Karlsruhe Institute of Technology

Comparison of Parallelization Strategies for a Real-time Audio Application

Diploma Thesis of

Jochen Bieler

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	M.Sc. Marc Aurel Kiefer
Second advisor:	Dr. Frank Padberg

Duration: February 10, 2014 – August 9, 2014

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, August 8, 2014

. . . .

(**Jochen Bieler**)

Kurzfassung

Da mittlerweile in den allermeisten Computer mehrere Prozessoren verbaut werden, muss Software parallelisiert werden, um die maximale Leistung herausholen zu können. Diese Arbeit beschreibt eine Fallstudie über eine Echtzeit-Audio-Anwendung, die durch einen Task-Graphen strukturiert ist. Nach einer tiefgreifenden Analyse der Softwarestruktur und Performance werden zwei alternative Parallelisierungen für die Ausführung des Task-Graphen implementiert und ihre Leistungsdaten vermessen. Die neuen Parallelisierungen werden mit der original Parallelisierung verglichen und in Bezug gesetzt zum theoretischen Maximum, welches durch die Echtzeitbedingung gegeben ist. Außerdem werden Möglichkeiten zur weiteren Parallelisierung und Beschleunigung beleuchtet.

Abstract

Since nowadays almost all computers carry multiple processors, software has to be parallelized in order to maximize the performance. This thesis describes a case study on a real-time audio application, which is structured using a task-graph. After an in-depth analysis of the software's architecture and performance, two alternative parallelizations for the execution of the task-graph are presented and their performance is measured. The new parallelization strategies are evaluated against the original strategy and set in relation to the theoretical maximum, given by a real-time constraint. Finally, additional possibilities for parallelization are presented.

Danksagung

Ich möchte mich bei allen bedanken, die mich beim Anfertigen dieser Diplomarbeit unterstützt haben!

Das sind zuerst meine Eltern Cornelia und Harald. Vielen Dank für eure Unterstützung! Ebenso danke ich meiner Freundin Julia für die Unterstützung, du warst mir eine große Hilfe. Desweiteren gilt mein Dank, Marc, dem Betreuer der Arbeit auf Seiten des KIT. Auf Seiten der Firma danke ich meinen Arbeitskollegen, insbesondere Simon, Serkan, André und Joachim für die Unterstützung. Zuletzt möchte ich mich bei Patryk, Daniel und Jamil für das Korrekturlesen und viele hilfreiche Tipps bedanken.

Contents

1. Introduction

Problem Statement

Though almost all computer systems nowadays use multiple processors, software has to be parallelized to work at maximum performance. Especially performance-intensive real-time software should make wise use of the available computing resources to satisfy its real-time condition. Furthermore, the software in this study is usually running on mobile computers, where computing resources are scarce and should be utilized efficiently.

The word cloud in figure 1.1 shows the most common words in this thesis to give the reader an initial idea and picture of the contents.

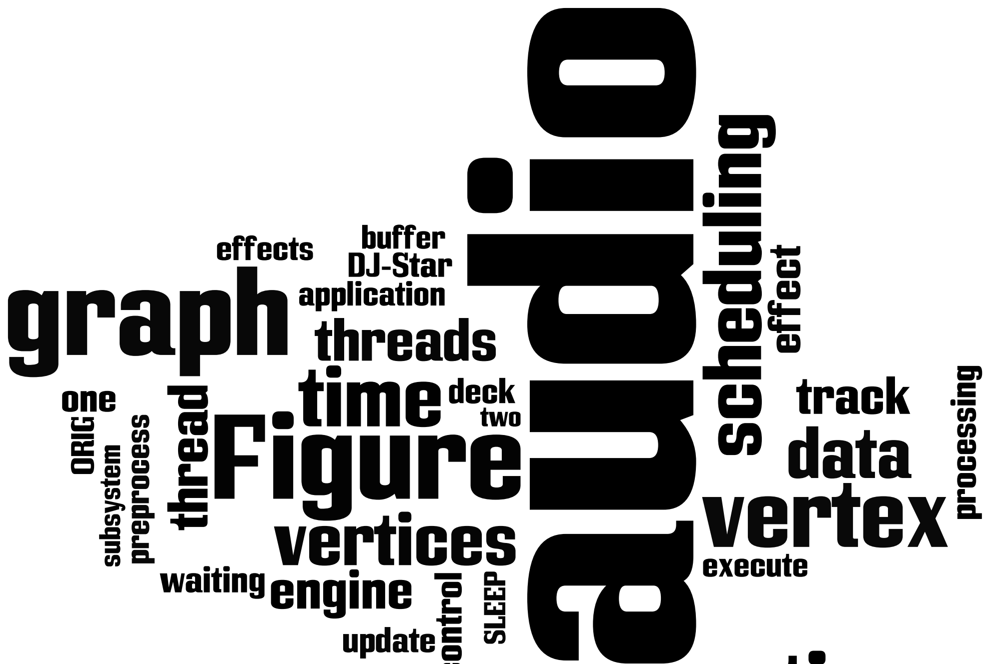


Figure 1.1.: Word cloud showing the most common words in this thesis by size.

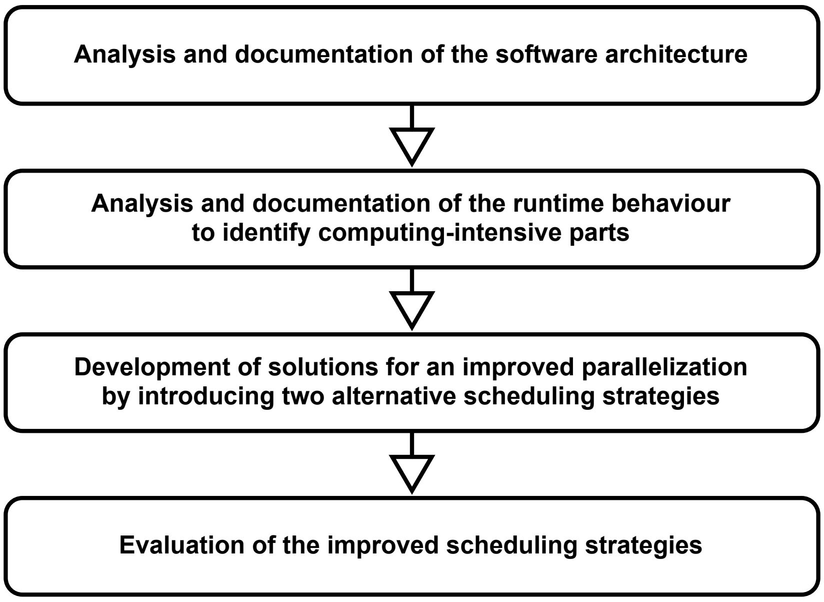

Analysis and documentation of the software architecture
Analysis and documentation of the runtime behaviour to identify computing-intensive parts
Development of solutions for an improved parallelization by introducing two alternative scheduling strategies
Evaluation of the improved scheduling strategies

Figure 1.2.: Approach of this thesis.

Idea and Goal

This thesis is a case study on a commercial real-time audio application. The software features more than 700.000 lines of code, and evolved over more than ten years, making the parallelization a complex undertaking, and providing a challenging scenario for the application of scholarly procedures in a commercial world.

The goal of this thesis is to compare different parallelizations of the application, and to compare them against the original parallel version.

Approach

The approach of this thesis is presented in figure 1.2. The first step is to analyze the application's architecture and document the results. Next, the application's runtime behavior is analyzed, and the results are again documented. The analysis reveals two potentials, where the parallelization could be improved, for which two enhanced scheduling strategies are developed. Finally, the evaluation compares the enhanced scheduling strategies against the original strategy in terms of performance.

Non-Disclosure Agreement

Because of a non-disclosure agreement with the collaborating company, we can neither reveal the name of the company nor the name of the actual application. To ensure this agreement, we use screenshots of various DJ applications in chapter 3. Furthermore, we anonymized all code symbols that could reveal the connection to a specific DJ application throughout the whole thesis. To refer to the software, we renamed it to DJ-Star.

2. Background and Related Work

This chapter establishes the background for the thesis by introducing the art form of DJing, giving an introduction to scheduling, and illustrating the parallel master/worker design pattern. It then concludes with an overview of related work.

2.1. Background

This section explains the background and concepts used in this thesis.

2.1.1. DJing

A disc jockey (DJ) is defined as "a person who mixes recorded music for an audience" by Wikipedia[1] [Wik]. Likewise DJing is the art form of mixing recorded music for an audience.

Traditional Equipment

In the beginning of DJing in the 20th century, recorded music was only available on vinyl records. These vinyl records were played back on turntables, like *TT1* and *TT2* in figure 2.1. The two turntables are connected with a mixer MX, allowing to blend the music from *TT1* and *TT2* together in continuously adjustable mixing ratios. The mixed audio signal is then sent to the speakers *SP*.

DJing Procedure

The DJ starts by playing a record on *TT1*. While the record is playing, he prepares the next record by placing it on TT2. Near the end of the playback of the record on *TT1*, he then starts the playback of *TT2* and blends both records together with the help of MX, without having any silence appearing between the two tracks.

[1]Wikipedia was chosen over the encyclopedias Brockhaus and Duden because their definition were found to be outdated.

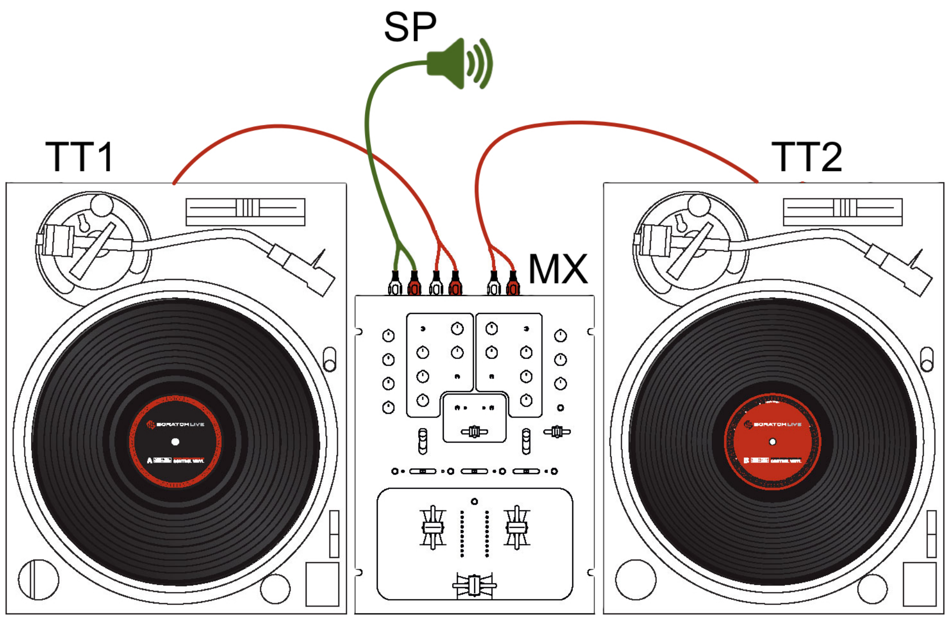


Figure 2.1.: Traditional DJ setup based on [Ran].

Digital Equipment

Digital DJing aims at delivering the same workflow as by using traditional analog equipment, but the vinyl records are replaced by digital audio tracks stored on a hard disk drive. This makes the carrying of tens or hundreds of vinyl records to the venue unnecessary. Additionally the digital collection is easier to maintain, and searching individual audio tracks is much more efficient.

For digital DJing, a multi-channel *audio interface* is added to the traditional equipment, so that all audio signals are looped through the computer. The audio signal of the turntables is sent to the computer, and the audio signal for the speakers is sent by the computer. The vinyl records are replaced with *purpose-made control vinyl records* providing a *control signal* to enable the control of the digital audio tracks with regular turntables.

2.1.2. Scheduling Model

The scheduling model in this thesis is based on [Pin12]. The author Pinedo defines scheduling as "the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives".

Scheduling Problem Classification

Scheduling problems are usually classified using a three field scheme $\alpha \mid \beta \mid \gamma$ to distinguish between the vast amount of different scheduling problems, with α denoting the machine environment, β expressing a variety of problem characteristics, e.g., dependencies among tasks and γ representing the objective function.

Formal Scheduling Definition

A scheduling problem consists of n *jobs* $J_i (i = 1, ..., n)$ that have to be processed on a *parallel machine* P with m *processors* $M_j (j = 1, ..., m)$. A *schedule* is a mapping of each job onto a processor and a start time. A job J_i has a *release time* r_i, a *deadline* d_i and a *finishing time* C_i. A job has *precedence relations prec* to other jobs. This is represented by a directed acylic graph $G = (V, A)$, where the set $V = 1, ..., n$ corresponds with the jobs, and a precedence constraint $(i, k) \in A$ means that job J_i has to be completed before job J_k can start.

The objective function in this thesis is the *makespan* $C_{max} = max_{i \in J} C_i$, or in other words: the completion time of the graph. Accordingly, the scheduling problem is classified as $Pm \mid prec; d_i \mid C_{max}$.

Deterministic scheduling

In deterministic scheduling, all problem data is known in advance. This includes especially the *processing time* p_i of a job J_i. The resulting schedule can then be calculated upfront in an offline manner.

Online Scheduling

Online Scheduling is an extension to the offline characteristic of the deterministic scheduling. It is applied if the processing times p are not known beforehand. With online scheduling, the decision maker decides at every completion of a job which job to process next.

2.1.3. Master/Worker Pattern

The Master/Worker Pattern (based on [MSM04, 143-152]) is a pattern to effectively load balance tasks between multiple processors when the runtime of the individual tasks is not known in advance. The pattern, summarized in figure 2.2, involves two actors, one *master* and one or more *worker(s)*. “The master initiates the computation and sets up the problem. It then creates the bag of tasks” and launches the workers. In this variation the master subsequently turns into a worker, whereas in the classic version he would sleep until all computation is done (fig. 5.14 in [MSM04]). The master then waits until all workers are finished with their computations, collects the results and terminates the computation.

The workers start to compute the results in a loop by fetching a new task, computing the results and checking for more available tasks in each iteration. If more work is available, he continues the loop by fetching a task, computing the results and so on. Finally, when the bag of tasks is empty, he stops the computations and exits, which is again recognized by the master.

The bag of tasks is implemented with a single shared queue, filled by the master with all tasks before launching the workers. This way, he can efficiently determine the completion of the computations.

2.2. Related Work

This section provides an overview of other work related to this thesis.

2.2.1. Super Collider

Super Collider is a real-time audio synthesis engine that consists of a client and a server. The server is responsible for synthesizing the audio. It uses a graph structure with nodes and directed connections to express transformations on the audio signal and the flow of the audio data. The synthesis server is not capable of using multiple processors. Supernova [Ble11] is a replacement for the original synthesis server, capable of using more than one processor. The parallelization however is exposed to the user. The author claims that the parallelization can not be done automatically.

In our scenario, we do not want the parallelism to be exposed to the user. This is possible because the user is not able to directly manipulate the audio synthesis tree.

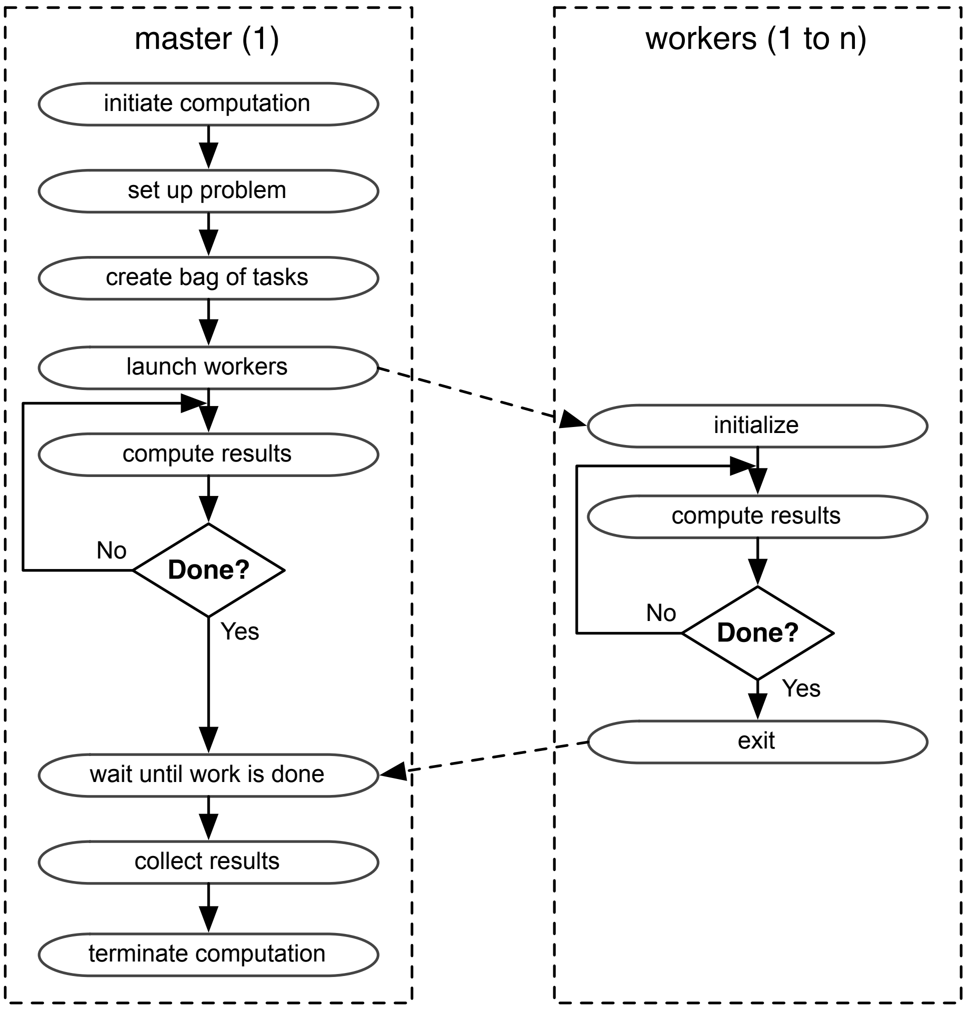


Figure 2.2.: Master/Worker pattern based on [MSM04]

Mixxx	DJ-Star
2 Decks	4 Decks
Limited sampler deck support	Full sampler deck Support
Effects not chainable	Effects are chainable
Basic tempo adjustment algorithm	High-quality tempo adjustment algorithm

Table 2.1.: Main differences in the feature set of Mixxx and DJ-Star

2.2.2. Mixxx

Mixxx [Mix] is an open source DJing application, similar to the application DJ-Star in this case study. Table 2.1 shows the differences between the features of the two applications. Mixxx does not use a directed acyclic graph for audio representation but a hardcoded audio data flow, and the audio related computations are not processed concurrently. This limitation is not a problem for Mixxx, because it does not need that much computational power due to the more limited feature set.

2.2.3. MCFlow

MCFlow is a real-time, multi-core-aware middleware for dependent task graphs [HGL12]. It uses a directed acyclic graph structure similar to the one used in DJ-Star but MCFlow's scheduling is done offline, in contrast to our work, where we use online scheduling to dynamically load-balance the work.

2.2.4. Tesselation OS

Tesselation OS [CSB+11] is an experimental operating system that supports quality of service guarantees. This is very valuable for a real-time audio application, since if you miss a deadline, the audio signal will be distorted. Programs running on Tesselation OS are characterized by a directed acyclic graph with plug-ins for nodes. In contrast to our application, the user is able to rearrange and exchange them as needed. But the main difference is that we have to run our application on general purpose operating systems such as mac os, windows and linux. If we want to ensure quality of service guarantees, we cannot rely on the operating system, but have to take care of them ourselves.

3. Software Description

This chapter illustrates all features of the application DJ-Star, whose purpose it is to load, manipulate, filter and mix audio data. For anonymization purposes, the screenshots are mixed between different applications. This chapter works with partial screenshots to focus on the particular feature, whereas appendix C provides screenshots of all sample applications used. To get a sense of how the various user interface elements are placed together, see appendix C.

3.1. Main Features

/F10/ Audio Sources

DJ-Star supports a maximum of four audio sources to be played back at the same time. An audio source can be either a single audio track, called *audio deck*, or a group of audio tracks, known as *sample deck*. Figures 3.1 and 3.2 show the graphical representations for a track deck and a sample deck, the main difference being that the sample deck has four buttons for playback (▷), while the audio deck only has one (◁) .

