stochastic signal with a constant spectral density—generated by japa in this case.

Pink noise A stochastic signal with a spectral density that is inversely proportional to the frequency: $S(f) \propto \frac{1}{f}$—also generated by japa here.

Linear chirp A signal with the shape $x(t) = \sin\left(2\pi(f_0 + \frac{k}{2}t) \cdot t\right)$, periodically repeating.

ZynAddSubFX The output signal generated by playing the *ZynAddSubFX* software synthesizer [27].

Silence $x(t) = 0$.

The pairwise differences D_{AB} of the resulting histograms are given in Table 7.4, the histograms themselves can be found in Figure 7.8. It can be concluded that

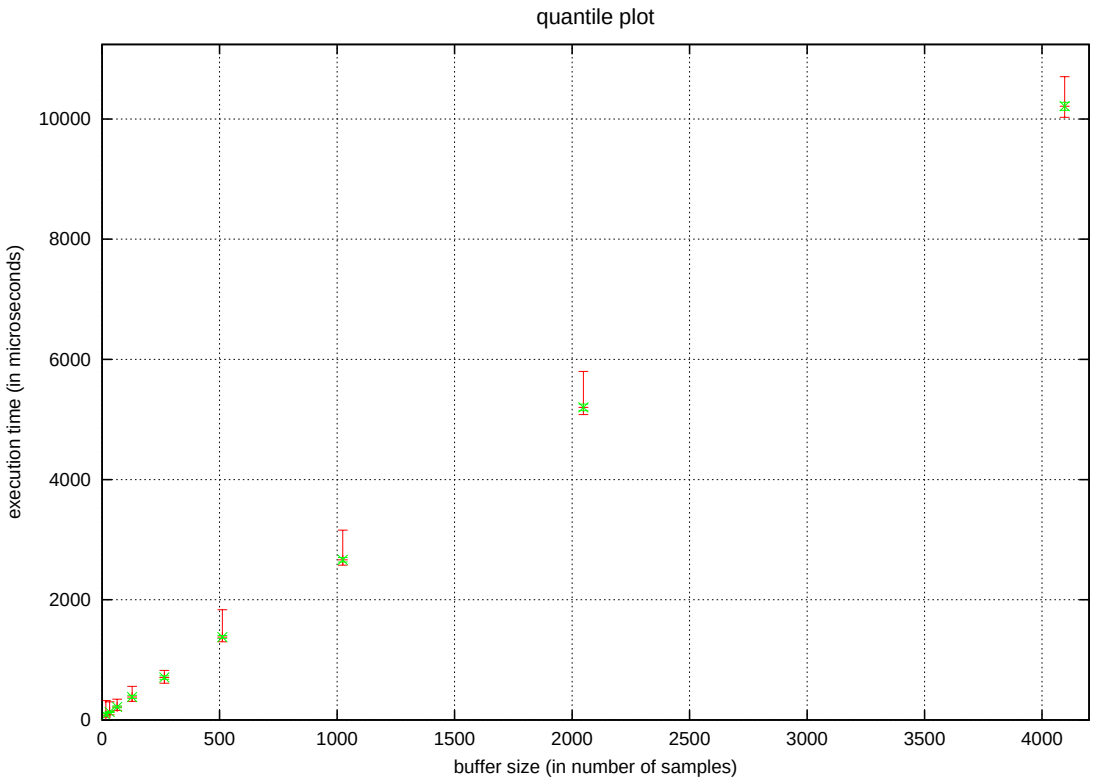

Figure 7.5: Quantile plot of the measured execution time of gverb for different buffer sizes (green: mean value, 0.1-quantile, 0.9-quantile, red: maximum and minimum).

the execution time of gverb is not or only weakly influenced by its input signal. The same experiment was repeated with the Dyson compressor (see Figure 7.9 and Table 7.5) and the multiband equalizer (see Table 7.6, the histograms are not depicted since they look exactly like the ones in Figure 7.10)—with the difference that a sine signal was added to the setup and ZynAddSubFX was substituted by a software MP3 media player. It can be concluded without any doubt that the execution time of the Dyson compressor does strongly depend on its input signal. Furthermore, the input signal seems to have only a minor influence on the execution time of the multiband equalizer, if at all.

7.3.3 Internal parameters

The internal state of a client may influence the execution time of its `process()` function. Measurements I did with Ardour (Jackdmp buffer size: 2048 frames, sample rate: 44.1 kHz) showed that its `process()` function's execution time had a mean value of 1603 μs when playing back one single stereo track, while it increases almost tenfold (mean value: 11234 μs) when mixing together twenty stereo tracks.