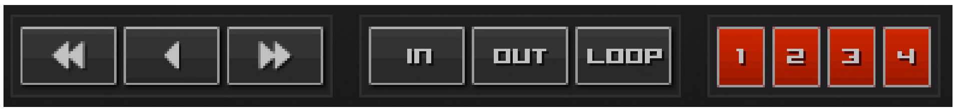


Figure 3.1.: The audio deck controls in *Mixxx* [Mix]

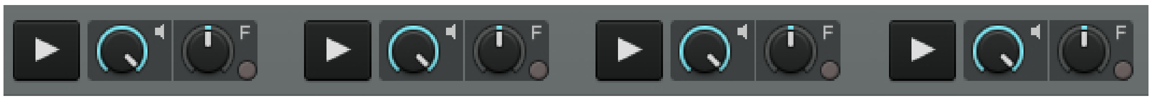


Figure 3.2.: The sample deck controls in *Traktor Pro 2* by *Native Instruments* [Nat]

/F11/ Mixing

The audio sources will be mixed together with individual volumes to form a combined audio signal. The combined audio signal will then be played back on the speakers. The graphical representation for the mixing controls is shown in figure 3.3, where each audio source has a set of controls, arranged a as vertical stripe, indicated by the colored boxes.

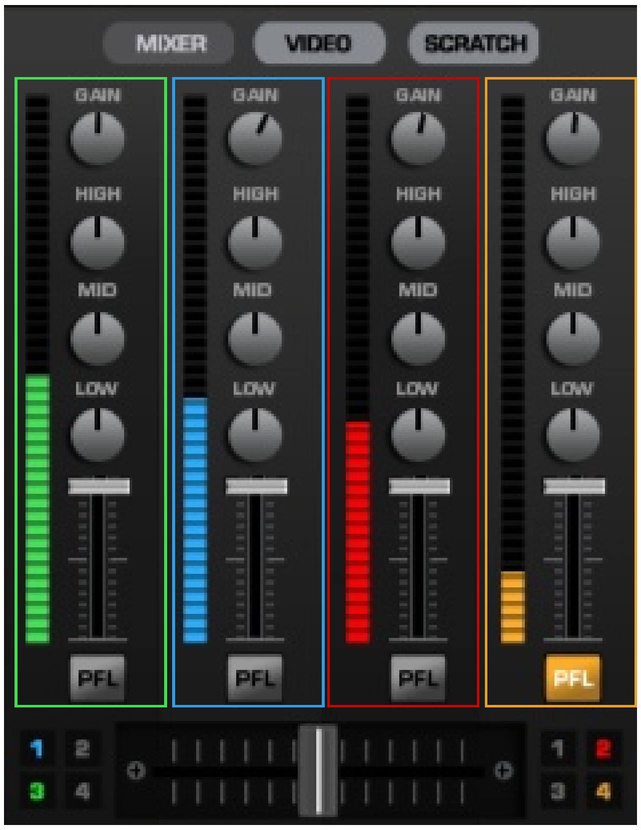


Figure 3.3.: The mixer in *Virtual DJ* by *Atomix Productions* [Ato]

/F12/ Tempo Adjustment

The tempo of an audio track can be changed with the tempo control shown in figure 3.4. The middle position is equivalent to no tempo adjustment, moving the control towards the top increases the tempo, whereas moving it towards the bottom reduces the tempo. This is known as *time stretching*, the technique of “contraction or expansion of the duration of an audio signal” [ZA11, p. 205].

The *time stretching* algorithm in use is able to change the tempo of an audio track without changing its pitch. This was not possible with analog turntables, where a change in tempo was inextricably bound to a change in pitch. An effect usually witnessed when speeding up a vinyl record, where the singer gets a high pitched *”mickey mouse”* voice.

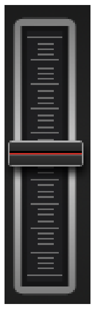


Figure 3.4.: The tempo control in *Mixxx* [Mix]

/F13/ Effects

The audio tracks can be filtered with effects. DJ-Star supports more than 20 effects, the most well known being

- echo,
- reverb,
- high-pass-, and
- low-pass-filter.

Figure 3.5 shows an effects section's graphical representation. Currently, the reverb effect is selected. The leftmost button, *ON*, activates respectively deactivates the effect section. The remainder of controls handles properties that are specific to the effect reverb.

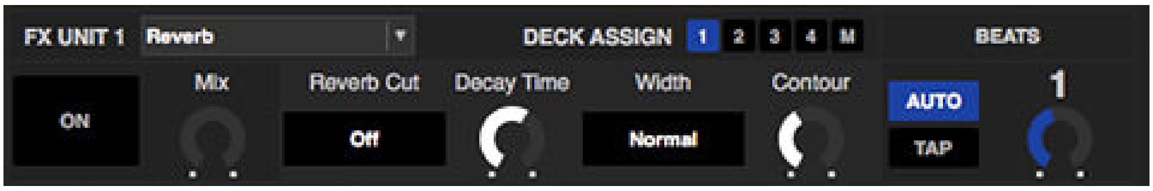


Figure 3.5.: An effect section in *Serato DJ* by *Serato* [Ser]

3.2. Complementary Features

The complementary features offer easier control over the audio manipulation with external control devices, a graphical representation of the audio tracks and the ability to organize the audio sources.

/F14/ Use of External Control Devices

Although DJ-Star is usable with just a regular keyboard and computer mouse, it is much more powerful and convenient to use a dedicated external control device. This external control device is usually connected via USB and communicates with the application in both ways. It can for example send a command to start the playback of an audio deck, but it can also receive commands to trigger a visual feedback, e.g., to signalize whether a track deck is loaded with an audio track or not (this depends highly on the model of the external control device). Figure 3.6 shows an example for an external control device, exhibiting both of the features just mentioned.

/F15/ Visual Audio Representation

DJ-Star preprocesses the audio tracks (/F16/), and generates a visual representation from the results.

This visual representation of a track deck (fig. 3.7) displays the tracks name, current position and duration, BPM[1], as well as two waveforms. The upper waveform visualizes the surrounding sounds at the current playing position indicated by the green line in the middle, looking three seconds in either direction, while the lower waveform visualizes the structure of the whole audio track. So the upper waveform is a magnified version of the lower waveform.

The visual representation of a sample deck (fig. 3.8) is similar, but since audio samples are usually very short compared to audio tracks, only the lower of the two waveforms of the track deck is displayed.

[1]Beats per minute – the tempo of a musical piece

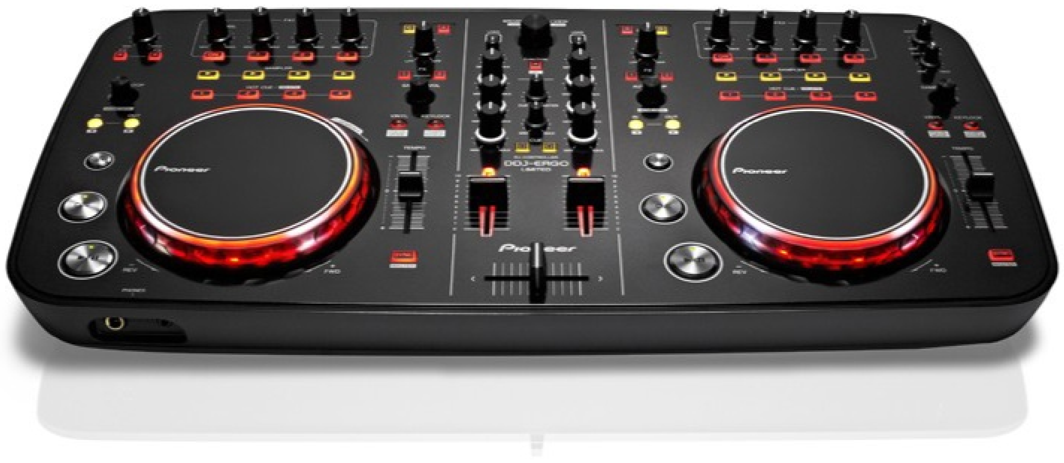


Figure 3.6.: Example for an external control device

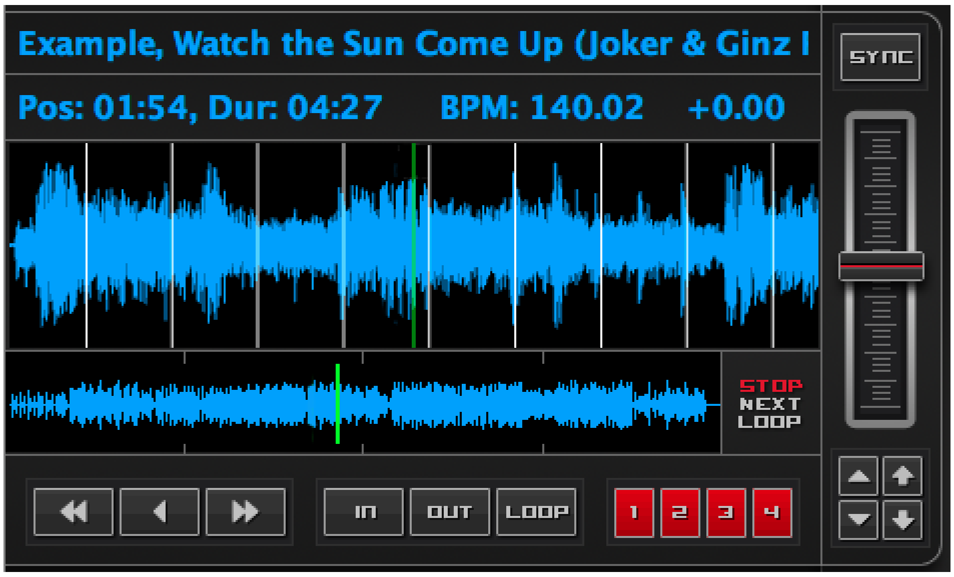


Figure 3.7.: An entire track deck in *Mixxx* [Mix]

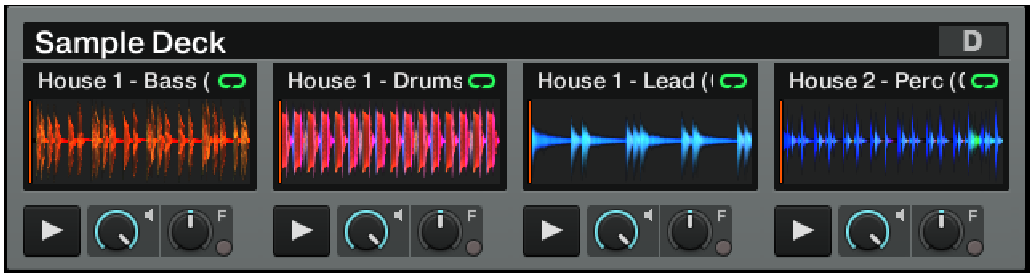


Figure 3.8.: An entire sample deck in *Traktor Pro 2* by *Native Instruments* [Nat]

/F16/ Audio Track Preprocessing

The audio tracks are analyzed to extract the following meta-information:

- Tempo; the speed of an audio track.
- Gain; the loudness of an audio track.
- Key; a group of notes on which a track is built upon.
- Visual audio representation featured in /F15/.

/F17/ Audio Recording

The mixed audio signal can be recorded and saved to the hard disk drive. This recording is a regular audio file playable in any audio application. Figure 3.9 shows the graphical representation of the audio recorder, featuring a red button to start and stop the recording.

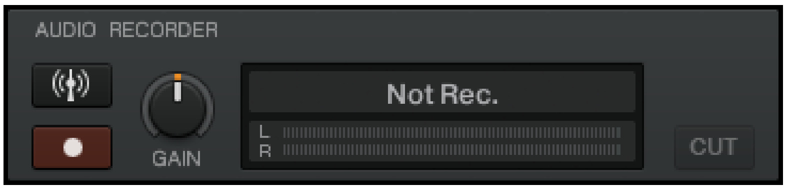


Figure 3.9.: The audio recorder in *Traktor Pro 2* by *Native Instruments* [Nat]

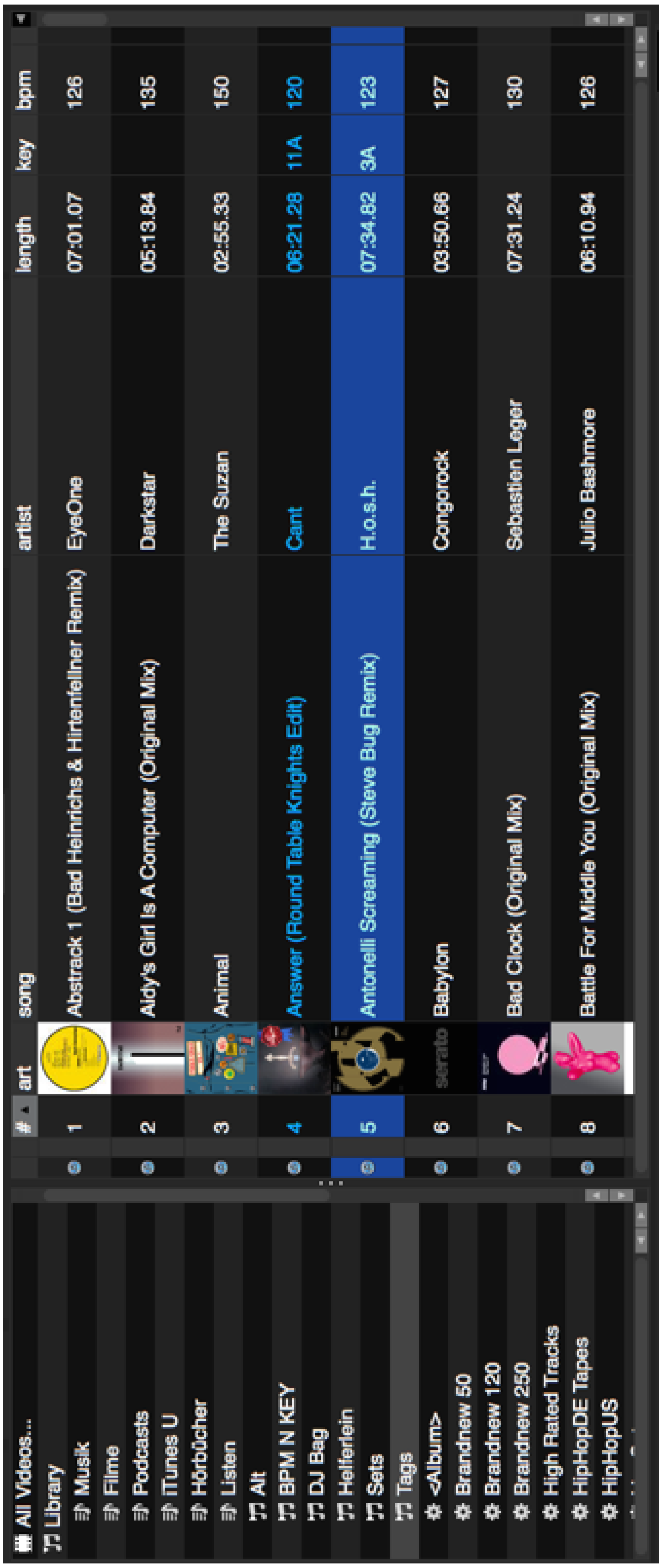
/F18/ Audio Track Organization

In DJ-Star, all audio tracks have to be imported before usage, so all available audio tracks are referenced inside the application. The audio track browser shown in figure 3.10 is split into two sections. The left section shows the user-created playlists, while the right section displays the individual audio tracks in the selected playlist including additional details such as track length, key or BPM. The audio tracks can be loaded from the browser into the individual decks for playback.

3.3. Experiment Setup

For our experiments, we used a typical DJ audio interface, which has multiple inputs and outputs and low latency, as well as two turntables (see fig. 3.11).

DJ-Star is configured to have four sample decks playing at the same time. Each of the decks is routed to all of the four effect sections. The effect sections are configured to use the reverb effect. This is the primary use case to perform music, while maximizing the computing power needed by the application.



All Videos...
Library
Musik
Filme
Podcasts
iTunes U
Hörbücher
Listen
Alt
BPM N KEY
DJ Bag
Helferlein
Sets
Tags
<Album>
Brandnew 50
Brandnew 120
Brandnew 250
High Rated Tracks
HipHopDE Tapes
HipHopUS

#	art	song	artist	length	key	bpm
1		Abstrack 1 (Bad Heinrichs & Hirtenfellner Remix)	EyeOne	07:01.07		126
2		Aldy's Girl Is A Computer (Original Mix)	Darkstar	05:13.84		135
3		Animal	The Suzan	02:55.33		150
4		Answer (Round Table Knights Edit)	Cant	06:21.28	11A	120
5		Antonelli Screaming (Steve Bug Remix)	H.o.s.h.	07:34.82	3A	123
6		Babylon	Congorock	03:50.66		127
7		Bad Clock (Original Mix)	Sebastien Leger	07:31.24		130
8		Battle For Middle You (Original Mix)	Julio Bashmore	06:10.94		126

Figure 3.10.: Audio track organization in *Serato DJ* by *Serato* [Ser]

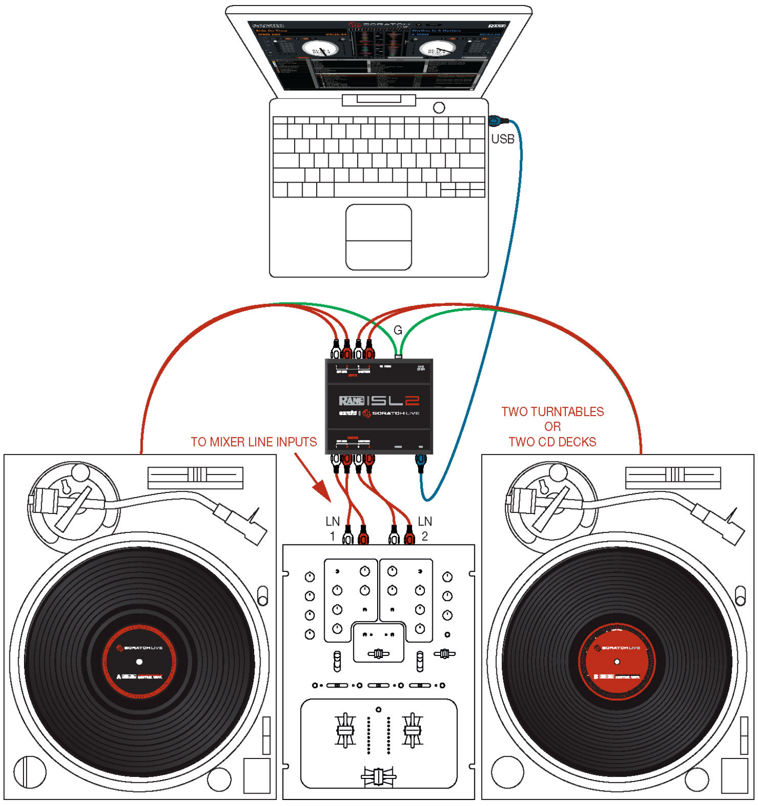


Figure 3.11.: Hardware setup for Serato DJ. [Ran]

4. Reverse Engineering

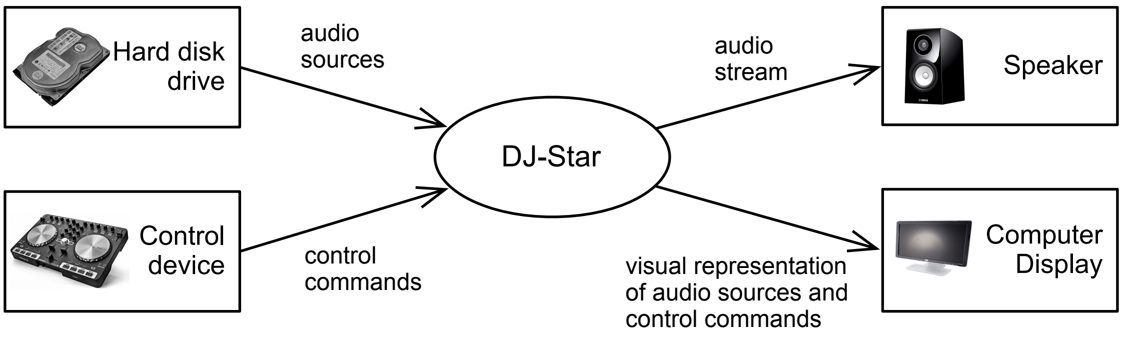


Figure 4.1.: Data flow for DJ-Stars main functions.

To be able to parallelize DJ-Star, first the current state of the application is examined. This chapter presents a comprehensive look at the architecture, the structure and function of the subsystems and their relationships.

Figure 4.1 shows the main data flow of DJ-Star. The audio data is read from the hard disk drive and is manipulated by commands sent from a control device. This control device is usually dedicated hardware with customized controls (like the one in figure 3.2), but can also be a regular computer keyboard and mouse. The manipulated audio data is being streamed to the speakers. In addition, the computer display shows a graphical representation of both the audio input data and the commands that manipulate it.

4.1. Architecture As Seen By The Company

Because the DJ-Star program code evolved historically, there is not a lot of programming documentation available. Figure 4.2 is taken from a presentation about the DJ-Star architecture and visualizes the application's subsystems and their dependencies. Each circle represents one subsystem. The arrows denote dependencies between subsystems. The subsystem in the middle does not have a name but a number of classes and subsystems that it consists of. AppModule is the main module of DJ-Star while AudioEngine, Control System and Settings are all subsystems. The GUI and Browser are tightly coupled, which is denoted with overlapping circles in the diagram.