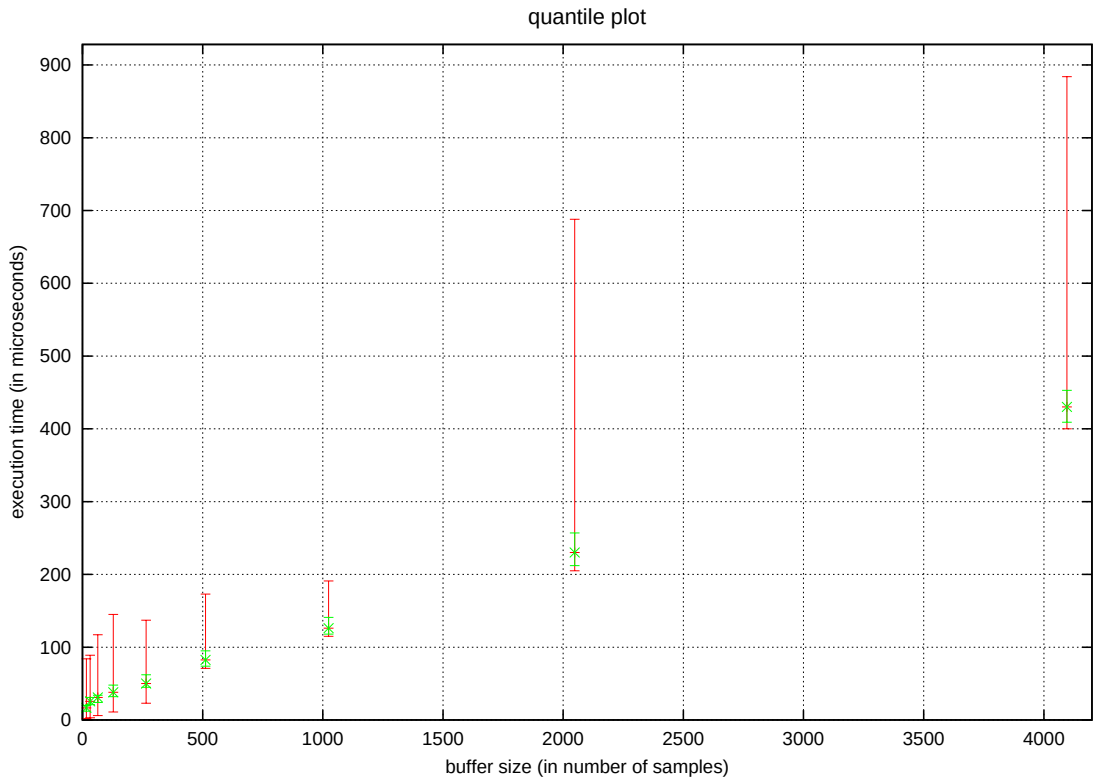

Figure 7.6: Quantile plot of the measured execution time of japa for different buffer sizes (green: mean value, 0.1-quantile, 0.9-quantile, red: maximum and minimum).

The histograms resulting if the execution time of the multiband equalizer is measured with different parameter sets (different values of the band gain) are depicted in Figure 7.10, their pairwise deviation parameters D_{AB} are listed in Tabular 7.7. The results indicate that the band parameters of this equalizer do not influence its execution time.

Furthermore, the execution time of DJ Flanger was analyzed with two different parameter sets: A configuration causing only a soft flanging effect, and a configuration that radically changes the processed signal. The histograms of these two measure series are depicted in Figure 7.11. The deviation parameter D_{AB} has a value of 2776. Therefore, I state that the execution time of this flanging effect filter is not or only weakly dependent of its internal parameters.

Finally, I measured the execution time of gverb with two different parameter sets. The first parameter set causes an extremly strong reverb effect, and an execution time mean value of 2809 μs, while gverb has an average execution time of 2592 μs when executed with a parameter set that only slightly changes the input signal. As the two histograms looked so similar to me at the first view, I increased the number of measure values to $K = 15000$. The calculated deviation parameter D_{AB}

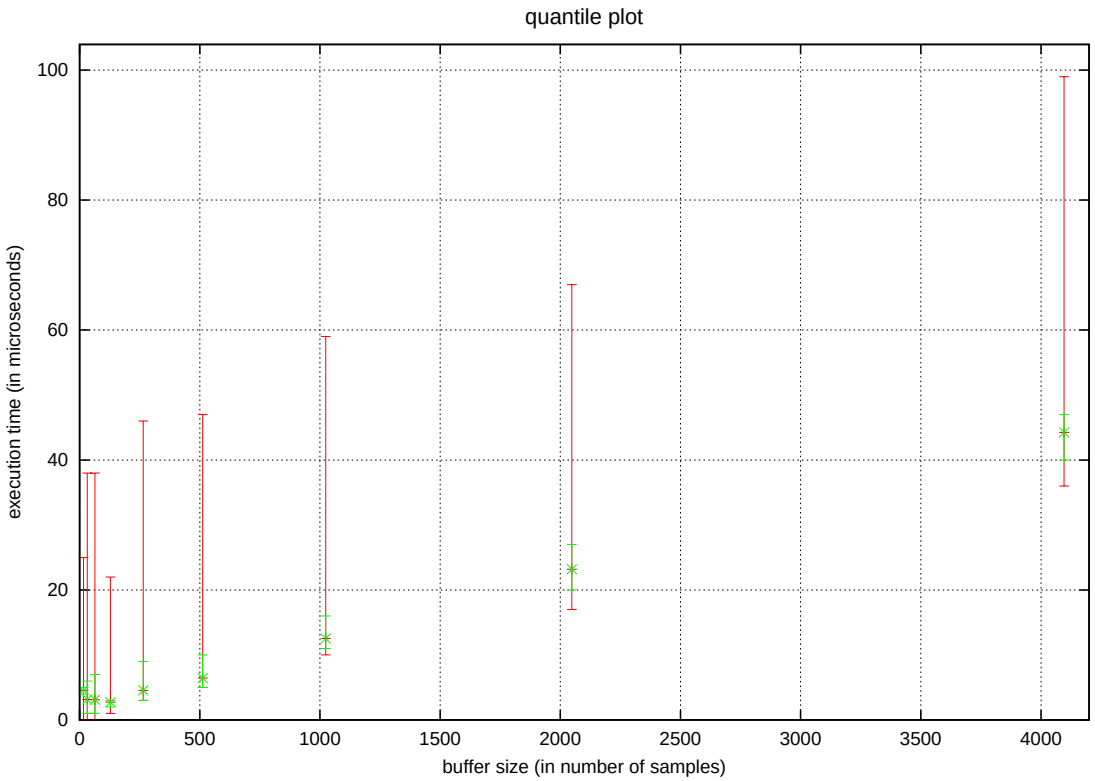

Figure 7.7: Quantile plot of the measured execution time of jack_metro for different buffer sizes (green: mean value, 0.1-quantile, 0.9-quantile, red: maximum and minimum).

counts 29946, which indicates that the execution time of gverb depends on its internal parameters.