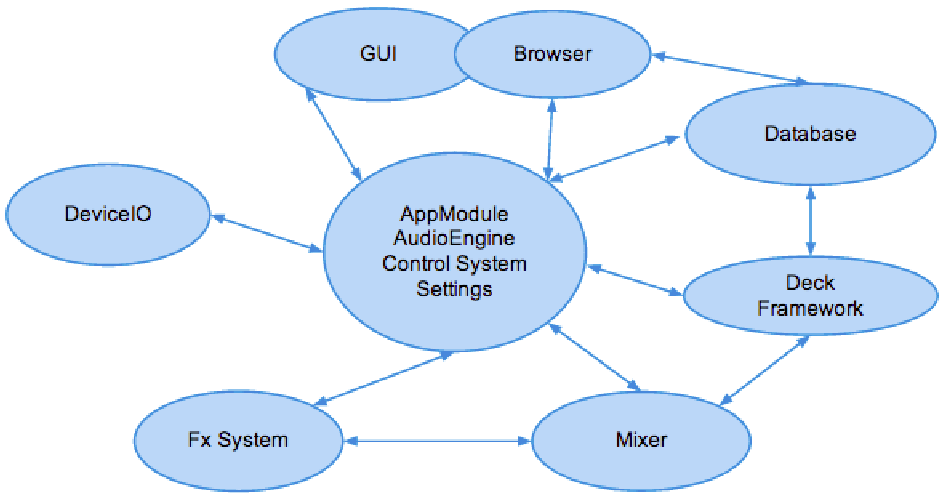


Figure 4.2.: DJ-Star architecture documentation diagram.

Since this is not enough to get a comprehensive understanding of the architecture, we recreated the documentation by reverse engineering the source code. The following sections document the result of this analysis.

4.2. Refined Architecture

The refined architecture diagram in figure 4.3 reveals a comprehensive view on the DJ-Star architecture. It consists of several layers (the layer pattern is based on [Bus98]), all of which are described in detail in the following subsections 4.2.1 – 4.2.4.

The communication between the *User Interface* and *Core* layers is handled through the *Event Middleware* layer. In contrary, the communication inside the *Core* layer, is realized by hardwired calls to the other subsystems. *Application Facade* is the central subsystem that coordinates the interaction of the subsystems.

All subsystems use a facade module for interaction with other subsystems in the application core. This facade encapsulates the control logic and is responsible for keeping the subsystems state.

4.2.1. User Interface

The User Interface layer hosts all subsystems that are responsible for the interaction with the user. Namely these are *Devices Representation*, *GUI* and *Waveform*. *Devices Representation* manages the displaying of the application's state on external control devices. *GUI* is responsible for displaying the application state on the computer display. Lastly, the *Waveform* subsystem maintains graphical representations of the audio sources, which are displayed by the *GUI* subsystem.

4.2.2. Event Middleware

The subsystem *Event Middleware* handles the communication between the *User Interface* and *Core* layer by using *Controls*. Each GUI element has one or more *Controls*. Upon a value change, the *Control* executes its handler function, which implements the functionality.

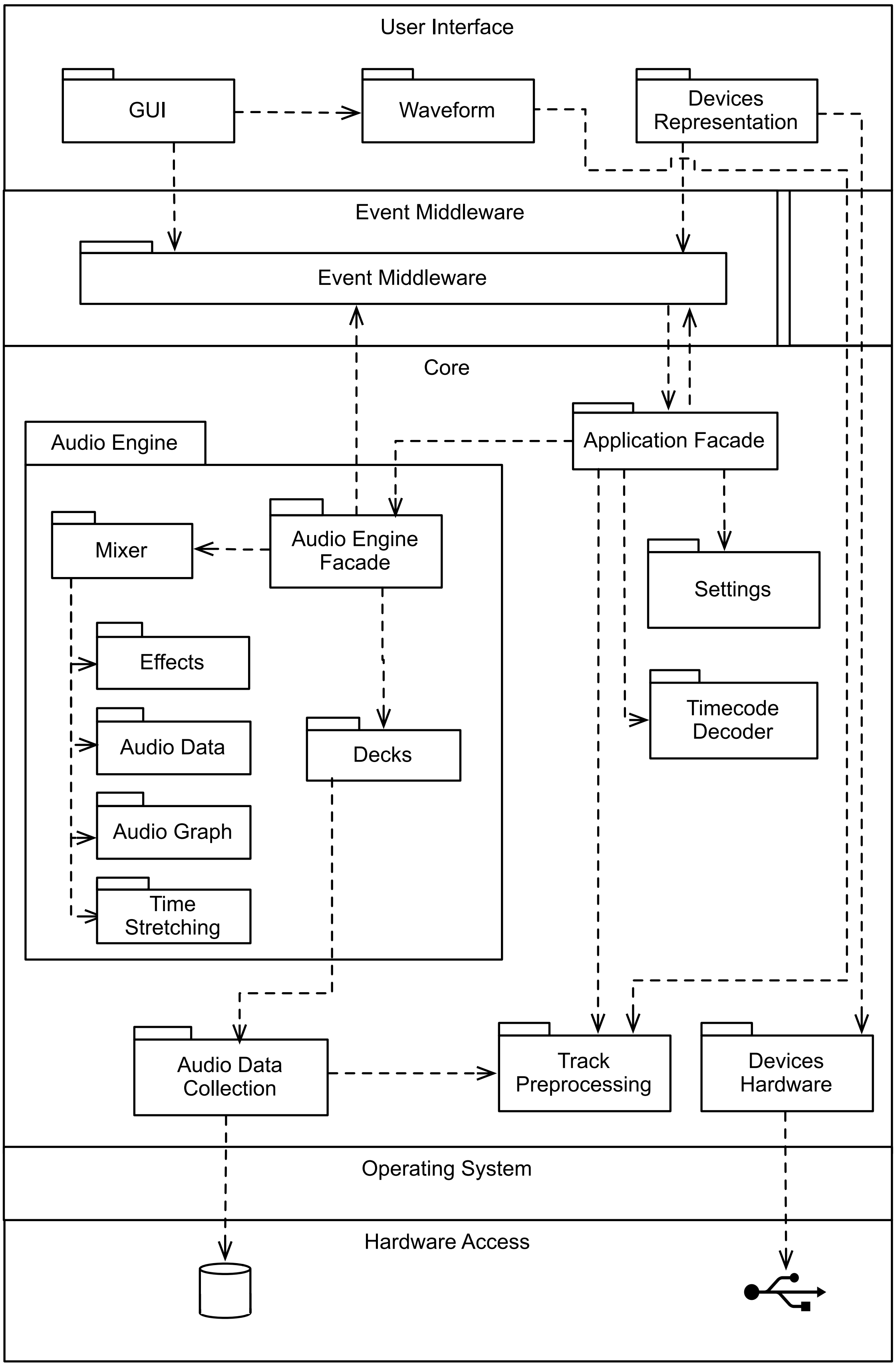
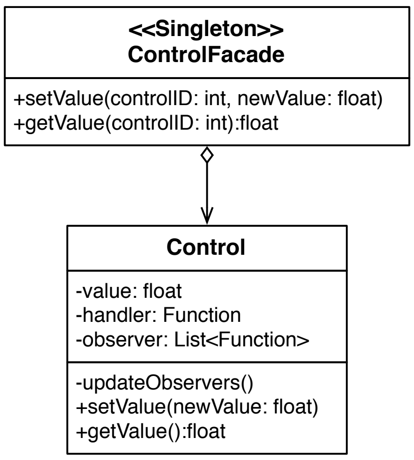


Figure 4.3.: Layers and corresponding subsystems.



<<Singleton>>
ControlFacade
+setValue(controlID: int, newValue: float)
+getValue(controlID: int):float
Control
-value: float
-handler: Function
-observer: List<Function>
-updateObservers()
+setValue(newValue: float)
+getValue():float

Figure 4.4.: Structural view of the Event Middleware.

Additionally, the subsystem realizes the observer pattern. The value of a *Control* is being changed or retrieved by the respective functions in the *ControlFacade* singleton. After the value changed, the observers are notified. Figure 4.4 depicts the *ControlFacade*, *Control* and their dependency.

4.2.3. Core

The *Core* layer hosts DJ-Star's main functionalities including the important *Control Logic* and *audio* subsystems. The following sections 4.3 – 4.10.5 give a detailed explanation of all subsystems in the *Core* layer.

4.2.4. Operating System and Hardware Access

The two lowest layers access hardware functions through the operating system to enable DJ-Star access to the hard disk drive and to establish USB connections. The hard disk drive is used to load audio data into the application and the USB connectivity is used to establish connections to the external control devices.

4.3. Devices Representation and Hardware

The subsystems *Devices Representation* and *Devices Hardware* offer connectivity for external control devices. They directly realize feature /F14/.

Services include managing connections and disconnections, sending and receiving messages in form of MIDI, HID and a company-developed protocol. In addition, multiple control devices can be connected at the same time, where a routing between control device and software function determines the assignment.

4.4. GUI and Waveform

To display the program state on a computer display, DJ-Star features the subsystems *GUI* and *Waveform*. Together, they realize feature /F15/, whereby *Waveform* serves *GUI* for generating and maintaining graphical representations of audio data.

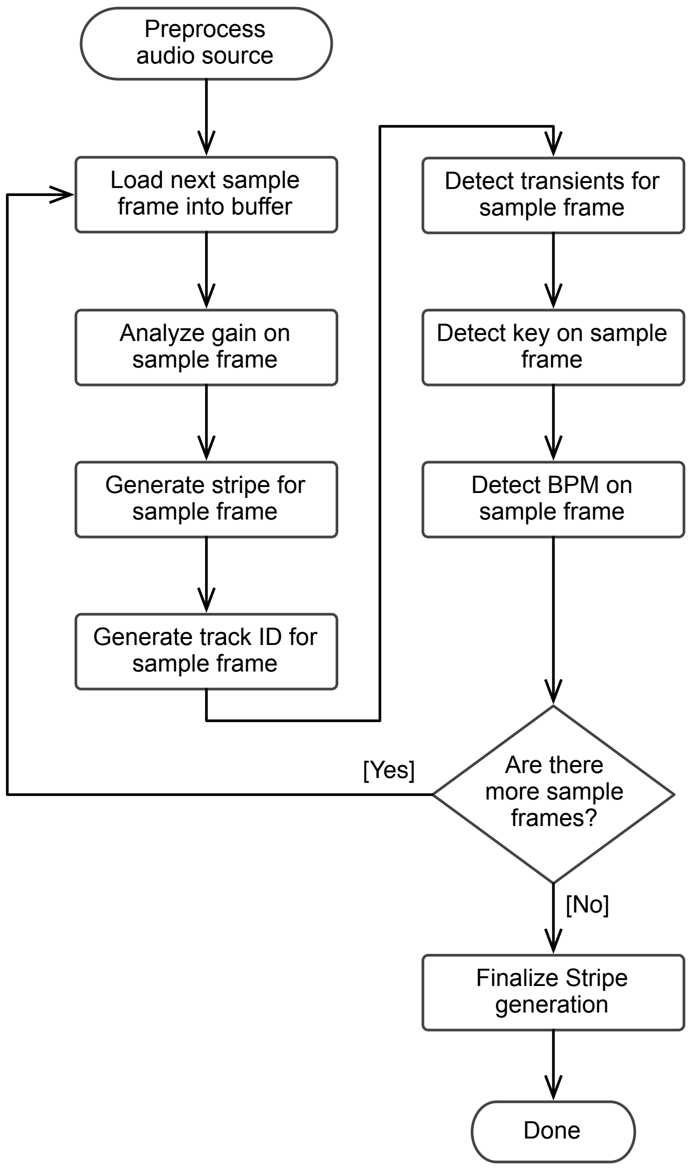

Preprocess audio source
Load next sample frame into buffer
Analyze gain on sample frame
Generate stripe for sample frame
Generate track ID for sample frame
Detect transients for sample frame
Detect key on sample frame
Detect BPM on sample frame
Are there more sample frames?
[Yes]
[No]
Finalize Stripe generation
Done

Figure 4.5.: Preprocessing of audio data.

4.5. Track Preprocessing

The subsystem *Track Preprocessing* (realizing feature /F16/) offers support for the preprocessing of audio data to extract metadata such as the speed and structure of an audio track. This functionality is used in *Audio Data Collection* to provide the user with audio metadata such as the musical key[1]. In addition, the subsystem *Waveform* uses *Track Preprocessing* to create the metadata for the visual audio representation.

Figure 4.5 shows the procedure for one audio track. The audio track is analyzed in chunks of frames, computing a variety of metadata on each chunk. These metadata is gain analysis[2], stripe generation, generation of a track identifier, detection of transients[3], key and BPM [4] detection. After the preprocessing of all sample frames is completed, the stripe is finalized, thus completing the preprocessing of the audio track.

4.6. Audio Data Collection

In DJ-Star, all audio tracks have to be imported into the *Audio Data Collection* and referenced inside the application before they can be used. The user is then able to organize the audio tracks into groups and to annotate them. This is essentially a library aiding the user in finding the right audio track effortlessly.

The *Audio Data Collection* stores a track's metadata composed of the title, artist, file path, musical genre and comments (much like other applications that organize audio tracks) and the musical key and BPM generated in the track preprocessing. All this information is managed and stored by the subsystem *Audio Data Collection*, realizing feature /F18/.

4.7. Timecode Decoder

The subsystem *Timecode Decoder* realizes feature/F14/ in section 3.2, enabling the use of special vinyl records to control DJ-Star's audio playback. The vinyl records encode the current speed, playback direction and needle position in an audio signal. This audio signal is sent through a low latency audio interface into the computer, where it is being processed by DJ-Star. After determining the speed, playback direction and needle position, DJ-Star plays back the corresponding part of the audio track loaded.

4.8. Settings

The subsystem *Settings* provides access to the global application settings.

4.9. Application Facade

The subsystem *Application Facade* is responsible for coordinating the interaction of all subsystems in the *Core* layer.

Figure 4.6 shows the procedure for the audio processing cycle (APC), which is central to DJ-Star. The audio interface requests new audio data from the *Application Facade* by sending signals at a fixed rate. One APC consists of updating the state of the timecode decoder (if enabled) and requesting a new frame of audio data from the *Audio Engine*, described in the next section 4.10.

[1]The musical key is a group of notes on which a musical piece is built upon.
[2]Gain is a metric for the loudness of an audio track.
[3]A transient is a high volume, short duration peak in an audio track.
[4]BPM (beats per minute) determines the speed of an audio track.

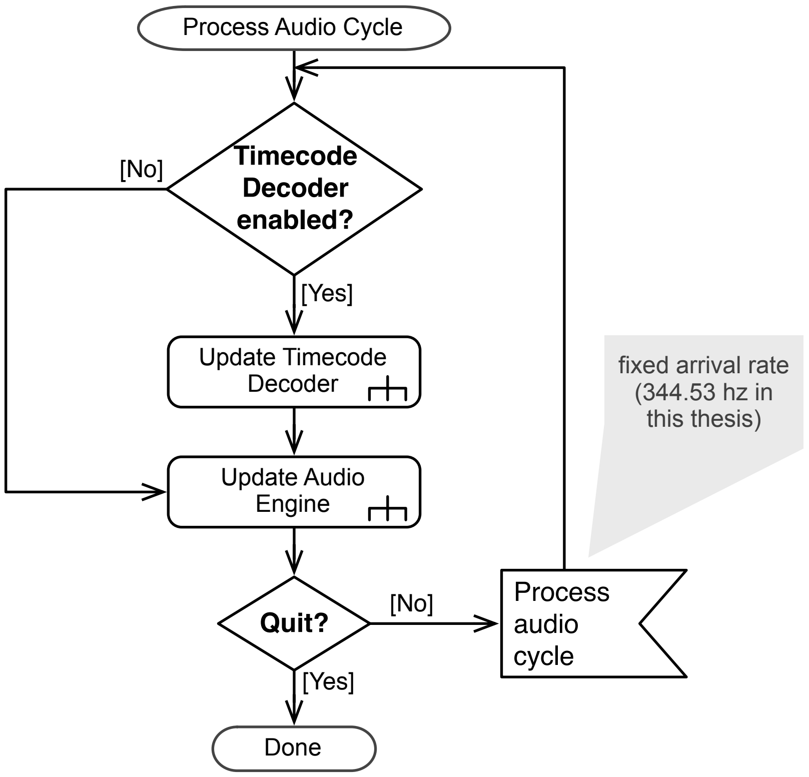


Figure 4.6.: One audio processing cycle (APC).

4.10. Audio Engine

The subsystem *Audio Engine* is central to the application. It provides DJ-Star with all the main features /F10/ – /F13/ by handling the audio data processing and processing of low-latency audio-related controls. The cyclic audio data processing is triggered by a signal, which arrives at a rate of 344.53 hz (see the next section 4.10.1 for details).

The *Audio Engine* consists of different subsystem components. The *Mixer* uses the *Audio Graph* to build a graph representation of the audio manipulations in DJ-Star, including *Time Stretching* and *Effects*.

4.10.1. Time Criticality

The work of the *Audio Engine* subsystem is time-critical. It delivers audio data in small packets to the audio interface to achieve low latency and a fast audio response upon user input. If it misses to respond in a given time, the audio will be distorted.

According to

$$PL * C = SR, \tag{4.1}$$

the number of packets C the audio engine has to compute depends on the packet length PL and the sample rate SR. So the audio engine has to process C packets per second, limiting the run-time T of one APC to

$$T(APC) < \frac{1s}{C}. \tag{4.2}$$

Combining the equations 4.1 and 4.2 defines the real-time constraint:

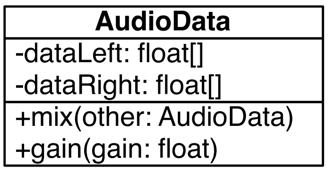


Figure 4.7.: Audio data format.

Inequation 4.1 (Real-time Constraint (RTC))

$$T(APC) < \frac{PL}{SR}$$

In this thesis, $PL = 128$ samples and $SR = 44100$ samples per second is used. According to definition 4.1, the maximum run-time of one APC is then 2.9 ms, and the arrival rate of the APC signal is 344.53 hz. Though this works in practice, the RTC is not proven to be satisfied with the current implementation.

4.10.2. Mixer

The *Mixer* is a vital part of the *Audio Engine*, realizing feature /F11/. It is responsible for mixing and manipulations of the audio tracks by using a graph representation, explained in detail in section 4.11. All audio related computations happen in this subsystem component.

4.10.3. Decks

The subsystem component *Decks* encapsulates the behavior and the audio processing characteristics of the deck types *track deck* and *sample deck*. Therefore, it is responsible for realizing feature/F10/.

4.10.4. Effects

The subsystem *Effects* offers audio effects for the audio tracks, e.g., reverb and echo. It realizes feature /F13/.

4.10.5. Audio Data

The *Audio Data* subsystem features the audio data type used in the whole application. An *Audio Data* object is characterized by two float arrays (figure 4.7), each characterizing one of the two stereo channels. The length of the audio data is a preference, chosen by the user, and can vary between 64–2304 samples. It allows mixing with another *Audio Data* object and setting the gain, which results in setting its volume.

4.10.6. Time Stretching

The subsystem component *Time Stretching* offers support for adjusting the tempo of the audio tracks, therefore realizing feature/F12/.

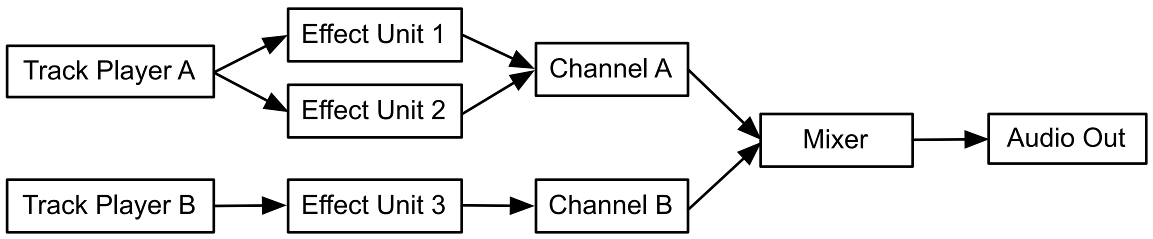


Figure 4.8.: Example audio graph.

4.11. Audio Graph

The subsystem *Audio Graph* includes an directed acyclic graph (DAG) for the representation of the audio flow. The vertices in the DAG represent tasks where the audio data is transformed, whereas edges represent the audio data flow. A simple audio graph is depicted in figure 4.8, where for example *Track Player B* is a task and its edge to *Effect Unit 3* indicates that the audio data is processed by *Effect Unit 3* afterwards. *Track Player B* is followed by two separate effect units *Effect Unit 1* and *Effect Unit 2*, so the audio is split first, processed by each effect unit and finally mixed together upon reaching *Channel A*.

Audio Graph Vertices

Figure 4.9 depicts the structure of the `AudioGraph` consisting of vertices. Each vertex is represented by one module.

Concrete Vertices are of type `BufferedVertex` or `WrappedVertex`. The `BufferedVertex` manages the buffer by itself, whereas with the `WrappedVertex` the buffer is set externally. Each vertex can have any number of effects and has the following functions:

`preprocess`, `execute`, `postprocess` Processing of the vertex, explained in detail in section 4.12.

`enableRendering`, `finishRendering`, `waitForInputsDone` For parallel audio graph execution, explained in detail in section 4.13.

Audio Graph Edges

Each abstract vertex has any number of input edges (composite pattern), characterized by the tuple (`sourceVertex`, `weightFunc`). `weightFunc` is a function returning a float $\in [0, 1]$, expressing the volume at which the `sourceVertex` audio should be mixed in.

The edges represent the data flow between vertices, and therefore mark the dependencies between them.

4.12. Audio Engine Update

Updating the audio engine works in three phases: preprocess, execute and postprocess (fig 4.10). Only the execution phase is the same for all vertices, in preprocess and postprocess, some vertices have special functions, which are discussed in the following sections.

For the processing of the audio engine, the vertices of the audio graph are put in a list and sorted by depth. The sorted order for the example audio graph in figure 4.8 is: TrackPlayer A, TrackPlayer B, Effect Unit 1, Effect Unit 2, Effect Unit 3, Channel A, Channel B, Mixer, Audio Out.

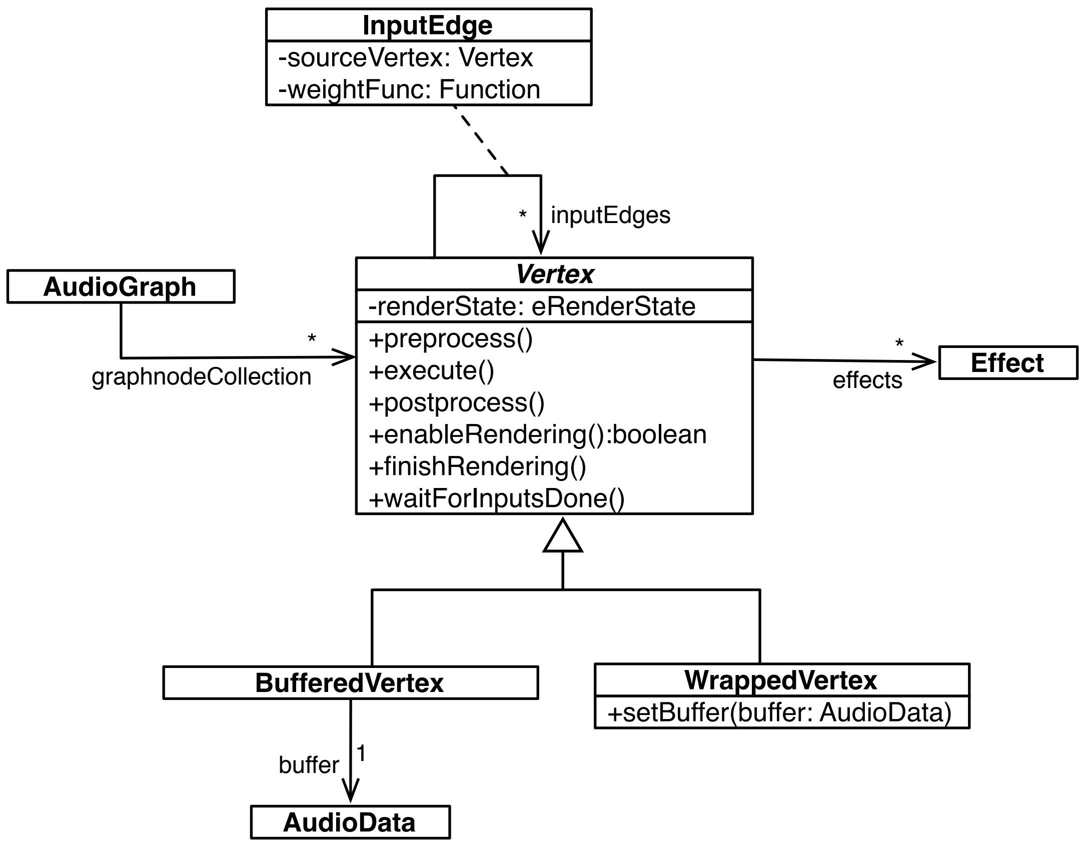


Figure 4.9.: Audio graph structure.

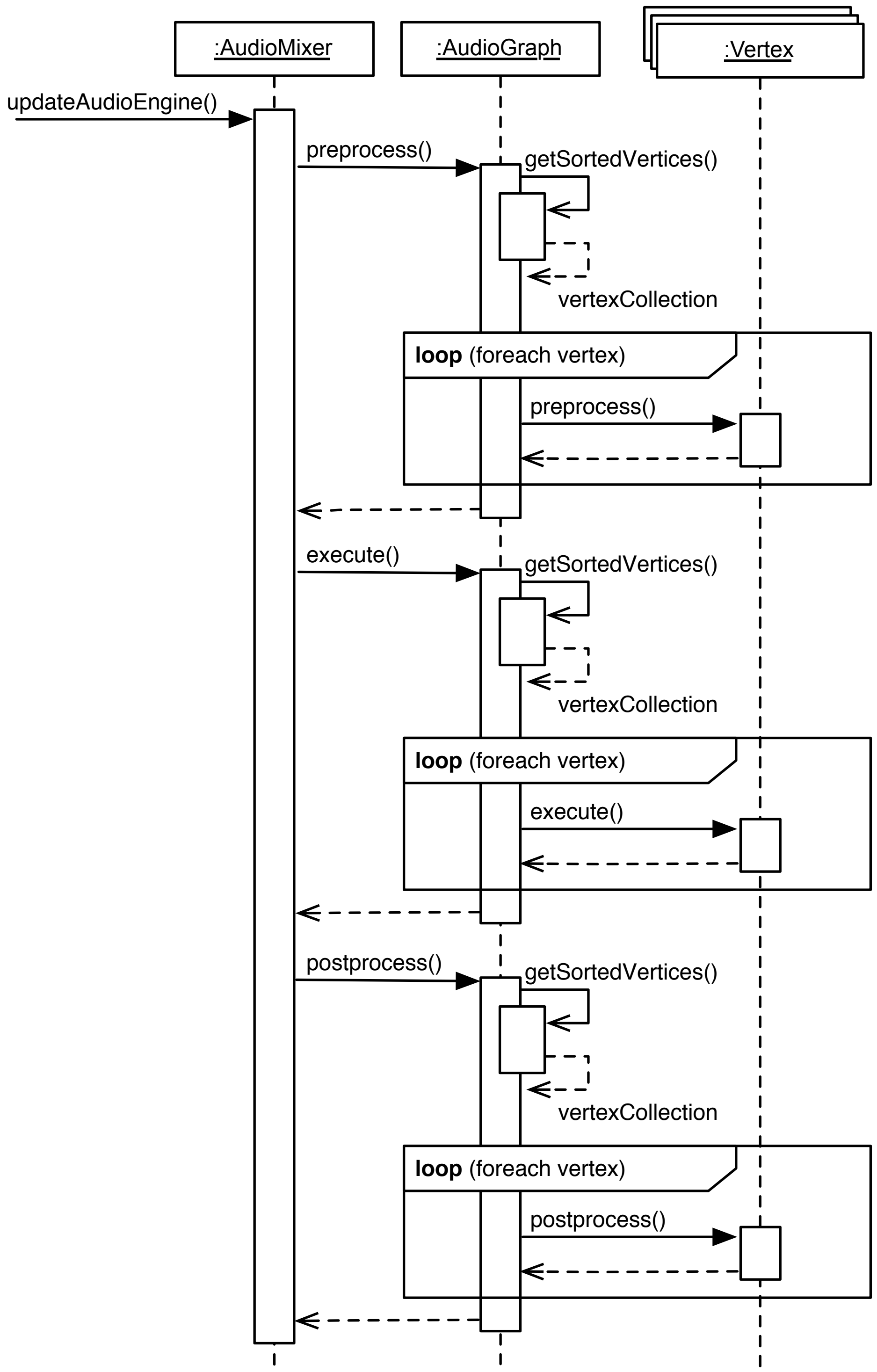


Figure 4.10.: Audio engine processing in one APC.

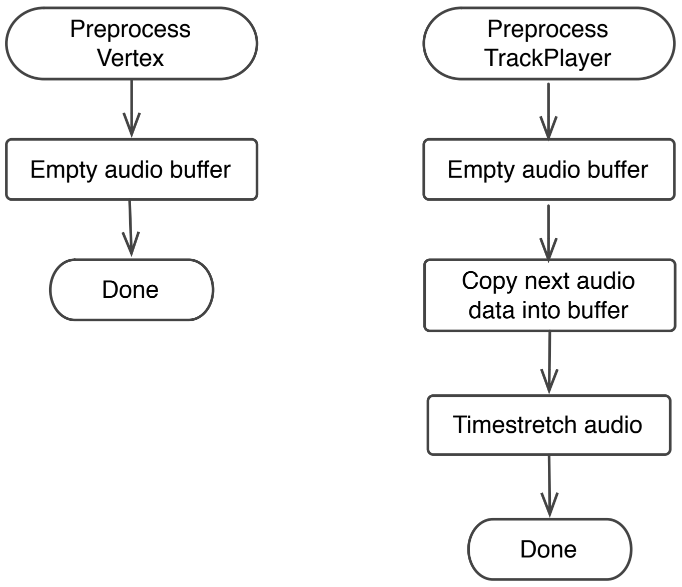


Figure 4.11.: Preprocessing of an audio graph vertex and the TrackPlayer vertex.

Preprocess

The phase preprocess is used to prepare the vertices for execution. All vertices empty their audio buffer and reset their render state. The TrackPlayer vertex additionally copies audio data into its buffer, time stretches the audio and sets the phase as depicted in figure 4.11.

Execute

The phase execute is the actual *audio graph execution*. The sorting ensures that all dependent vertices are already executed upon starting the execution of a vertex.

The execution (fig 4.12) is the same for all vertices and is composed of *mixing the input connections* and *applying the effects* associated with a vertex.

Figure 4.13 shows the steps for mixing the input connections. For each input connection, the weight function is applied to the connections audio data and the audio data is subsequently mixed into the vertex audio buffer.

Applying the effects is shown in figure 4.14, where the effects are applied one after another to the vertex audio buffer.

Postprocess

In postprocessing, only TrackPlayer and AudioRecorder vertices implement any logic (fig. 4.15), while all other vertices don't. The TrackPlayer vertex updates the track position and loads audio data for hot cues[5]. An AudioRecorder vertex copies the content of the audio buffer into an audio save buffer, where it will be written to the hard disk drive later.

4.13. Parallel Audio Engine Update

The audio engine update already supports parallel processing. Multiple vertices can be processed task-parallel if all of their input connections are processed. The number of

[5]A hot cue is a saved position in an audio track.

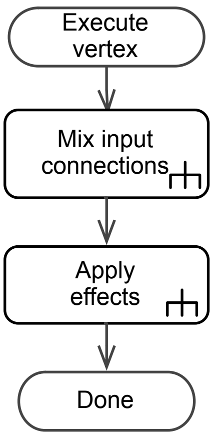


Figure 4.12.: Execution of one vertex.

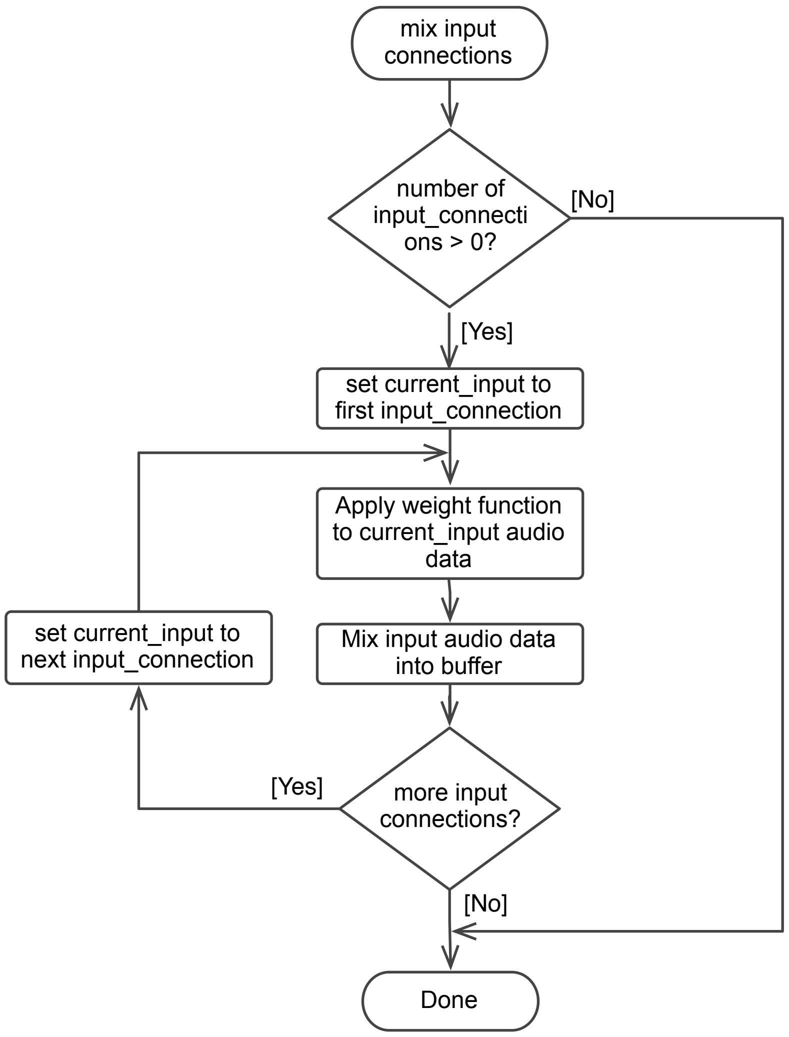

mix input connections
number of input_connections > 0?
[No]
[Yes]
set current_input to first input_connection
Apply weight function to current_input audio data
Mix input audio data into buffer
set current_input to next input_connection
[Yes]
more input connections?
[No]
Done

Figure 4.13.: Mixing input connections of one vertex

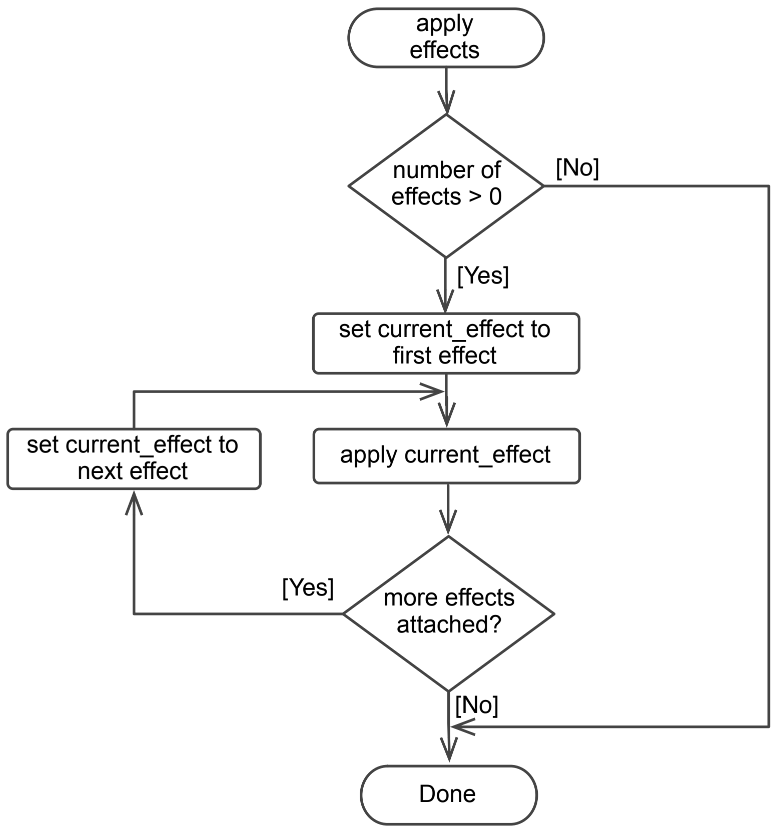


Figure 4.14.: Applying effects of one vertex.

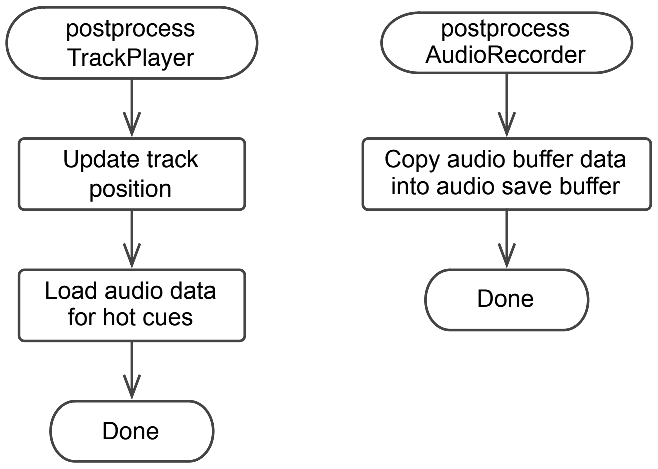

postprocess TrackPlayer
Update track position
Load audio data for hot cues
Done
postprocess AudioRecorder
Copy audio buffer data into audio save buffer
Done

Figure 4.15.: Postprocessing of a TrackPlayer and AudioRecorder vertex.

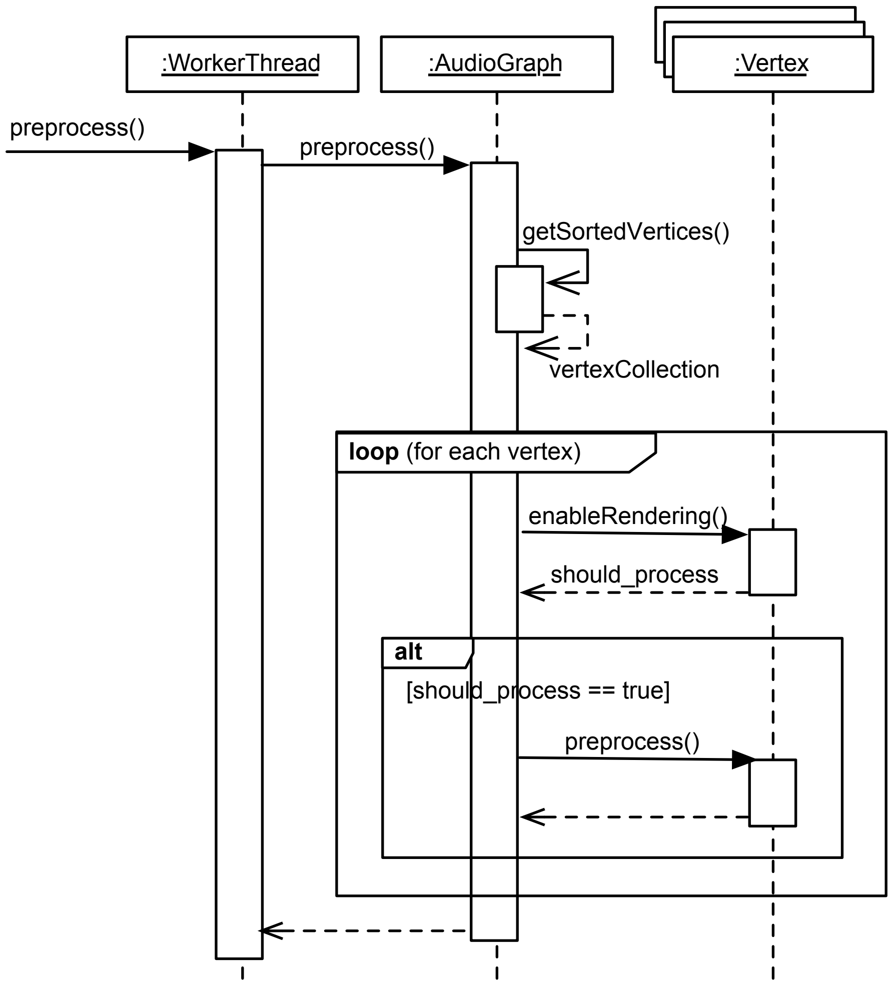


Figure 4.16.: Multithreaded graph preprocessing.

threads t for the concurrent processing is limited to $t = min(p - 1, 8)$ where p is the amount of processors of the machine.