7.3.4 Summary

The measurements presented in this section show that, depending on the specific client, its execution time can be influenced by the chosen buffer size, the input signal, or the internal client state. These results validate the assumption, on which the client parameterization methodology proposed in Section 6.2 is based. For the development of the methodology I assumed that only the clients themselves are capable of providing a correct characteristic, and that therefore these characteristics should be transmitted at runtime.

The dependencies uncovered by the measurements can be considered comprehensible and showed the behavior I expected: Neither the parameters nor the input signal significantly influence the execution time of a fixed-band equalizer. The execution time of a reverb filter does depend on its parameters, while it is

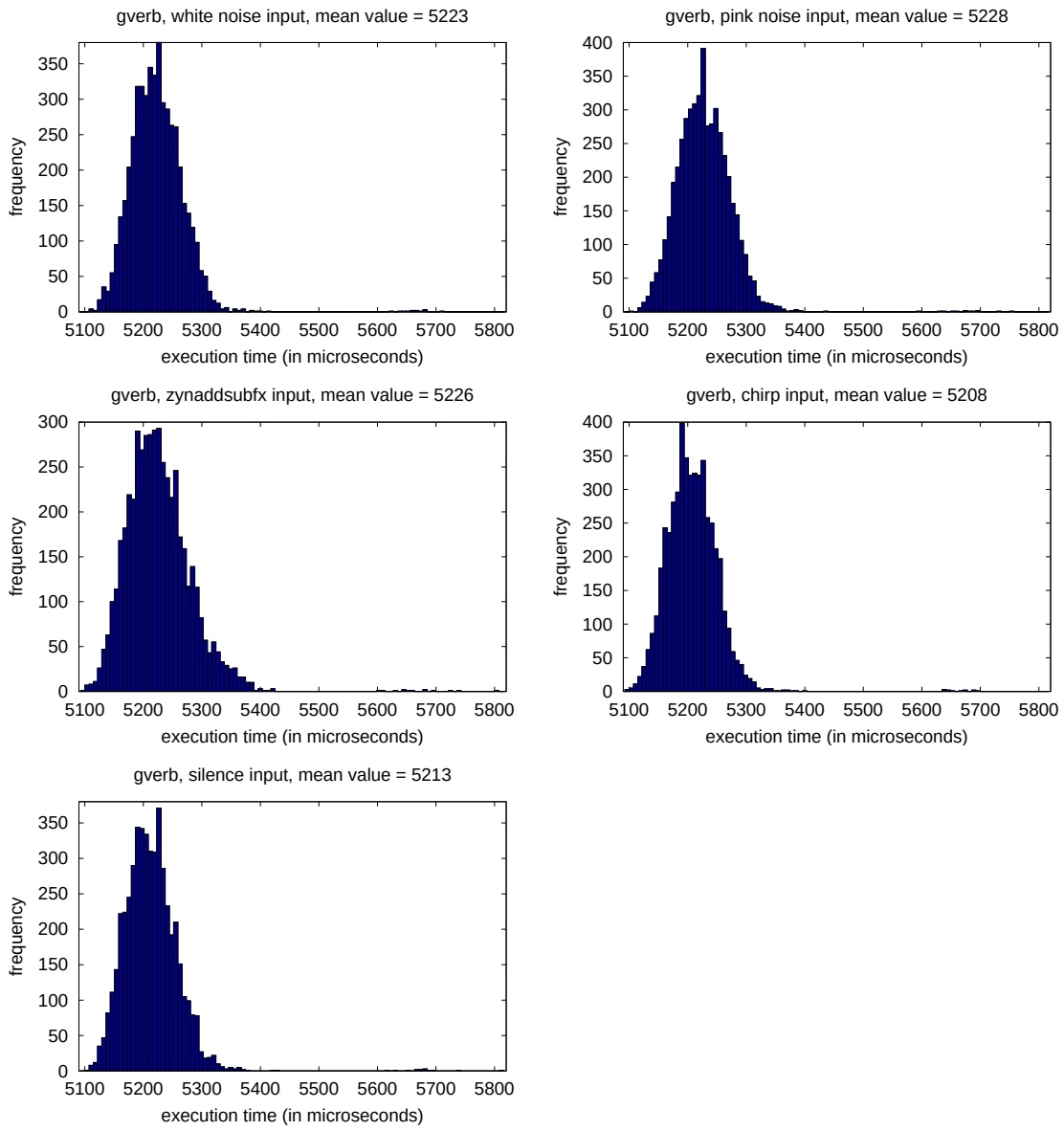

Figure 7.8: The histograms of the execution time of gverb resulting when it is fed with different input signals.

independent of the signal the filter processes. Whether silence or noise is fed into a compressor radically influences its execution time.

The soft real-time approach suggested in Section 6.2 may only be applied if the execution time of the handled clients is not influenced by their input signal. The measurements showed that this condition does not hold for all typical JACK clients. Consequently, this approach is only of limited use, and the intermediate approach presented in the same section promises a better balance between engineering effort and precision of the predicted timing behavior.