Parallel processing uses the master/worker pattern (see section 2.1.3), where the work items are in a queue (the vertexCollection) and the thread handling the audio engine update becomes the master thread. The master thread signals the worker threads to start processing as shown in figure 4.18 for the graph execution. The preprocessing of the graph is identical, the only difference being that it runs the `preprocess()` function. After signaling all the worker threads, the master thread itself becomes a worker thread.

All worker threads `preprocess` (fig. 4.16) or `execute` (fig. 4.17) all vertices, respectively, in the `vertexCollection`. Upon reaching the end of `vertexCollection`, the master thread waits until all vertices are processed, which acts as a barrier.

Preprocess

The preprocess phase is identical to the serial audio engine update, except that multiple threads process the vertices. Because the vertices do not have any dependencies on each other, the worker threads can process all the vertices they can get, as shown in figure 4.16.

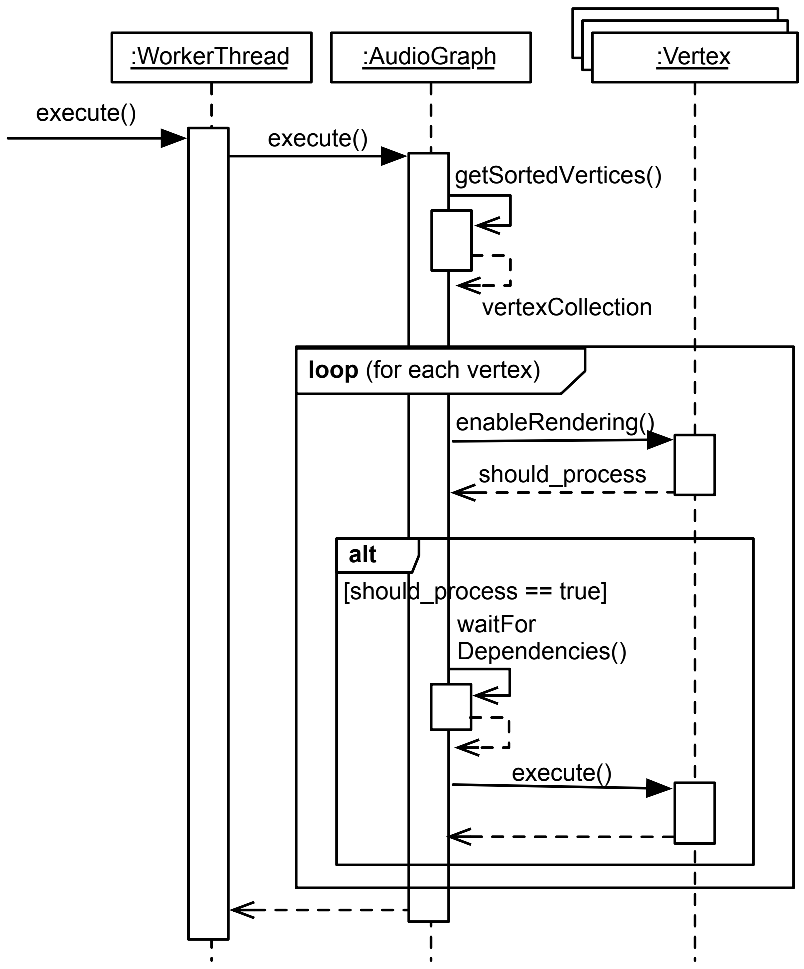


Figure 4.17.: Multithreaded graph execution.

Execute

The execute phase in the audio engine update works similar to the preprocess phase, except that since the graph is executed, the dependencies in the audio graph have to be considered. This is done by `waitForDependencies()`, which is explained in section 4.14 since it is a synchronization mechanism.

Postprocess

The postprocess phase is done exactly as in the serial audio engine update, there is no concurrency exploited here.

4.14. Synchronization In Parallel Audio Engine Update

DJ-Star uses two mechanisms to prevent data races from happening in parallel audio engine update. The following sections will illustrate them.

enableRendering()

Access to the `vertexCollection` is shared among multiple threads. Therefore, access must be regulated to prevent data races.

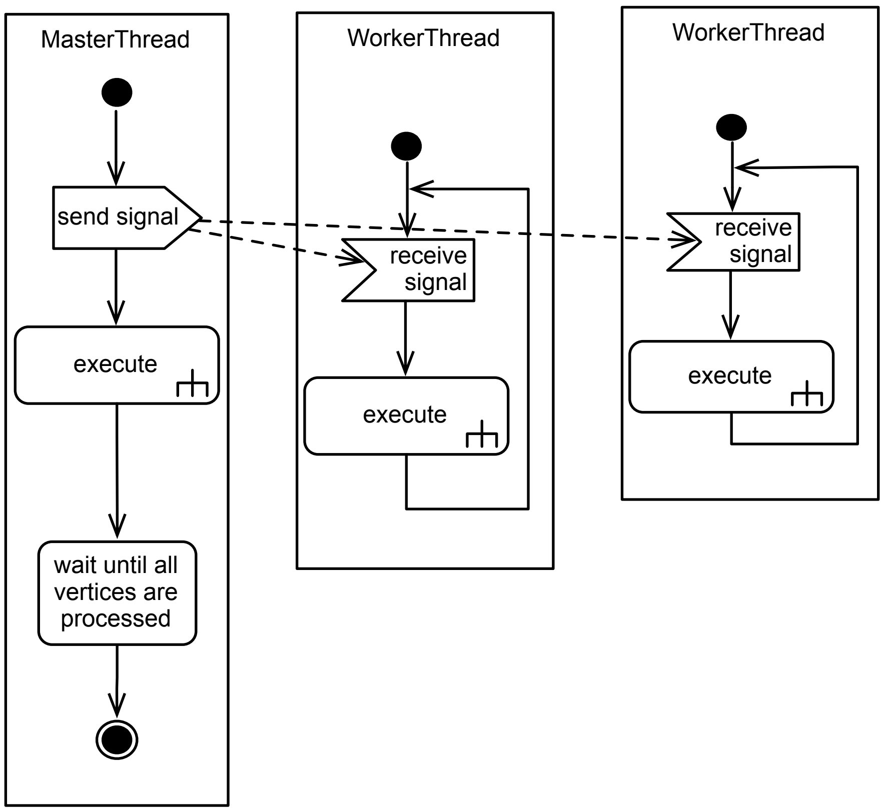


Figure 4.18.: Communication in parallel graph execution with two worker threads.

Before any thread starts preprocessing/executing a vertex, it must check with `enableRendering()` to identify whether the vertex needs processing (fig 4.16/4.17). The function uses an atomic variable to keep its status. The first thread calling `enableRendering()` on a vertex will be given permission to process this vertex by returning `true`. All subsequent threads in the same audio engine update will not get permission to process the vertex, therefore receiving `false`.

waitForDependencies()

When updating the audio engine with one thread only, the sorted `vertexCollection` prevents any violation of the data dependencies between the vertices. However, in the multithreaded audio graph execution, race conditions can arise when using the same strategy. Therefore, a synchronization mechanism must be implemented to ensure the correct processing order.

In the multithreaded graph execution shown in figure 4.17 the worker thread has to call `waitForDependencies()` before executing a vertex. `waitForDependencies()` waits actively until all data dependencies have finished their execution.

Figure 4.19 shows the situation with four worker threads $T1 - T4$. $T1$ just finished executing *TrackPlayer A* and moved on to the next unprocessed vertex, which is *Effect Unit 3*. This vertex, however, depends on *TrackPlayer B* which is not executed yet. In this situation, $T1$ waits actively until $T2$ finishes executing *TrackPlayer B*.

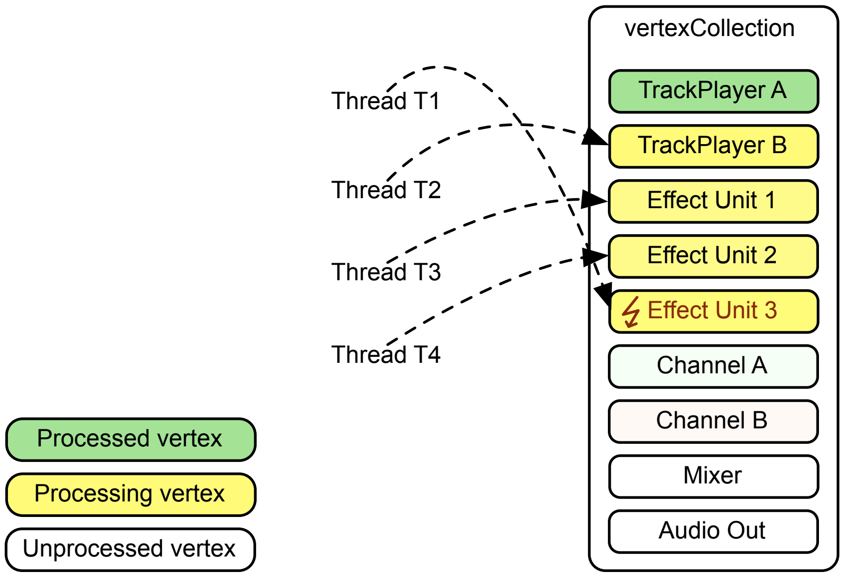


Figure 4.19.: Example for active waiting in the execution of the audio graph from figure 4.8.

5. Performance Analysis

In order to gain a first insight on where DJ-Star's computing power is consumed, the profiler of Microsoft's Visual Studio 2010 Ultimate is used with its *sampling* profiling method. Its documentation [Mic] states that the sampling profiling method is useful for "initial explorations of the performance of your application". It "interrupts the computer processor at set intervals and collects the function call stack" and reports the count of the executing functions by having "little effect on the execution of the application methods." These function counts are used to show the *absolute* rate (indicated by ⌛) of computing power for the respective program part.

Since the resolution of this profiling method is not good enough for the second part of this chapter, where the performance of the individual audio graph vertices is examined, a customized profiler is introduced in section 5.5 and used subsequently.

5.1. Audio Processing Cycle

The sampling profiling reveals that 88% of the computing power in DJ-Star is spent inside the audio processing cycle (the remaining 12% are spent for drawing the waveforms and updating the event middleware). Figure 5.1 shows the flow diagram for the APC annotated with the profiling results. *Update Timecode Decoder* accounts for 16% and *Update Audio Engine* accounts for 72% of the computing power. The following section provides a more thorough examination of *Update Audio Engine*.

5.2. Update Audio Engine

Updating the audio engine accounts for 72% of DJ-Stars computing power. This is divided by the audio graph's methods *preprocess()*, *execute()* and *postprocess()*, whereby *postprocess()* with under 1% is negligible. The main computing power in the runtime of the audio engine update is shared by *preprocess()*, with 33%, and *execute()*, with 38%.

$$T(UpdateAudioEngine) = T(preprocess) + T(execute) + T(postprocess) \tag{5.1}$$

Figure 5.3 shows the preprocessing of TrackPlayer vertices, in which the time stretching of the audio data accounts for almost all of the computing power by having 32%.

When executing the vertices, the majority of the computing power is consumed by *Apply Effects* with 32% (fig. 5.4). *Mix input connections* is almost effortless with less than 1% computing power.

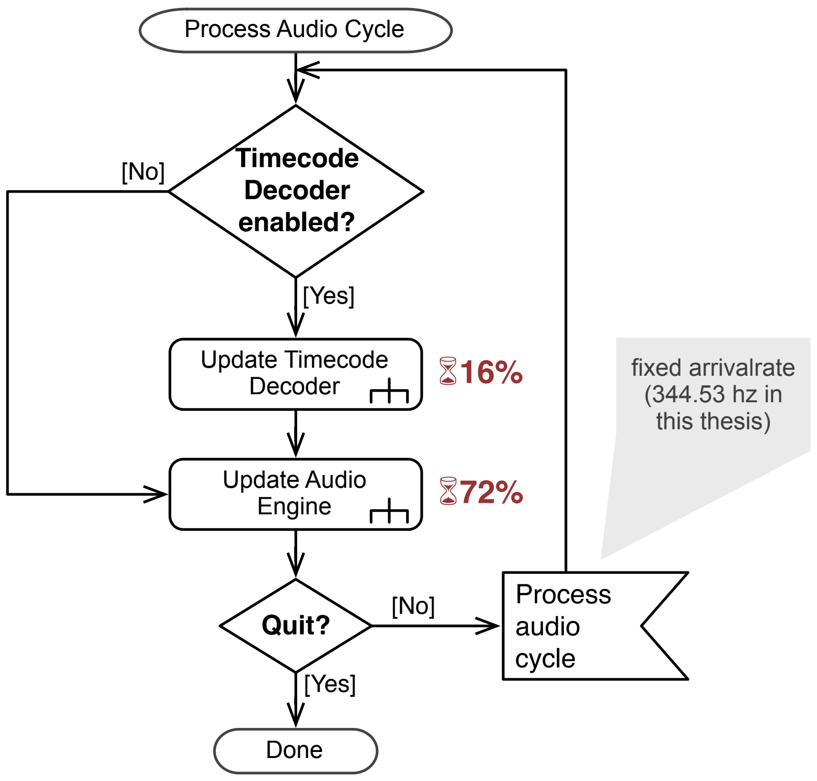


Figure 5.1.: Computing power for one APC.

5.3. Active Waiting In Parallel Graph Execution

The active waiting (fig 5.5), happening before the execution of a vertex, is responsible for 13% of DJ-Star's computing power, when executing the graph with four threads on an eight-core machine (introduced in chapter 7).

This is a waste of computing power. On one hand, threads waiting for their dependencies burn processor cycles without making any progress, on the other hand these threads could possibly execute another vertex while waiting, offering potential for improvement.

Potential for Improvement 5.1 (Active Waiting)
Threads should sleep while waiting to not waste processor cycles. When they can start with their computation, they should be woken up.

Potential for Improvement 5.2 (Idle Threads when Work Is Ready)
Threads should not wait for vertices that are not yet executable, but look for a different vertex that is executable already.

5.4. Actual Audio Graph Configuration

All the computing power in the above sections, such as *Apply Effects* (section 5.2), is used by a combination of vertices. For the further analysis, it is necessary to switch views and look at the computing requirements of the individual vertices. Therefore, another profiling method, discussed in section 5.5, was required.

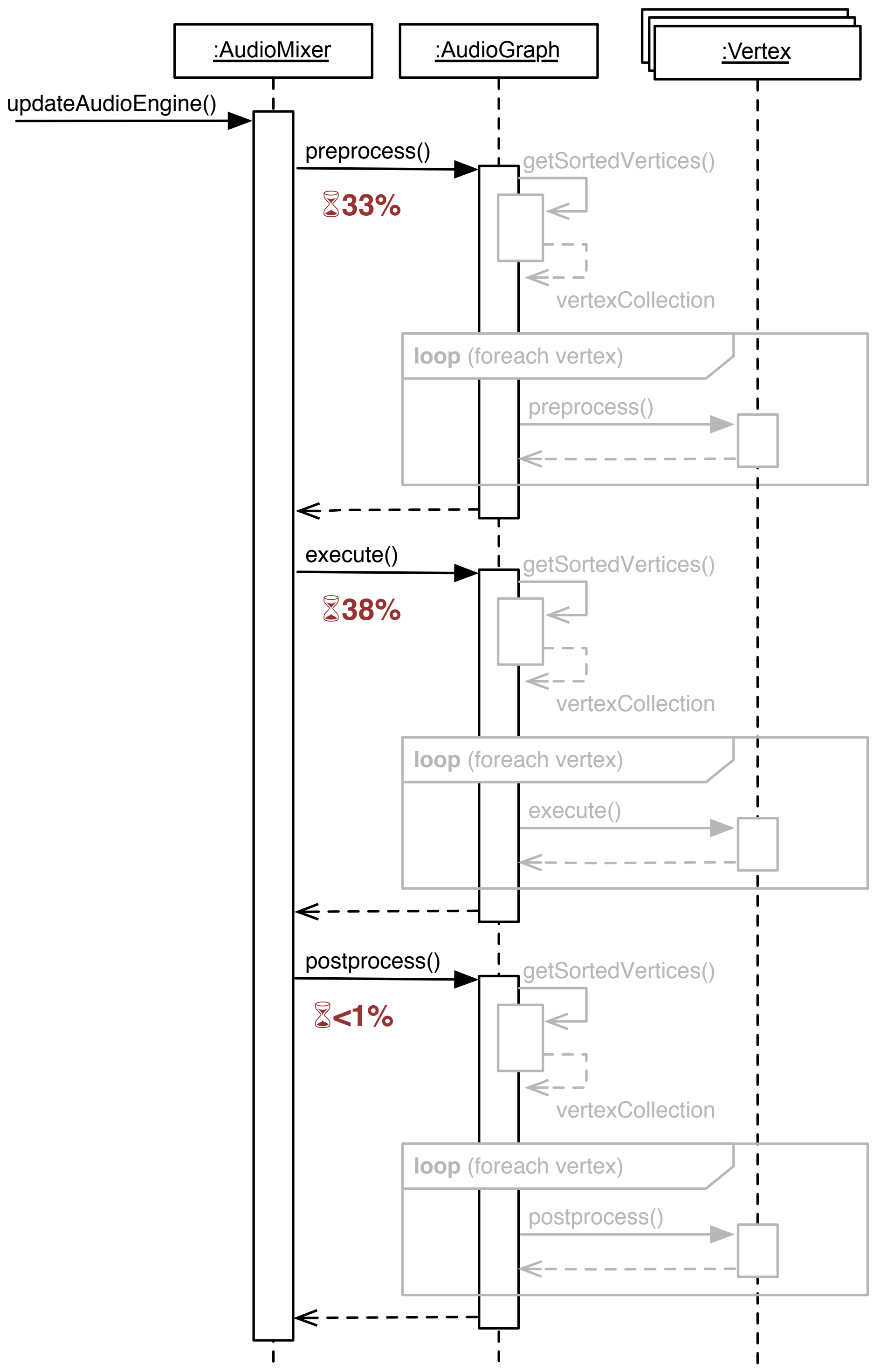


Figure 5.2.: Computing power for the serial audio engine update.

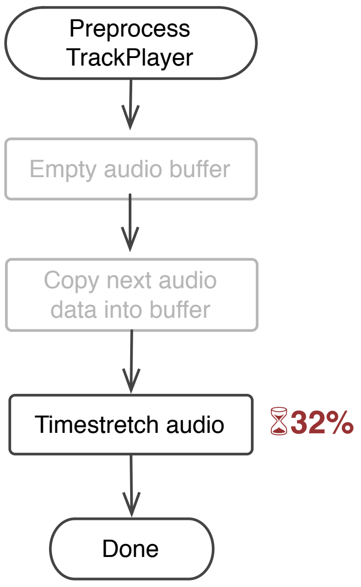


Figure 5.3.: Computing power for the TrackPlayer vertices in the preprocess phase of the audio engine update.

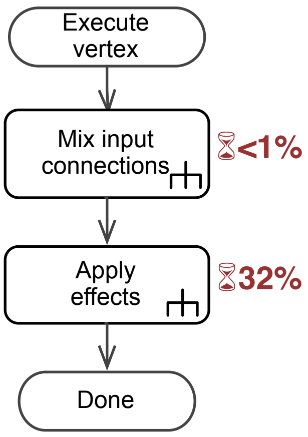


Figure 5.4.: Computation power for the vertices in audio graph execution.

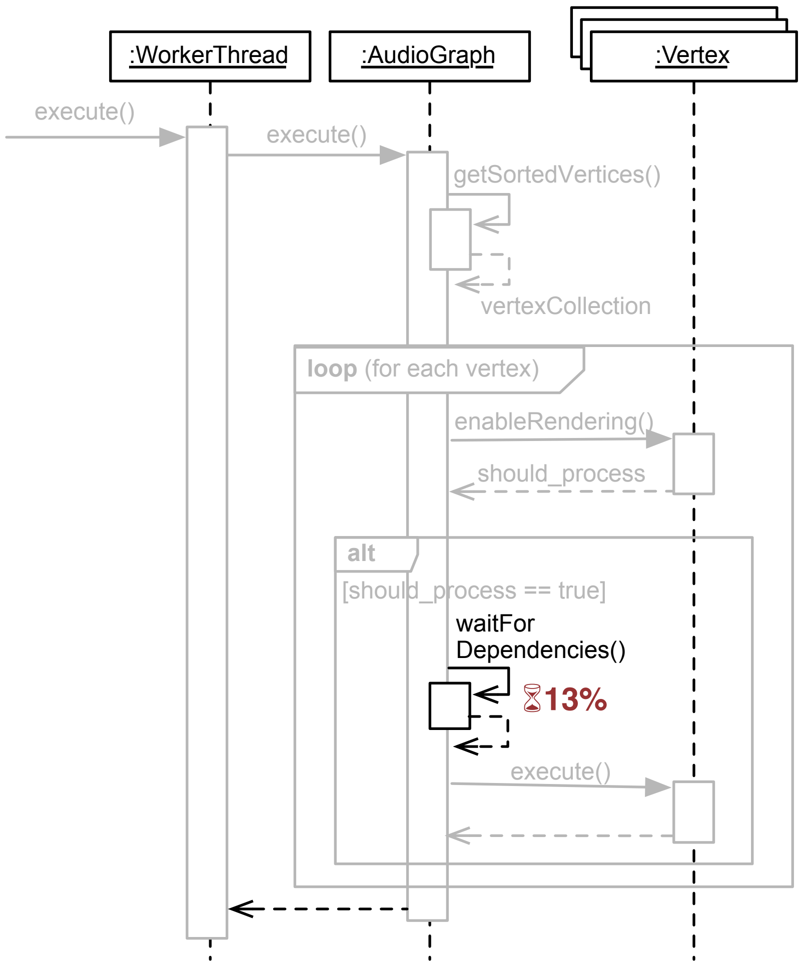


Figure 5.5.: Computation power for the active waiting in parallel audio engine update.

Vertex name	Avg. exec. time ↓
EffectsD	0.2062
EffectsA	0.2042
EffectsB	0.2037
EffectsC	0.2037
ChannelC	0.0229
ChannelB	0.0227
ChannelA	0.0192
RecordBuffer	0.0187
AudioOut1	0.0185
ChannelD	0.0185

Table 5.1.: Average execution time of the ten most time-consuming vertices (msec).

Figure 5.6 shows the actual configuration of the audio graph in our experiment setup, leaving out all vertices that do not change the audio data (identity function) in this setup. Each vertex is denoted as a box, with the name in regular and the effects used in italic font. The names of the leftmost vertices *SPA1* .. *SPD4* are short versions. The written-out version for SPA1 is *SamplePlayer A-1* and so forth.

Easily noticeable is the partitioning of the graph in sections *Deck A–D* and *Master Section*. Also, all vertices except for *Cue Buffer* and *Mixer* process at least one effect.

5.5. Profiling The Audio Graph Execution

The details on the computing power requirements of the individual vertices could not be found in the sampling profiling method introduced in the beginning of this chapter. The application had to be instrumented by hand in order to obtain this information. This profiler, which is used throughout the rest of this chapter, logs the beginning and end of the execution of each vertex.

5.6. Vertex Execution Times In Experiment Setup

For the profiling of the execution times of the vertices, a recording with 10K audio graph executions was made. The following two sections compare the execution time of the vertices by using averages and histograms as metrics.

Average Execution Times

The data (table 5.1) shows that the *Effect* vertices are the most computing-intensive vertices, taking around 0.2 msec each for execution. The *Channel* vertices are the runner-up, taking about ten times less execution time.

Execution Time Histograms

The execution time histograms reveal that the execution time of the vertices are neither stable nor all similar to each other. Figure 5.7 shows the logarithmic histograms for the *MasterBuffer* in purple and *EffectsB* vertex. The former executing in average in 0.002 msec indicated by the amplitude at around 0 msec in the histogram. Sometimes though, it executes a lot slower, with individual measurements taking more than fifty times longer. The latter's execution time is more stable. The longest measured execution time is about three times larger than the average.

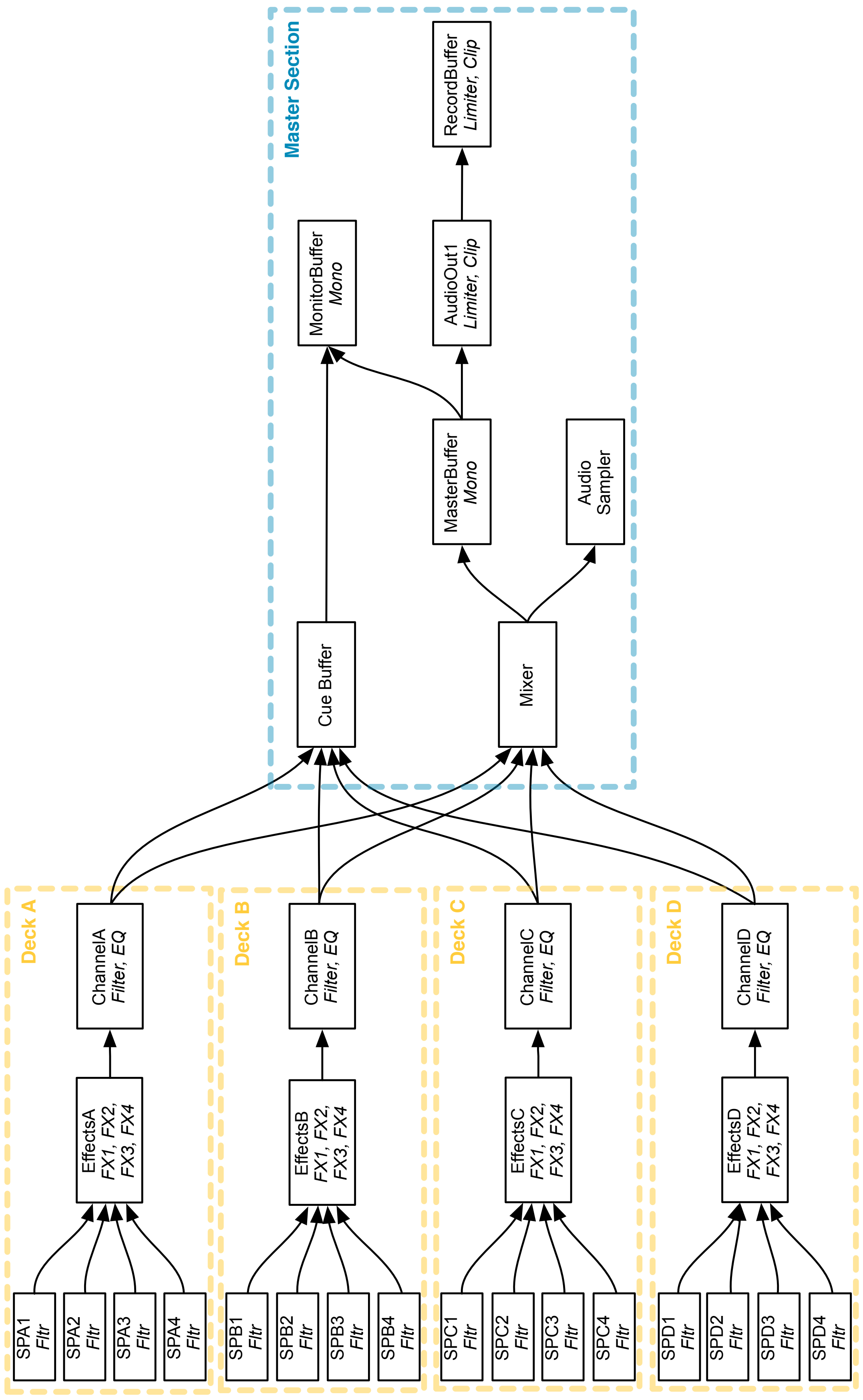


Figure 5.6.: Actual Audio Graph configuration in the experiment setup.

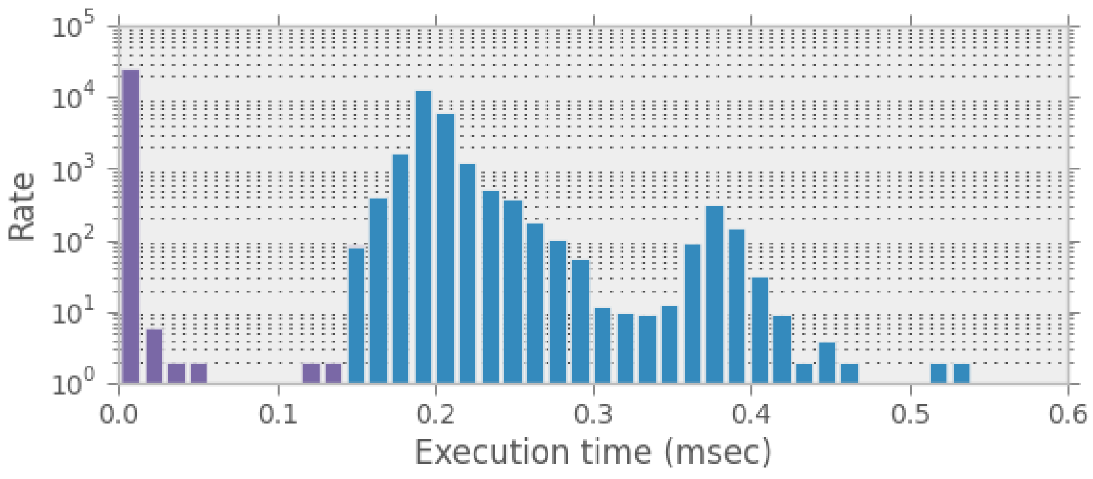


Figure 5.7.: Logarithmic execution time distribution exemplary for MasterBuffer (purple) and an Effects vertex (blue).

5.7. Vertex Execution Times In General

Though the structure of the graph is fixed, it can be configured in a lot of ways. Changing the configuration changes the execution times of the graph vertices.

The length of the execution time of the graph vertices depends on 14 individual configuration parameters. Each of them can have two or more states. The following list shows examples of configuration parameters.

Mixing is *internal* or *external*,

Number of audio sources is *2* or *4*,

Time stretching is *on* or *off*,

Effect can be any of the effects introduced in /F13/,

Filter is a real number between 0 and 1, indicating the amount of filter to apply,

Audio source tempo is a percentage value between 0 and 200, with 100% being normal playback and 200% playback at double speed.

All of these configuration parameter states can be mixed with each other, resulting in hundreds of different states of the audio graph.

5.8. Limitations Through The Real-Time Constraint

Because of the application's real-time behavior (inequation 4.1), the audio data and manipulation parameter become visible at defined intervals, the signal that starts the APC. It is not possible to use speculative forecasting because the parameters have continuous value ranges, and the calculation cannot be interpolated without being noticeable in the audio signal. Therefore, the intuitive approach of using a pipeline architecture for the audio engine is not applicable, because the parallel and replicable pipeline stages cannot be filled.

5.9. Summary

This chapter presented DJ-Star's runtime behavior. It was found that no single algorithm makes up for the computing power but a combination of algorithms that are all run in the Audio Engine subsystem. The scheduling of these algorithms, which are heavily data-dependent on each other, is crucial for a good performance of the system.

6. Improved Scheduling Strategies

This chapter first interprets the execution of the audio graph as a scheduling problem, and explains the scheduling characteristics of the current graph execution algorithm. Next, it gives theoretical speed-up bounds induced by the structure of the graph, concluding with two improved strategies that address the potentials for improvement from the performance analysis.

6.1. Identified Scheduling Problem

The execution of the audio graph is interpreted from now on as a scheduling problem. Section 2.1.2 introduces the background of the used scheduling model. Table 6.1 maps the scheduling model to this specific form.

6.2. Current Scheduling Strategy (ORIG)

DJ-Star's current scheduling strategy is a non-preemptive static list policy defined by Pinedo as follows. "Under a non-preemptive static list policy the decision maker orders the jobs at time zero according to a priority list. This priority list does not change during the evolution of the process and every time a machine is freed the next job on the list is selected for processing" [Pin12]. The priority list is called *vertexCollection* in DJ-Star.

The static list is ordered by the level of the vertex in the graph.

Scheduling model identifier	DJ-Star
Parallel machine p with m processors	One parallel machine p with m processors.
N jobs	N vertices in the audio graph
Release time r_i	Release time is 0 for all jobs: $\forall j_i \in J : r_i = 0$.
Precedence relations	Dependencies between tasks in the audio graph. For example: (*EffectsA*, *ChannelA*) in figure 5.6.
Deadline d_i	The deadline is the same for all jobs: $\forall j_i : d_i = X$.
Processing time p_j	The processing time p_j for a vertex is not known in advance, and will only become known upon completion.

Table 6.1.: Mapping of scheduling model identifiers to DJ-Star.

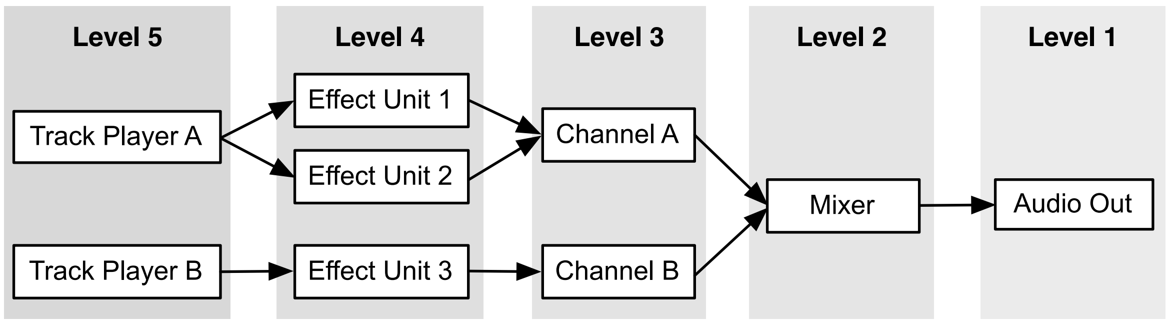


Figure 6.1.: Example audio graph with corresponding levels.

Given the example audio graph (fig 6.1), vertices on the left side have the highest level, while vertices to the right have the lowest level. For example, *Track Player A* has a higher level than *Effect Unit 1* and is therefore ranked before *Effect Unit 1* in the priority list. The ordering policy, however, does not specify the order of *Track Player A* and *Track Player B* since they lie on the same level.

6.3. Lower Bound

This and the following section introduce two lower bounds for the execution time of the audio graph. The scheduling is done offline with average vertex execution times simulated in RESCON [DDH11]. Table B in the appendix matches the vertex IDs and the corresponding names.

The first bound is known as an *earliest start scheduling strategy (ES)*. It schedules each vertex as soon as all its dependencies have finished executing, disregarding resource constraints. The earliest start scheduling is similar to the critical path analysis, but in addition it reveals the maximum concurrency in the respective graph.

Figure 6.2 visualizes the maximum concurrency possible with the current audio graph. The graph execution can be divided into three stages St_1, St_2 and St_3. In the beginning of the graph execution in St_1, up to 33 vertices are executed in parallel. But the execution time of those vertices is relatively short compared to the whole graph execution time. Next, in St_2, the execution time of the vertices is much longer, but the number of concurrent executing vertices dropped to four (with the exception of vertex *1*). Stage St_2 has the biggest influence on the execution time of the graph, with the execution of the vertices *8*, *51*, *34* and *33* calculating the effects of the four audio sources. Finally, in St_3, the number of concurrent vertices is even less than in St_2. The last two vertices, *15* and *6*, have to be executed serially, because *6* depends on *15*.

Since the ES scheduling starts each task as soon as possible, the scheduling results in figure 6.2 show that the maximum possible concurrency is bound by approximately four because of most time consuming stage St_2 being limited to four concurrent vertices.

6.4. Lower Bound With Resource Constraints

The second lower bound is even more precise because it takes the resource constraint of processors into account. Figure 6.3 shows the scheduling result calculated by the *tabu search* heuristic in RESCON, which “relies on list scheduling and includes a simple diversification scheme” [DDH11]. The tabu search option was used because the scheduling problem is NP-hard and RESCON is not able to calculate an exact solution for this graph

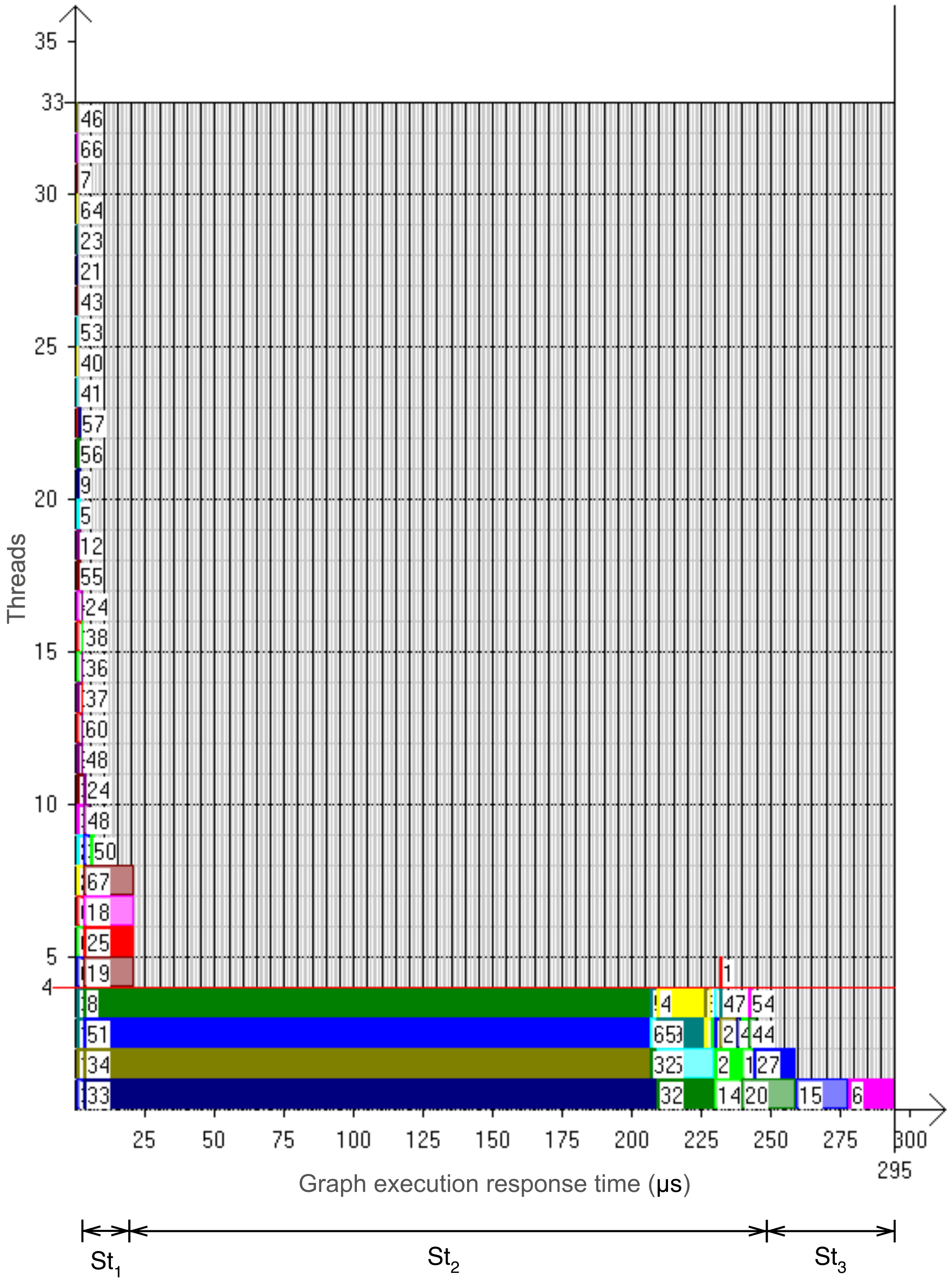


Figure 6.2.: Earliest start scheduling revealing the lower bound of graph execution time.

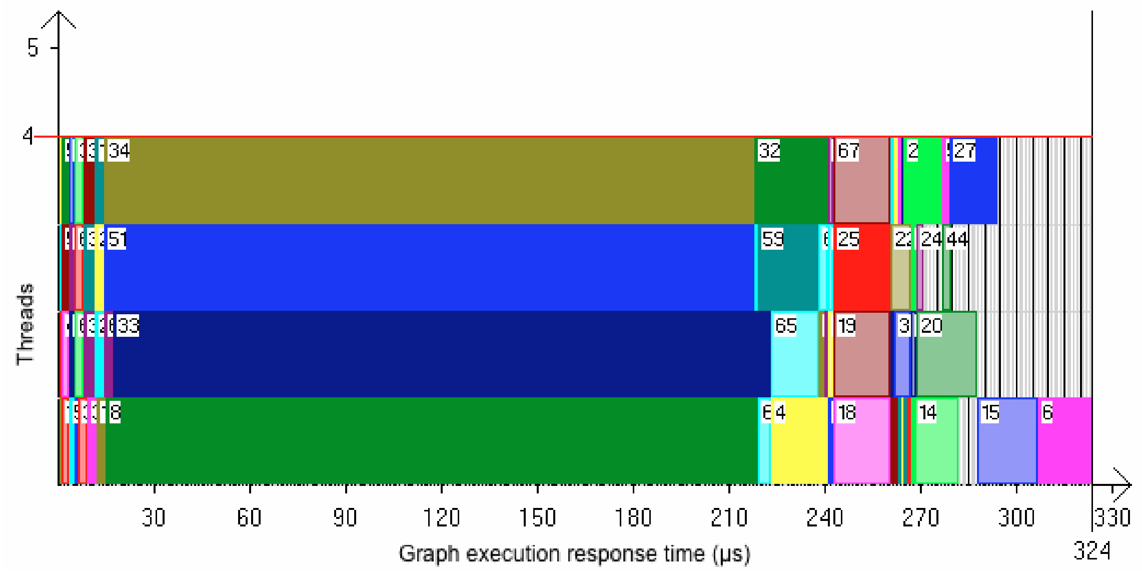


Figure 6.3.: Scheduling on four cores.

instance. The tabu search was allowed to run for twelve hours to retrieve a good solution. The makespan of the calculated scheduling is 9% bigger than the makespan of the ES-strategy, because the vertices in St_1 are being scheduled on four processors.