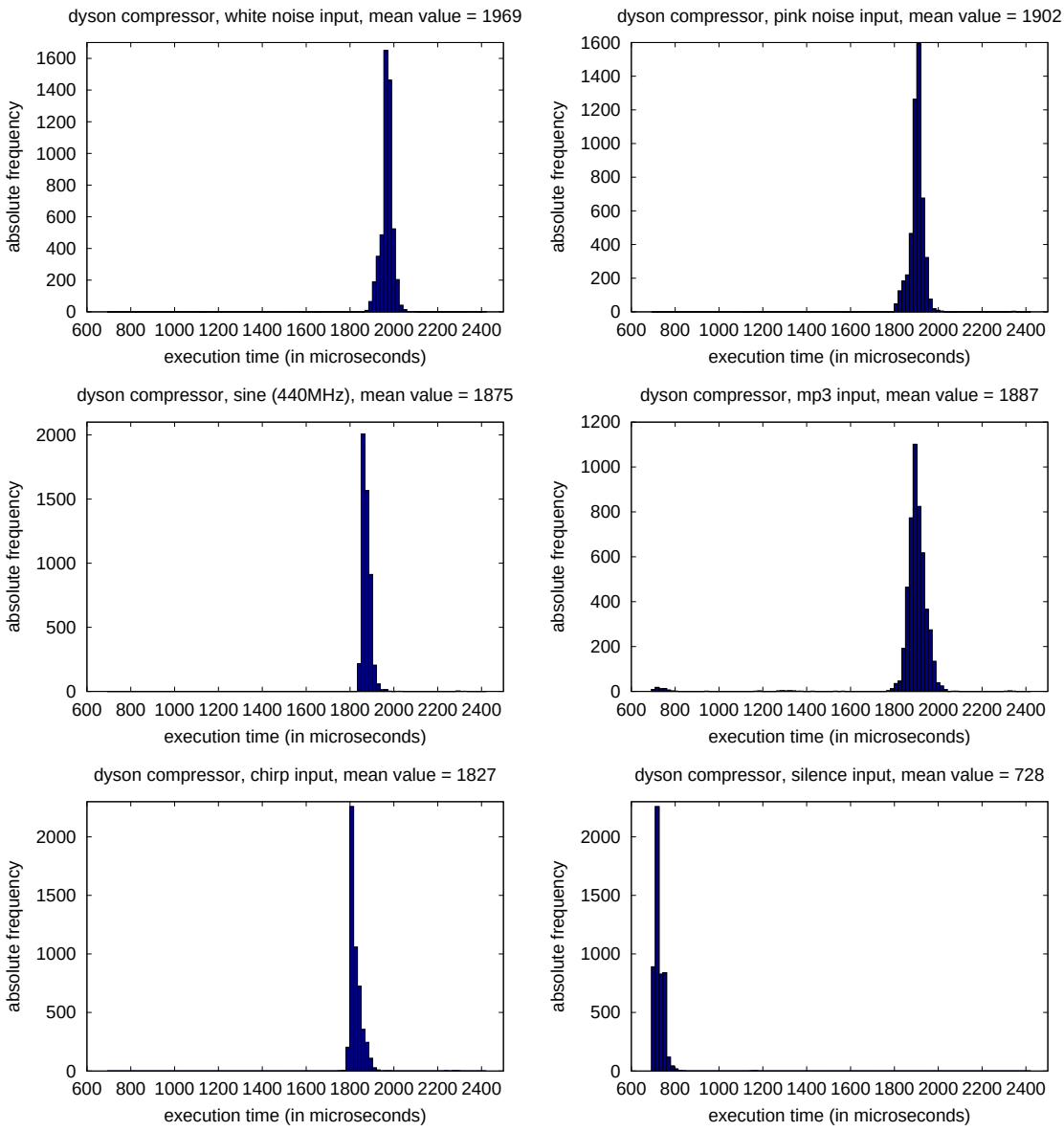

Figure 7.9: Histograms of the execution time of the Dyson compressor fed with different input signals.

Finally, these results indicate that the condition set by Paul Davis [34]—the execution time of the client's `process()` function must depend linearily on the JACK buffer size—holds. This too is a reasonable result, as an audio filter should produce the same output, independent of whether it processes its data in blocks of size n or of size $2n$.