6.5. Sleep Scheduling (SLEEP)

The performance analysis showed that active waiting is used as a synchronization mechanism for the dependencies in the audio graph. This active waiting wastes 13% of the computing power of DJ-Star. This scheduling strategy aims at improving the active waiting (potential for improvement 5.1) by sending threads to sleep. The sleeping thread will then be woken up again, when the execution of the vertex can begin.

The mechanism is depicted in figure 6.4. In (a) thread T_2 just finished execution of *ChannelA* and moves on to *Cue Buffer*. When checking the dependencies, it finds out that it cannot start the execution because of the dependency to *ChannelB* which is still executing. T_2 then registers itself with *Cue Buffer* as executor and goes to sleep. As soon as T_1 finishes the execution of *ChannelB* (b), it signals *Cue Buffer* that all dependencies are executed and subsequently *Cue Buffer* wakes up T_2 which immediately starts the execution.

The SLEEP strategy enhances the design flaw of active waiting introduced in potential for improvement 5.1 by not wasting any processor power. However, while a thread is sleeping there could be another vertex ready for execution. This potential for improvement is introduced in 5.2 and will be addressed with the second scheduling strategy WS in the following section.

6.6. Work Stealing Scheduling (WS)

The work stealing strategy addresses the problem that threads do not look for other executable vertices but go to sleep when they find a vertex that needs to be executed but has unfinished dependencies (potential for improvement 5.2).

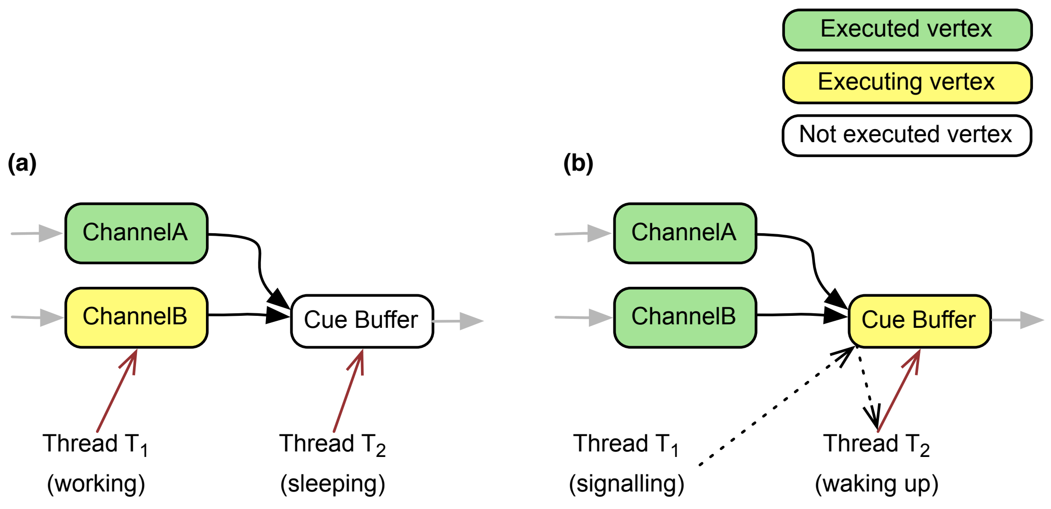


Figure 6.4.: Thread T_2 goes to sleep (a) and is woken up by T_1 after after it finished ChannelB (b).

Changing The Waiting List

First, the waiting queue is changed to only contain vertices that do not have unfinished dependencies. This new waiting list will now also be write-accessed, making mutual exclusion necessary to prevent data races. This regulated access becomes a bottleneck for access of multiple threads simultaneously, therefore the waiting list is split in a way that each worker thread has its own waiting list.

Subsequent Vertex Queueing

Each time a worker thread finishes the execution of a vertex v, it checks all successors of v for being ready for execution. It then adds these successors to its waiting list.

Initialization

In the beginning of an audio engine update the waiting lists have to be initialized. This is done by the master thread before it wakes up the worker threads. In this initialization, all vertices that do not depend on other vertices (i.e. input vertices) are added to the lists (fig. 6.5 (a)).

The input vertices are categorized as *Deck A*-, *Deck B*-, *Deck C*-, *Deck D*- or *Master Section*- related (see fig. 5.6 for the assignment) before they are added to the waiting list. The categories are then distributed over the waiting lists. This aims at cache efficiency because vertices in one category work on related audio data, which should be avoided to move between processors.

Load Balancing

The multiple waiting lists introduce a load balancing problem. One worker thread can run out of work quickly while another worker thread still has a lot of executable vertices in his waiting list. Therefore, a *work stealing* approach is introduced, where a worker thread that finds its list empty, will try to steal work from the waiting list of one of the other worker threads, as depicted in figure 6.5 (b). If all other lists are also empty, it goes to sleep.

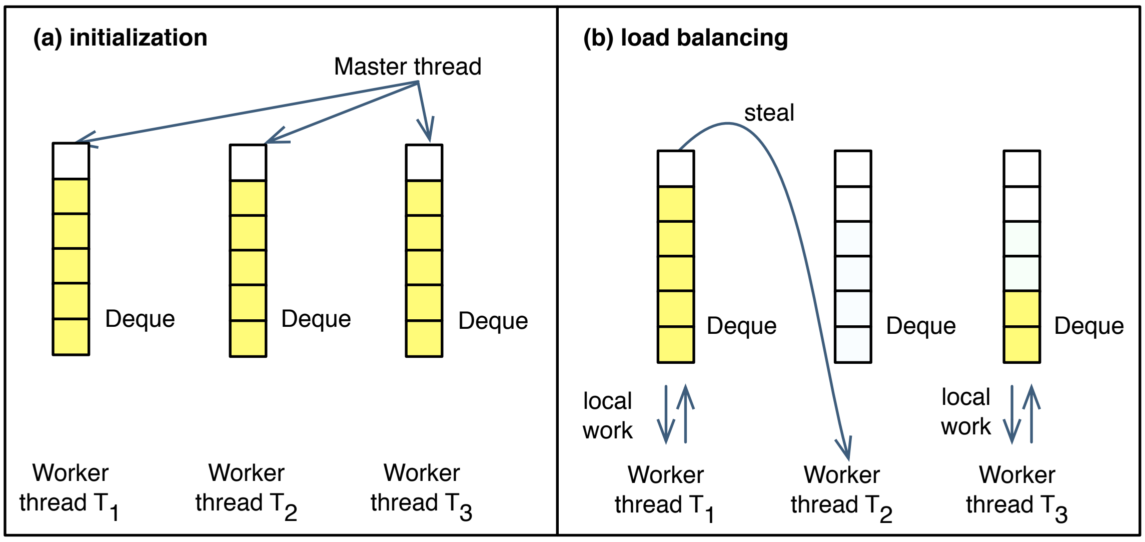


Figure 6.5.: Work stealing with 3 threads.

Waiting List Implementation

The waiting list is implemented as a *double ended queue* (deque) that (as the name suggests) can be accessed from both sides. Stealing threads access the queue at the top and local working threads access the queue at the bottom (fig. 6.5). This makes it possible for a steal and a local access to happen at the same time (if $length(deque) \geq 2$). Another advantage is the cache efficiency: A thread prioritizes nodes that it put in last (LIFO principle), maximizing the chance of the related data being already in the processors cache. On the other hand, if a thread steals a node n, it always gets the node with the longest waiting time in the queue. Transferred to the underlying graph model, this means n will be leftmost in the graph and will therefore produce a maximum number of new tasks after its completion. These new tasks can then be processed locally by the stealing thread.

7. Evaluation

This chapter analyzes the behaviour of the improved scheduling strategies by measuring the response time of the graph execution with the improved scheduling strategies and comparing the results against the original strategy. The measurements were recorded using a machine with the following specifications:

Processor 8 cores, 3.1 GHz clock speed (AMDFX8120h)

Memory 8 GB in dual channel mode (PC3-10700)

Hard Disk Drive 128 GB Solid State Disk (SamsungSSD830)

Operating System Windows 7 64-Bit Premium

7.1. Response Time

The response time specifies the time span from the beginning to the completion of the graph execution. It was measured for the original scheduling strategy ORIG and the improved strategies SLEEP (which sends threads to sleep instead of actively awaiting) and WS (which does not send threads to sleep while work is available).

The averaged runtimes of 10K graph executions are shown in table 7.1, using up to four threads since the maximum concurrency of the audio graph is bound by four (found in section 6.3). These results indicate that the difference in the average runtime of the scheduling strategies is rather small compared to the runtime itself. Though the SLEEP strategy with three threads appears to be an unfavorable configuration.

Figure 7.1 shows a comparison of the speed-up, confirming the results from the average graph response times, visualizing the very similar response times, with the exception of the SLEEP strategy utilizing three threads.

7.2. Real-Time Constraint

If the audio graph execution takes too long, the sound on the speakers will be distorted. Therefore, not only the average response time of the audio graph execution is important but also worst case execution times should be as low as possible.

Threads	1	2	3	4
ORIG	1.0785	0.6371	0.5683	0.4516
SLEEP	1.1130	0.6447	0.6444	0.4657
WS	1.1111	0.6394	0.5844	0.4690

Table 7.1.: Average response times for the graph execution of the scheduling strategies (ms).

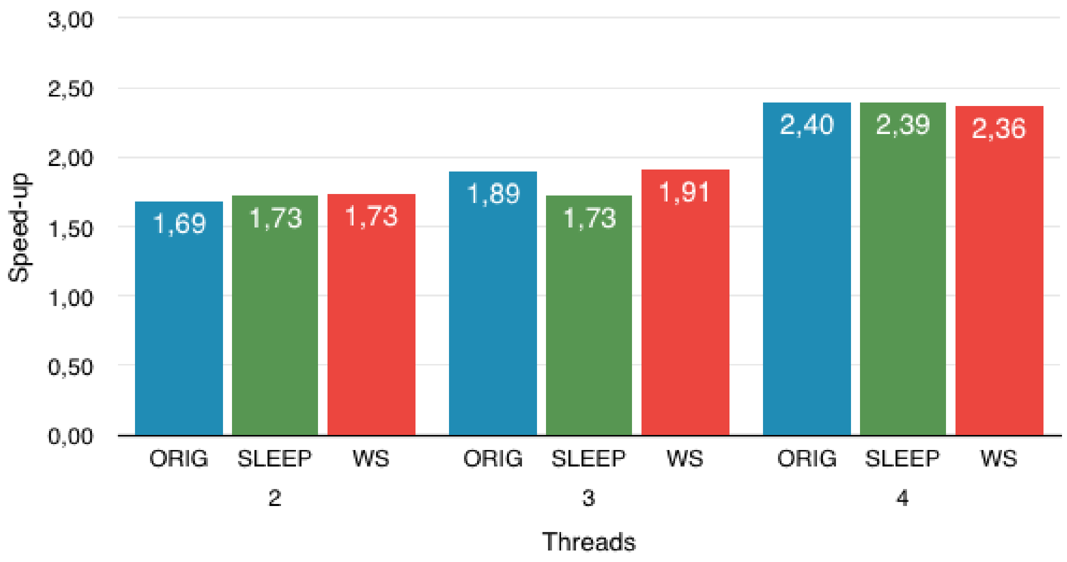


Figure 7.1.: Speed-up of the scheduling strategies.

Upper Bound For The Runtime Of The Graph Execution

The runtime of the graph execution $T(Exec)$ is bound by the maximum runtime of one APC minus the runtime of the other calculations *Timecode Processing* TP, *Graph Preprocessing* Pre and *Various Calculations*[1] VC (see chapter 4):

$$T(APC) = T(TP) + T(Pre) + T(Exec) + T(VC).$$

$T(APC)$ has to be smaller than 2.9 ms (according to the real-time constraint 4.1), and the runtimes of the APC components where empirically averaged by measuring 10K APC's to be

$$T(TP) = 0.28ms,$$
$$T(Pre) = 0.43ms,$$
$$T(VC) = 0.09ms.$$

In conclusion, the graph execution needs to be faster than 2.1 ms in order to satisfy the real-time constraint:

$$T(Exec) \leq 2.1ms. \tag{7.1}$$

[1] Accounting calculations, for example updating the master tempo.

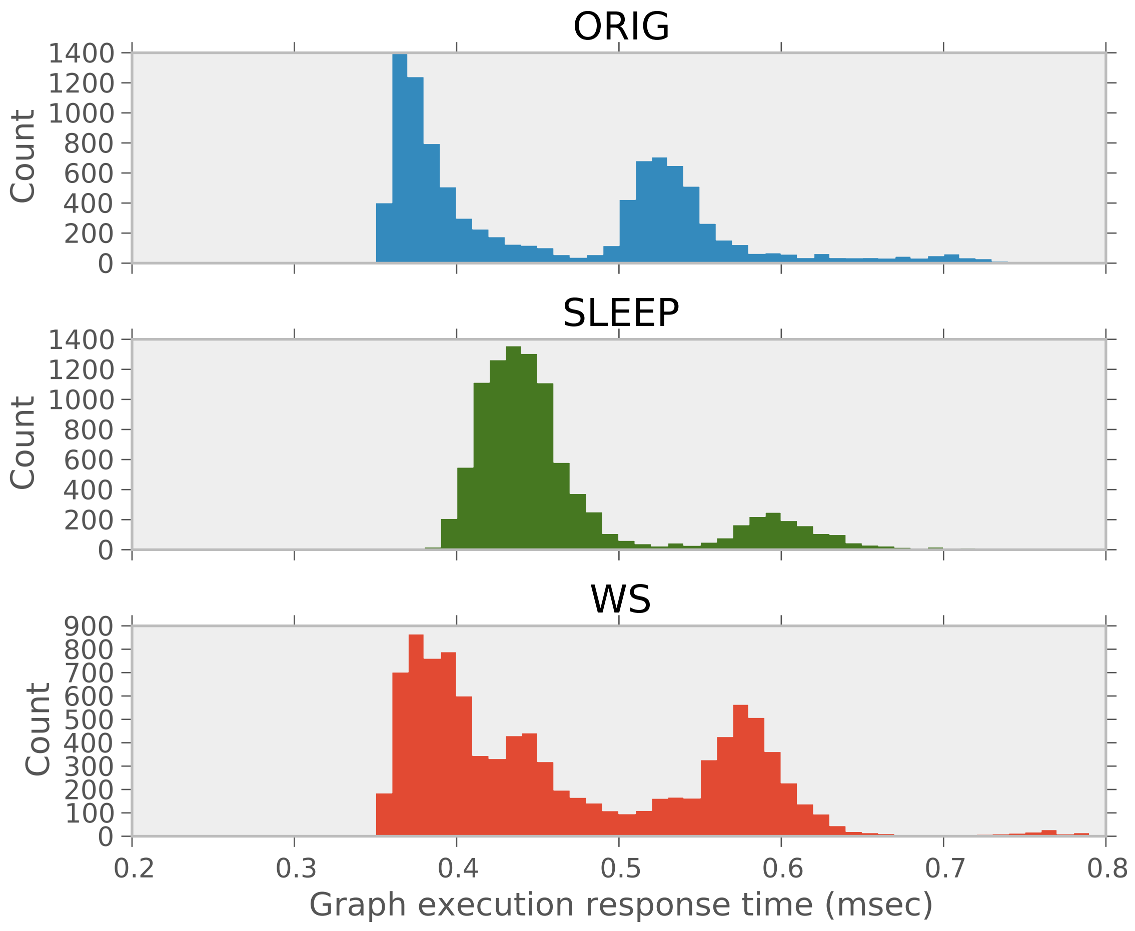


Figure 7.2.: Graph execution response time distributions of the scheduling strategies with four threads.

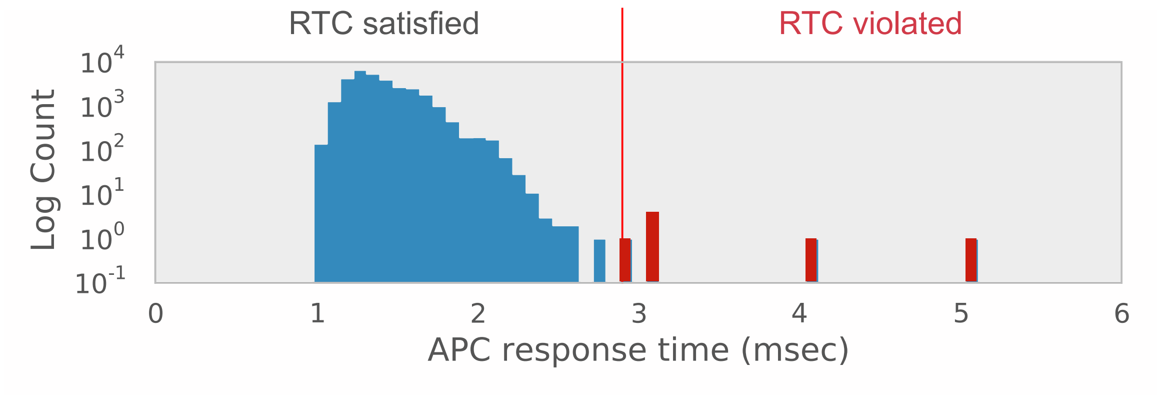


Figure 7.3.: APC response time distribution for ORIG.

Distributions Of The Graph Execution Length

To satisfy the real-time constraint, the distribution of the runtimes should spread as little as possible. The scheduling strategies all show a similar distribution (fig 7.2) by having two peaks.

The ORIG strategy features a very high left peak, the second peak being about half as big and a flat tail from 0.6 ms up to 0.73 ms.

The SLEEP strategy's distribution shows similar peaks compared to ORIG, but they shifted to the right by about 0.4 ms because of the added overhead for the context changes of the threads. In contrast to ORIG, it does not have a tail at the right end of the distribution.

The WS strategy's distribution again features two peaks, the left one being at the same position as ORIG at about 0.36 ms, but the amplitude is just about two-thirds the height of ORIG's left peak. The right peak is about the same height as with the original strategy ORIG. WS shows some occasional runtimes between 0.75 ms and 0.8 ms.

In conclusion, the original scheduling strategy ORIG indicates the best distribution because of the strong peak at around 0.36 ms. The tail at the right end could be improved, but does not cause any problems for the real-time condition. Likewise, the SLEEP and WS strategies satisfy the real-time constraint of a runtime smaller than 2.1 ms easily.

Distributions Of The APC Length

The last view on the real-time shows the combined runtime of all the partial calculations, or in other words the runtime of the APC itself. As stated in inequation 4.1, one APC can not take longer than 2.9 ms without violating the real-time constraint. The logarithmic distribution of the joint APC's runtimes are shown in figure 7.3 for the ORIG strategy using four threads, the red line indicating the realtime boundary. Though all of the graph executions by themselves are sufficiently fast, some of the measured APC's miss the real-time boundary because of an unfavorable combination of the runtime of the APC's parts. This violation occurred five times in the measured 10K APC's.

7.3. CPU Core Assignment

Figure 7.4 visualizes the number of threads involved in the execution of one hundred consecutive graphs, revealing that not all of the four threads contribute to the execution

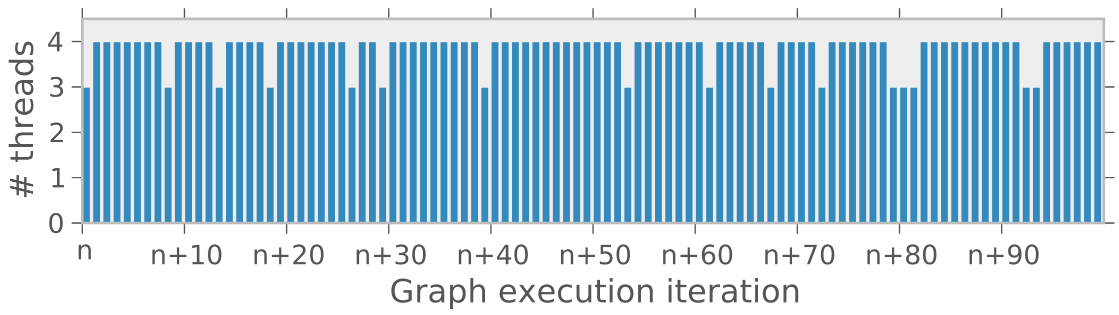


Figure 7.4.: Excerpt showing the actual number of threads per graph execution for ORIG which should be executed by four threads.

all the time. In about 20% of the graph executions, the actual number of contributing threads is just three. Even in those cases the threads are proven to get the wake-up signal, but as they are ready to participate, the graph execution is already completed.