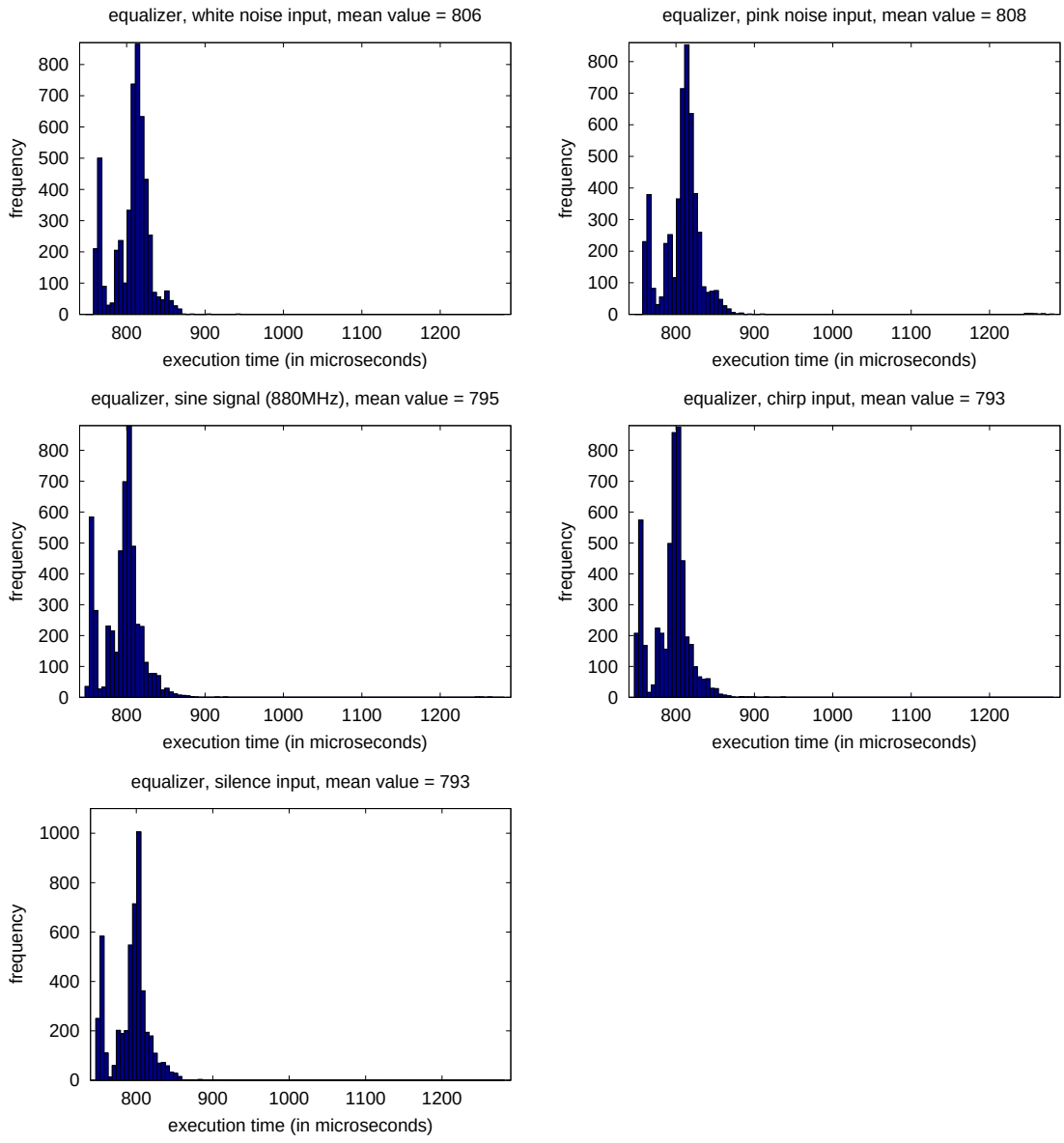

Figure 7.10: Histograms of the execution time of the multiband equalizer (Multiband EQ) measured with different parameter sets.

Table 7.1: Real-time performance measurement of Jackdmp running on DROPS with activated *deceit* bit.

jack_thru 1	mean value (in μs)	variance (in μs^2)
signal latency	8.383	0.352
sleep latency	4.858	0.208
sleep time	21955.441	23026.111
duration	24.120	0.750

jack_thru 2	mean value (in μs)	variance (in μs^2)
signal latency	10.969	0.272
sleep latency	3.866	0.194
sleep time	21950.256	23023.122
duration	30.295	0.727

jack_thru 3	mean value (in μs)	variance (in μs^2)
signal latency	51.803	0.890
sleep latency	4.098	0.161
sleep time	21949.974	23032.248
duration	30.342	0.682

jack_thru 4	mean value (in μs)	variance (in μs^2)
signal latency	51.262	0.860
sleep latency	1.860	0.158
sleep time	21955.460	23049.370
duration	27.093	0.935

jack_thru 5	mean value (in μs)	variance (in μs^2)
signal latency	46.312	1.094
sleep latency	3.378	0.242
sleep time	21950.553	23016.158
duration	30.482	1.115

jack_thru 6	mean value (in μs)	variance (in μs^2)
signal latency	9.918	0.351
sleep latency	2.159	0.146
sleep time	21961.099	22936.118
duration	21.153	0.692

jack_metro	mean value (in μs)	variance (in μs^2)
signal latency	8.953	0.478
sleep latency	2.585	0.251
sleep time	21967.290	22967.180
duration	14.544	2.311

Table 7.2: Real-time performance measurement of Jackdmp running on DROPS with deactivated *deceit* bit.

jack_thru 1	mean value (in μs)	variance (in μs^2)
signal latency	2.666	0.246
sleep latency	200.916	15.651
sleep time	20772.378	11029.376
duration	21.294	0.706

jack_thru 2	mean value (in μs)	variance (in μs^2)
signal latency	84.992	3.943
sleep latency	77.365	3.022
sleep time	20887.591	10983.369
duration	29.610	0.756

jack_thru 3	mean value (in μs)	variance (in μs^2)
signal latency	6.822	0.292
sleep latency	40.797	1.219
sleep time	20924.712	10964.925
duration	29.067	0.967

jack_thru 4	mean value (in μs)	variance (in μs^2)
signal latency	7.063	0.236
sleep latency	37.335	1.057
sleep time	20929.831	10971.005
duration	27.403	0.961

jack_thru 5	mean value (in μs)	variance (in μs^2)
signal latency	6.894	0.234
sleep latency	1.609	0.249
sleep time	20965.317	10950.180
duration	27.645	0.712

jack_thru 6	mean value (in μs)	variance (in μs^2)
signal latency	7.171	0.256
sleep latency	2.415	0.244
sleep time	20969.472	10959.716
duration	22.679	0.857

jack_metro	mean value (in μs)	variance (in μs^2)
signal latency	2.265	0.253
sleep latency	233.819	19.278
sleep time	20747.209	11039.080
duration	13.545	2.342

Table 7.3: Real-time performance measurement of Jackdmp running on a GNU/Linux distribution with a low latency patched Linux kernel.

jack_thru 1	mean value (in μs)	variance (in μs^2)
signal latency	27.176	10.639
sleep latency	11.909	5.846
sleep time	21313.631	2112.627
duration	26.310	7.189

jack_thru 2	mean value (in μs)	variance (in μs^2)
signal latency	101.950	51.225
sleep latency	8.366	9.006
sleep time	21300.241	2542.593
duration	43.235	34.111

jack_thru 3	mean value (in μs)	variance (in μs^2)
signal latency	31.587	14.066
sleep latency	10.417	10.038
sleep time	21301.321	2330.557
duration	40.109	18.454

jack_thru 4	mean value (in μs)	variance (in μs^2)
signal latency	98.056	83.026
sleep latency	10.627	8.043
sleep time	21300.167	2972.614
duration	41.044	49.297

jack_thru 5	mean value (in μs)	variance (in μs^2)
signal latency	103.299	77.969
sleep latency	2.126	1.024
sleep time	21304.771	2749.889
duration	44.943	42.831

jack_thru 6	mean value (in μs)	variance (in μs^2)
signal latency	31.228	27.013
sleep latency	3.260	3.839
sleep time	21310.492	3209.308
duration	38.081	48.468

jack_metro	mean value (in μs)	variance (in μs^2)
signal latency	38.912	23.074
sleep latency	9.802	3.721
sleep time	21317.120	1957.395
duration	24.932	5.532

Table 7.4: Pairwise differences D_{AB} of the histograms of the execution time resulting when gverb is fed with different input.

input signal	white noise	pink noise	ZynAddSubFX	linear chirp	silence
white noise	0	618	1062	1464	1092
pink noise	618	0	1042	1888	1480
ZynAddSubFX	1062	1042	0	1564	1206
linear chirp	1464	1888	1564	0	578
silence	1092	1480	1206	578	0

Table 7.5: Pairwise differences D_{AB} of the histograms of the execution time resulting when the Dyson compressor is fed with different input.

input signal	white noise	pink noise	sine (440 MHz)	mp3	linear chirp	silence
white noise	0	7916	9298	7080	9780	10000
pink noise	7916	0	5858	2162	8078	10000
sine (440 MHz)	9298	5858	0	4728	8074	10000
mp3	7080	2162	4728	0	7934	9876
linear chirp	9780	8078	8074	7934	0	10000
silence	10000	10000	10000	9876	10000	0

Table 7.6: Pairwise differences D_{AB} of the histograms of the execution time resulting when the multiband equalizer is fed with different input.

input signal	white noise	pink noise	sine (880 MHz)	linear chirp	silence
white noise	0	432	4980	5436	5524
pink noise	432	0	4708	5254	5344
sine (880 MHz)	4980	4708	0	760	1048
linear chirp	5436	5254	760	0	682
silence	5524	5344	1048	682	0

Table 7.7: Pairwise differences D_{AB} of the histograms of the execution time of the multiband equalizer (Multiband EQ) measured with different parameter sets.

parameter set	1	2	3	4	5
1	0	1434	778	1252	1602
2	1434	0	1420	1298	1284
3	778	1420	0	1524	1424
4	1252	1298	1524	0	2092
5	1602	1284	1424	2092	0

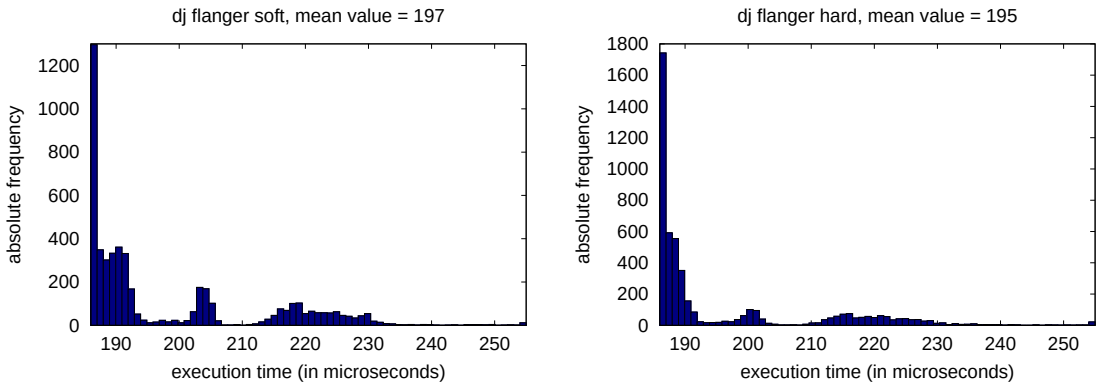

Figure 7.11: Histograms of DJ Flanger's execution time, measured with two different parameter sets.

8 Summary, conclusion, and outlook

8.1 Summary and conclusion

Several available free software audio solutions were analyzed, and Jackdmp—a C++ reimplementation of the renowned JACK Audio Connection Kit—was selected as the most appropriate solution for a real-time audio architecture on DROPS. The JACK sound architecture provides the lowest processing latency possible on a desktop computer for a given set of sound card parameters. It reduces the latency jitter caused by software to zero and synchronizes streams at sample accuracy.

A real-time admission scheme for JACK clients is proposed. The execution time of different typical JACK clients was analyzed with measurements to validate the assumptions the proposal is based on, but also to gain further knowledge about their timing behavior. The measurements showed that the condition set by Paul Davis—the time to process a client must be a linear function of the buffer size—holds for all tested clients.

Jackdmp was ported to DROPS. The developed design of the port and its implementation is documented here. Measurements showed that—although the real-time performance of the Linux kernel is continously being improved in the mainline and on special external branches—DROPS can provide a signaling latency that is two times lower on average than the values that can be achieved on the same machine running with a low latency patched Linux kernel. Thus, it can be stated that DROPS is well-suited for real-time audio processing and that the pursued path to use it as the foundation of a truly real-time capable audio workstation should be followed.

8.2 Outlook

DROPS shows outstanding real-time performance in running Jackdmp. It would be therefore interesting to investigate how the Jackdmp port can be used in combination with L^4Linux to run JACK applications for Linux on DROPS, in a way that the real-time critical part of the application runs natively on DROPS without being disturbed by the non-real-time part running on L^4Linux. Because the handling of the JACK related threads is accomplished by JACK on client-side as

well, in the best case it should be possible to simply start an unmodified JACK application for Linux under L^{4}Linux with the JACK client library of this port. A deeper analysis then needs to be performed how well the aspired separation of the real-time and non-real-time JACK client parts works, and whether the real-time performance achieved when running Jackdmp alone can be preserved under high load. It would be interesting to see, whether it is possible to use an audio production system based on DROPS and Jackdmp as a tool in a productive environment.

DDE needs to be extented with an infrastructure that enables ALSA to run in `mmap` mode. As an interim solution a JACK–ALSA backend operating in `read()/write()` mode could be written. The performance of the ALSA port to DROPS has to be analyzed.

Furthermore, the admission scheme proposed in Chapter 6 could be implemented. An appropriate method for obtaining the values transmitted to the JACK admission server as the clients' worst case execution time has to be found for this purpose. Using a priori knowledge about a client's algorithm could help to improve the determination of the client's execution characteristic—possibly using a hybrid approach of measurement and calculation.

Neither MIDI (Musical Instrument Digital Interface) nor any of its advanced successors have been examined in this project. It would be interesting to know in what way the results of the thesis could be adopted to them.

As regards the long-term vision presented in Section 2.1, another promising approach would be researching what real-time features of the Mach microkernel have remained in Mac OS X and whether they can be used to run JACK as a real-time application on a desktop computer with Mac OS X.

Finally, the Jackdmp port could be adapted to the L4 Runtime Environment (L4RE), the successor of L4Env, which will be released in the near future.

Bibliography

[1] LADSPA. http://www.ladspa.org/. 15

[2] Advanced Linux Sound Architecture. http://www.alsa-project.org/. 11

[3] Apple Quicktime. http://www.apple.com/quicktime/. 16

[4] Ardour sequencer. http://ardour.org/. 13

[5] Audacious. http://audacious-media-player.org/. 15

[6] Enlightened Sound Daemon. ftp://ftp.gnome.org/pub/GNOME/sources/esound/0.2/. 14

[7] freedesktop.org project. http://www.freedesktop.org/. 13

[8] Genode Operating System Framework. http://genode.org/. 35

[9] Jack Audio Connection Kit. http://jackaudio.org/. 13

[10] Jack Rack project homepage. http://jack-rack.sourceforge.net/. 46

[11] libao: a cross platform audio library. http://www.xiph.org/ao. 15

[12] Linux Audio projects at Kokkini Zita. http://www.kokkinizita.net/linuxaudio/. 45

[13] MPlayer - The Movie Player. http://www.mplayerhq.hu/. 15

[14] Open Sound System. http://www.opensound.com/. 14, 15

[15] GStreamer open source multimedia framework. http://www.gstreamer.net/. 13

[16] Real-time preemption Linux patch, mainly developed by Ingo Molnár. http://www.kernel.org/pub/linux/kernel/projects/rt/. 45

[17] Resource Description Framework (RDF). http://www.w3.org/RDF/. 16

[18] SLOCCount project hompage. http://www.dwheeler.com/sloccount/. 46

[19] Small ALSA Library. http://www.alsa-project.org/main/index.php/SALSA-Library. 11

[20] SWH Plugins project hompage. http://plugin.org.uk/. 46

[21] The analog Real time synthesizer. http://www.arts-project.org/. 14

[22] The Disposable Soft Synth Interface. http://dssi.sourceforge.net/. 16

[23] The Enlightenment window manager. http://www.enlightenment.org/. 14

[24] The K Desktop Environment. http://www.kde.org/. 14, 16

[25] The Phonon multimedia API. http://phonon.kde.org/. 16

[26] The GNOME desktop environment. http://www.gnome.org/. 14

[27] The ZynAddSubFX open source software synthesizer. http://zynaddsubfx.sourceforge.net/. 51

[28] VLC media player. http://www.videolan.org. 16

[29] X MultiMedia System (XMMS). http://www.xmms.org/. 15

[30] xine - A free video player. http://www.xine-project.org/. 15, 16

[31] Ronald Aigner. DICE *Version 3.3.0. User's Manual*. Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 2007. http://www.inf.tu-dresden.de/content/institutes/sya/os/forschung/projekte/dice/manual-3.3.0.pdf. 10, 26, 33

[32] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill Framework for Virtual Memory Diversity. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC2001)*, pages 3–10, Gold Coast, Australia, January 2001. https://eprints.kfupm.edu.sa/71199/1/71199.pdf. 28

[33] Ross Bencina and Phil Burk. PortAudio - an Open Source Cross Platform Audio API. In *Proceedings of the International Computer Music Conference*, pages 263–266. International Computer Music Association, 2001. http://www.audiomulch.com/~rossb/writings/portaudio_icmc2001.pdf. 15

[34] Paul Davis. The Jack Audio Connection Kit, 2003. Presentation given at the Linux Audio Developers' Conference (LAC) in Karlsruhe. Slides: `http://lad.linuxaudio.org/events/2003_zkm/slides/paul_davis-jack/title.html`, audio recording: `http://lad.linuxaudio.org/events/2003_zkm/recordings/paul_davis-jack.ogg`. 21, 56

[35] Gerd Grießbach. USB for DROPS. Diploma thesis, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 2003. `http://os.inf.tu-dresden.de/papers_ps/griessbach-diplom.pdf`. 36

[36] Claude-Joachim Hamann. On the Quantitative Specification of Jitter Constrained Periodic Streams. In *Proceedings of MASCOTS' 97*, Haifa, Israel, 1997. `http://os.inf.tu-dresden.de/papers_ps/mascots2.pdf`. 3, 44

[37] Claude-Joachim Hamann, Jork Löser, Lars Reuther, Sebastian Schönberg, Jean Wolter, and Hermann Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001. `http://os.inf.tu-dresden.de/papers_ps/rtss01.pdf`. 43

[38] Claude-Joachim Hamann, Michael Roitzsch, Lars Reuther, Jean Wolter, and Hermann Härtig. Probabilistic Admission Control to Govern Real-Time Systems under Overload. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS 07)*, Pisa, Italy, July 2007. `http://os.inf.tu-dresden.de/papers_ps/hamann07-qrms.pdf`. 43

[39] Claude-Joachim Hamann and Steffen Zschaler. Scheduling Real-Time Components Using Jitter-Constrained Streams. In *Proceedings of 10th IEEE The Enterprise Computing Conference (EDOC) Hong Kong*, 2006. `http://os.inf.tu-dresden.de/papers_ps/aquserm2006`. 3, 44

[40] Michael Hohmuth. Linux-Emulation auf einem Mikrokern. Diplomarbeit, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 1996. In German; with English slides. `http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/`. 6

[41] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`. 10

[42] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time*

Systems (PART '98), Adelaide, Australia, September 1998. http://os.inf.tu-dresden.de/papers_ps/part98.ps. 6

[43] Adam Lackorzynski. L^{4}Linux on L4Env. Großer Beleg, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 2002. http://os.inf.tu-dresden.de/papers_ps/adam-beleg.pdf. 6

[44] Nelson Posse Lago. *Distributed Real-Time Audio Processing*. Extended english abstract of the phd thesis *Processamento Distribuído de Áudio em Tempo Real*, University of São Paulo, Department of Computer Science, 2004. http://gsd.ime.usp.br/~lago/masters/extended_abstract.pdf. 3

[45] Stéphane Letz, Yann Orlarey, and Dominique Fober. Jack audio server for multi-processor machines. In *Proceedings of the International Computer Music Conference*, pages 1–4, 2005. http://www.grame.fr/pub/Jackdmp-ICMC2005.pdf. 25

[46] Stéphane Letz, Yann Orlarey, and Dominique Fober. jackdmp: Jack server for multi-processor machines. In *LAC2005 Proceedings. 3rd International Linux Audio Conference*, pages 29–36, 2005. http://www.grame.fr/pub/Jackdmp-lac2005.pdf. 25

[47] Stéphane Letz, Yann Orlarey, Dominique Fober, and Paul Davis. Jack Audio Server: MacOSX port and multi-processor version. In *Proceedings of the first Sound and Music Computing conference - SMC'04*, pages 177–183, 2004. http://www.grame.fr/pub/SMC-2004-033.pdf. 25

[48] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995. http://l4ka.org/publications/1995/ukernel-construction.pdf. 10

[49] Jochen Liedtke. L4 reference manual (486, Pentium, PPro). Technical report, GMD — German National Research Center for Information Technology, September 1996. http://os.inf.tu-dresden.de/L4/l4refx86.ps.gz. 28

[50] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, April 2000. 5

[51] Jork Löser and Michael Hohmuth. Omega0 – a portable interface to interrupt hardware for L4 systems. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999. http://os.inf.tu-dresden.de/~jork/papers/omega0.pdf. 27

[52] Lennart Poettering. Cleaning up the linux desktop audio mess. In *Proceedings of the Linux Symposium*, 2007. `http://ols.108.redhat.com/2007/Reprints/poettering-Reprint.pdf`. 15

[53] Michael Roitzsch and Hermann Härtig. Ten Years of Research on L4-Based Real-Time Systems. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006. `http://os.inf.tu-dresden.de/papers_ps/roitzsch06ten_years_rt.pdf`. 9, 36

[54] Sergio Ruocco. Real-time programming and L4 microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006. `http://ertos.nicta.com.au/publications/papers/Ruocco_06.pdf`. 6

[55] Henry Spencer and Geoff Collyer. `#ifdef` considered harmful or portability experience with C news. In *USENIX Summer 1992 Technical Conference*, pages 185–198, 1992. `http://doc.cat-v.org/henry_spencer/ifdef_considered_harmful.pdf`. 31

[56] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Diplomarbeit, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 2004. `http://os.inf.tu-dresden.de/papers_ps/steinberg-diplom.pdf`. 42

[57] Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme. — *L4Env — An Environment for L4 Applications*. `http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf`. 10

[58] Dirk Vogt. USB for the L4 Environment. Großer Beleg, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 2008. `http://os.inf.tu-dresden.de/papers_ps/vogt-beleg.pdf`. 11, 27, 34

[59] Michael Voigt. Introduction to TUD:OS. *OSnews.com*, 2006. `http://www.osnews.com/story/15814/Introduction-to-TUD-OS/`. 9

[60] Wikipedia. System Management Mode — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=System_Management_Mode&oldid=278498133`, 2009. 43

[61] Wikipedia. Time Stamp Counter — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Time_Stamp_Counter&oldid=280298552`, 2009. 34, 45

[62] Wikipedia. Virtual Studio Technology — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Virtual_Studio_Technology&oldid=267620453, 2009. 15

[63] Wikipedia. Zero configuration networking — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Zero_configuration_networking&oldid=270284962, 2009. 15

Appendix A

Implementation details of the ALSA port

1. **Symbol clash:** As we want to link the ALSA kernel sources together with SALSA-Lib—which is uncommon, there is the problem of the symbols defined in both of the libraries, which clash when both parts are being linked together. We use a little symbol renaming hack (see `lib/alsa-kernel-lib/src/Makefile`) to work around it.

2. **ALSA sources:** In `l4/pkg/dde/linux26/contrib/include/linux/utsrelease.h` we find the Linux kernel version number this version of DDE emulates. To achieve the best possible compatibility between DDE and ALSA, we took the ALSA sources for the ALSA kernel library from exactly this version of the linux kernel.

3. **Linux kernel build system:** If one wants to compile ALSA on DROPS for a specific sound card and with a particular set of options, it might be hard to determine the list of source files to compile and symbols to define. Therefore, we ask the Linux kernel build system for help in the following steps.

   a) Go to the source tree of the downloaded Linux kernel, execute `make menuconfig` and configure the ALSA kernel part the way it should be on DROPS.

   b) Then issue:

   ```
   make sound | sed -n '/sound\//p' |
   sed '/^ *LD/d' |
   sed 's/.*\(sound\/.*\)\.o/\1.c \\/g'
   ```

   This leads to the list of source files for `alsa/lib/alsa-kernel-lib/src/Makefile`

   c) To find out, which of the symbols in DDE `autoconf.h` have to be redefined

      i. In the DDE package directory do:

      ```
      find -name autoconf.h |
      ```

```
xargs egrep 'CONFIG_SND|CONFIG_SOUND' |
sed -n '/#define/p' |
sed 's/.*#define\(.*\) .*/#undef\1/g'
```

ii. In the Linux tree issue:

```
cd include/linux/; cat autoconf.h |
egrep 'CONFIG_SND|CONFIG_SOUND'
```

This leads us to the list of symbols to be (un-)defined in `lib/alsa-kernel-lib/include/linux/autoconf.h`.