The fraction of graph executions with less than four contributing threads is 18% for ORIG and 21% for the WS strategy. This yields in average 3.81 threads for ORIG, and 3.78 threads for WS of the desired four threads (table 7.2). With the SLEEP strategy, only 2% of the graph executions are completed with less than four threads, resulting in 3.98 thread in average per graph execution.

Scheduling Strategy	ORIG	SLEEP	WS
Average threads per graph execution	3.81	3.98	3.78
Fraction of graph executions using less than four threads	18%	2%	21%

Table 7.2.: Avg. threads per graph execution with four threads.

For another comparison of the scheduling strategies performances, the average response time was calculated only on graph executions where four threads participated (table 7.3). The results of the SLEEP strategy does not change a lot, due to the average threads of 3.98 per graph execution. However, WS outperforms ORIG by 19 µs because WS has slightly less average threads per graph execution, which does not have an impact anymore.

Scheduling Strategy	ORIG	SLEEP	WS
Avg. response time of graph executions with four threads	0.4379	0.4614	0.4353

Table 7.3.: Average response times of graph executions utilizing four threads (ms).

7.4. Typical Scheduling Realizations

Typically measured realizations of the scheduling strategies are shown in figure 7.5. The threads on the y-axis are colored differently each and the number of vertices with very short execution times have been omitted to increase the visibility. Table B gives a mapping of the numbers to the vertex names.

The similar response times already indicated that the actual scheduling might look similar and it turned out to be true. ORIG's small gaps are based on overhead of traversing the `vertexCollection` to find executable vertices. In contrast to the other scheduling strategies, it does not show any big gaps because it waits actively when it can not execute a vertex. Active waiting is indicated by the grey boxes inside the colored boxes, the length of the grey box indicating the length of the active waiting period. The main difference between WS and ORIG is that WS already schedules a lot of the small jobs before executing the effect jobs 34, 9, 51 and 35, while ORIG starts earlier with the effect jobs and executes the small jobs afterwards.

The SLEEP strategies scheduling looks very similar to ORIG, due to the fact that they utilize the same prioritization of the vertices. Since SLEEP has a slower response time than ORIG, the scheduling looks stretched. The big gaps arise from the situation where the threads go to sleep instead of waiting actively. The grey colored boxes indicating active waiting are not present any more in the scheduling of the SLEEP strategy.

The typical scheduling realization of the WS strategy reveals a response time in between the ORIG and SLEEP strategies response time. Since WS does not allow for active waiting, no grey boxes are visible. Just as seen with the SLEEP strategy, when no vertices are ready for execution, the the threads go to sleep (indicated by the big gaps).

7.5. ORIG Simulation

The evaluation showed, that the WS scheduling strategy yields about the same response time than the ORIG strategy, though the theoretical lower bound with resource constraints (section 6.4) indicated that a performance gain is possible. Hence, the ORIG strategy was implemented in RESCON to directly compare the results of the two strategies simulations.

The resulting scheduling (fig. 7.6) yields a response time of 327 µs which is near-optimal compared to the 324 µs response time of the lower bound with resource constraints. The difference in response time between the simulated lower bound with resource constraints and the average measurements (452 µs) is most likely based on the varying execution times of the vertices, negatively influencing the overall response time.

7.6. Summary

The evaluation showed that the response time of the scheduling strategies is similar, SLEEP not being much slower and WS not being much faster than the original strategy ORIG.

The SLEEP strategy almost reached the same response time as the ORIG strategy, but with the benefit that the threads did not waste processor cycles by waiting actively.

Then again, the WS strategy, which does not send threads to sleep as long as vertices are ready for execution, was a little bit slower when calculating the average response time on all measurements, but a little bit faster when calculating the average response time only on graph executions where four threads participated.

In conclusion, the ORIG strategy is already a near-optimal scheduling strategy for the audio graph instance in DJ-Star.

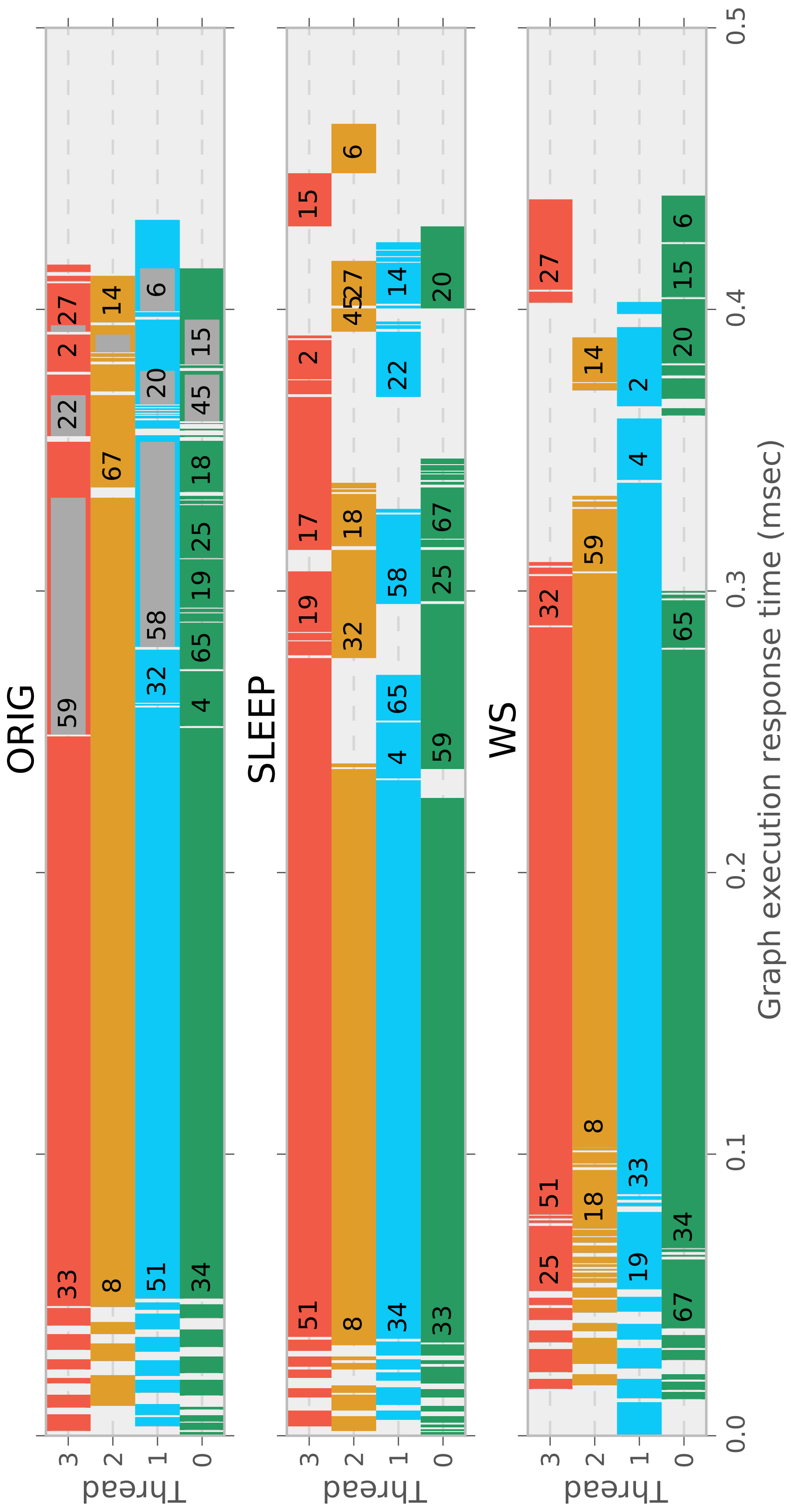


Figure 7.5.: Comparison of typical realizations of the scheduling strategies with four threads.

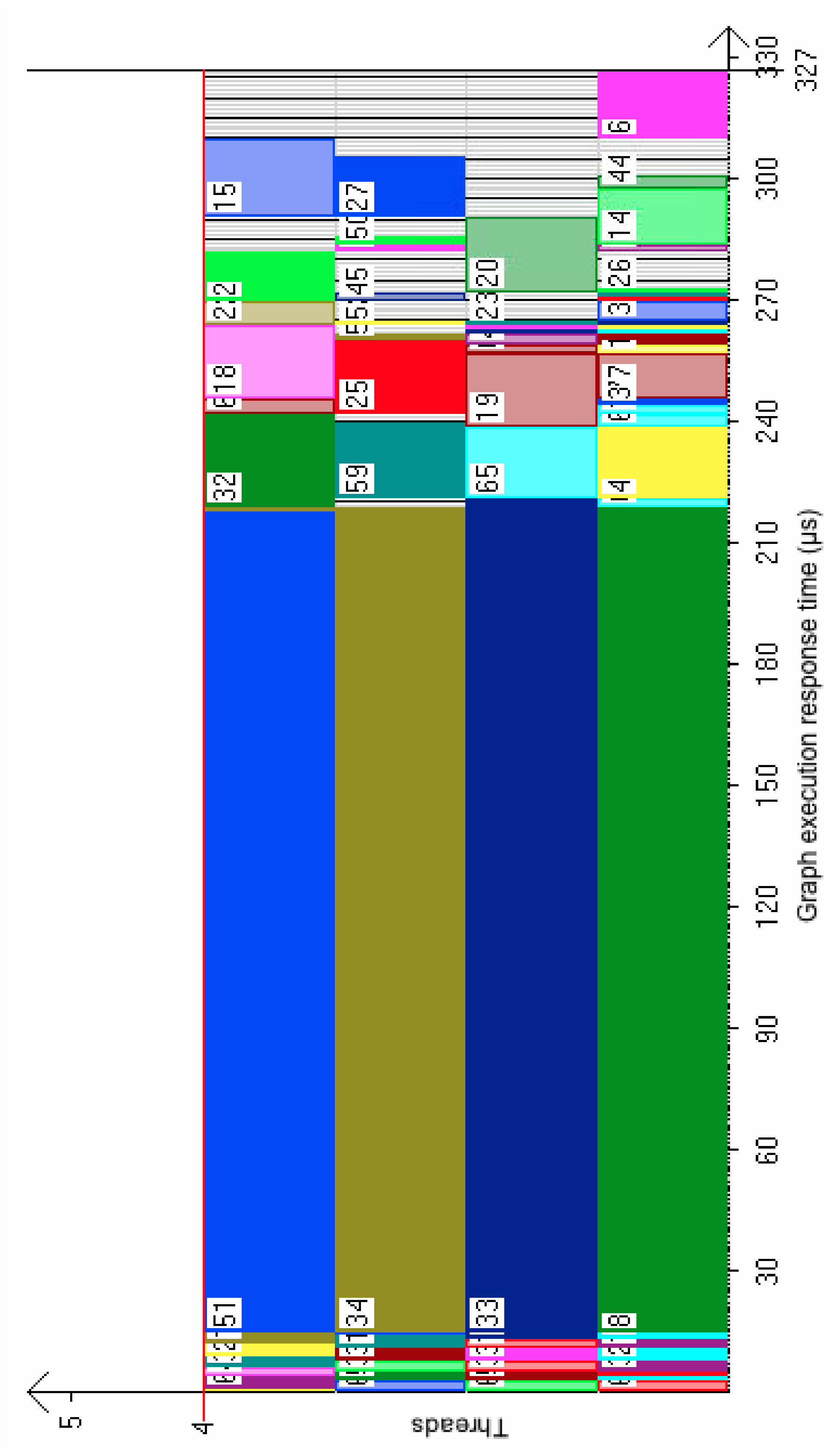


Figure 7.6.: Scheduling results for the simulation of ORIG.

8. Conclusion and Outlook

8.1. Conclusion

In this thesis, different parallelization strategies for a real-time audio application were tested against each other. Initially, the application's structure and run-time behavior had to be analyzed in depth. The result of this analysis was that no single algorithm, but a series of transformations on the audio signal in the central audio-engine subsystem accounts for most consumption of computing power. Due to the real-time behavior of the application, the intuitive approach of using a pipeline architecture for parallelization of a multimedia application was not feasible.

Instead, the series of transformations was interpreted as a scheduling problem, and two potentials for performance improvements were identified. The improvements were addressed by two enhanced scheduling strategies, which were then tested against the original strategy.

The evaluation showed that the first new scheduling strategy, SLEEP, is not much slower, but stops the application from waiting actively, freeing 13% of the computing power for other applications. The second improved scheduling strategy, WS, did not wait actively, but in addition never put any threads to sleep as long as there was work available. WS proved to be as fast as the original scheduling strategy, while being slightly slower using one measurement and slightly faster using another one.

The next section provides additional ideas, which could allow the enhanced WS strategy to outperform the original strategy.

8.2. Outlook

With the current series of transformations, the enhanced WS strategy is not able to outperform the original strategy. Some of the ideas in this section might enable the WS strategy to perform at its full potential and to outperform the original strategy, while others were not implemented because they were considered to be *embarrassingly parallel.* The ideas for future work are:

Splitting Audio Channels: The audio data in the application is two-channel stereo. Currently, the transformation is calculated for both channels. If the transformation, however, would be calculated independently for the two channels, the current speed-up limit of around four could be doubled up to about eight.

Parallelizing The Audio Transformations: The individual audio transformations are not making use of concurrent execution so far. Especially, the effect transformations proved to be very computing-expensive compared to other transformations, and should therefore be parallelized to allow for a faster manipulation of the audio signal, and increase the maximum speed-up.

Joining Small Transformations: Many of the transformations on the audio signal take very little time to calculate. These could be joined to reduce the overhead of assigning and tracking these transformations.

Parallel Track Preprocessing: The tracks have to be preprocessed before they can be played back inside the application. This preprocessing is not done in parallel yet. In the common use case, tens or hundreds of audio tracks are preprocessed in batch, so a naive parallelization, which preprocesses multiple audio tracks concurrently, could speed up the preprocessing drastically. But because this was considered to be *embarrassingly parallel*, it is not part of this thesis.

Parallel Timecode Processing: The processing of the timecode signal accounts for a significant amount of computing time. The timecode signal for different audio decks is not processed concurrently yet. This, again, *embarrassingly parallel* optimization could improve the runtime of the application severely.

Appendix

A. Average Vertex Execution Times (ms)

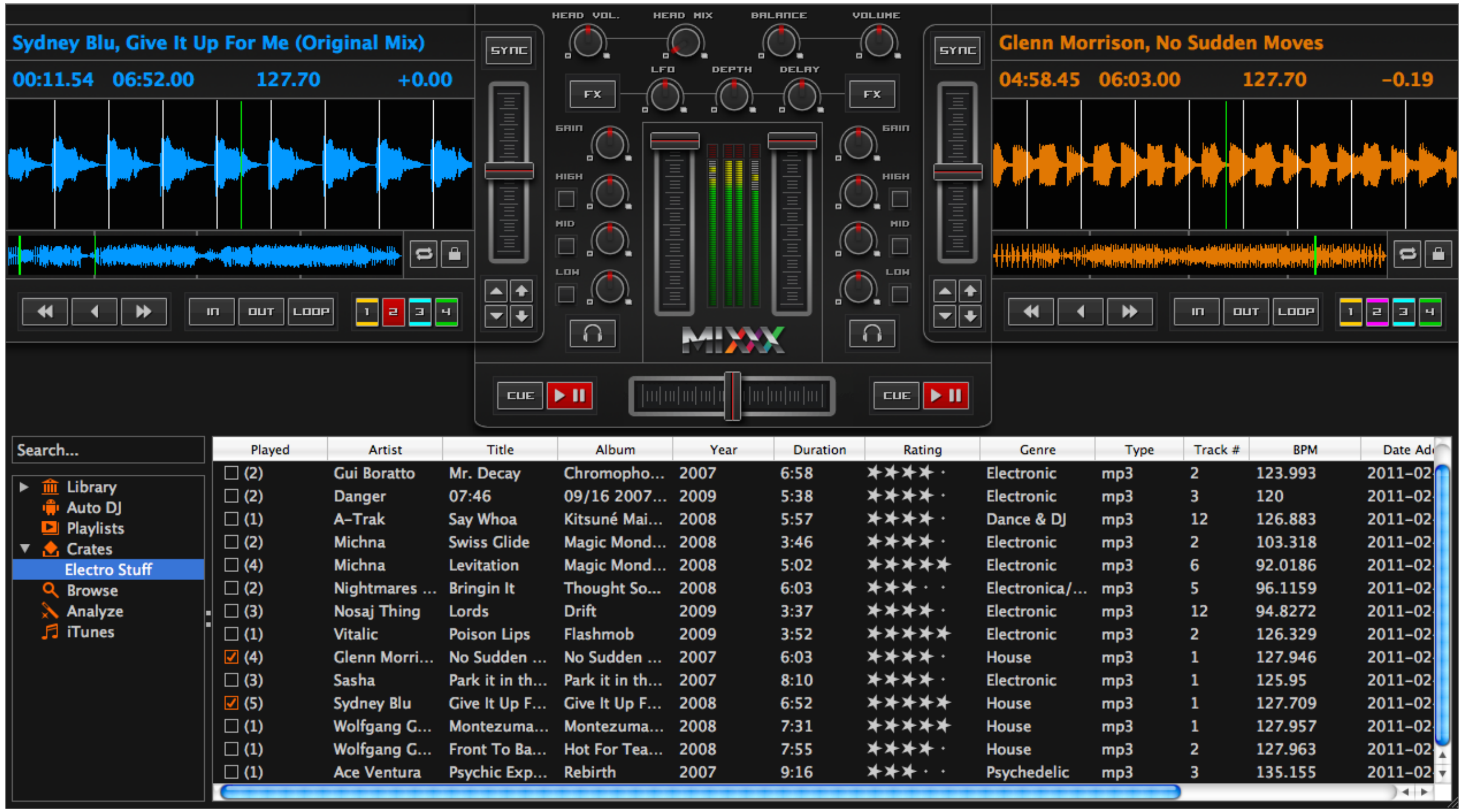
Vertex name ↑	Avg. exec. time
.Audio Out 1	0.0185
.Audio Out 2	0.0149
.Audio Out 3	0.0168
.Audio Out 4	0.0144
.Audio Out 5	0.0015
.Audio Out 6	0.0018
.Aux In	0.0009
.Send Effects In	0.0047
ChannelA.Audio In 1	0.0018
ChannelA.ChannelA	0.0192
ChannelA.Click	0.0011
ChannelA.External Mixer Out	0.0013
ChannelA.Effects	0.2042
ChannelA.Effects Bypass	0.0182
ChannelA.Post Fader Effects	0.0021
ChannelA.SamplePlayer0	0.0027
ChannelA.SamplePlayer1	0.0025
ChannelA.SamplePlayer2	0.0025
ChannelA.SamplePlayer3	0.0024
ChannelA.TrackPlayer	0.0015
ChannelB.Audio In 2	0.0019
ChannelB.ChannelB	0.0227
ChannelB.Click	0.0008
ChannelB.External Mixer Out	0.0011
ChannelB.Effects	0.2037
ChannelB.Effects Bypass	0.0185
ChannelB.Post Fader Effects	0.0024
ChannelB.SamplePlayer0	0.0031
ChannelB.SamplePlayer1	0.0029
ChannelB.SamplePlayer2	0.0026
ChannelB.SamplePlayer3	0.0030
ChannelB.TrackPlayer	0.0015
ChannelC.Audio In 3	0.0010

Vertex name ↑	Avg. exec. time
ChannelC.ChannelC	0.0229
ChannelC.Click	0.0009
ChannelC.External Mixer Out	0.0010
ChannelC.Effects	0.2037
ChannelC.Effects Bypass	0.0183
ChannelC.Post Fader Effects	0.0019
ChannelC.SamplePlayer0	0.0027
ChannelC.SamplePlayer1	0.0028
ChannelC.SamplePlayer2	0.0030
ChannelC.SamplePlayer3	0.0027
ChannelC.TrackPlayer	0.0013
ChannelD.Audio In 4	0.0017
ChannelD.ChannelD	0.0185
ChannelD.Click	0.0011
ChannelD.External Mixer Out	0.0009
ChannelD.Effects	0.2062
ChannelD.Effects Bypass	0.0183
ChannelD.Post Fader Effects	0.0019
ChannelD.SamplePlayer0	0.0026
ChannelD.SamplePlayer1	0.0025
ChannelD.SamplePlayer2	0.0030
ChannelD.SamplePlayer3	0.0026
ChannelD.TrackPlayer	0.0018
Master Section.Cue Buffer	0.0122
Master Section.Master Buffer	0.0020
Master Section.Master Click	0.0007
Master Section.Master PreSampler	0.0060
Master Section.Microphone In	0.0031
Master Section.Monitor Buffer	0.0020
Master Section.Record Buffer	0.0187
Preview.Preview	0.0014
Preview.TrackPlayer	0.0008
Sampler.Audio Sampler	0.0027
Sampler.Audio Player	0.0006

B. Mapping Of Vertex IDs To Names

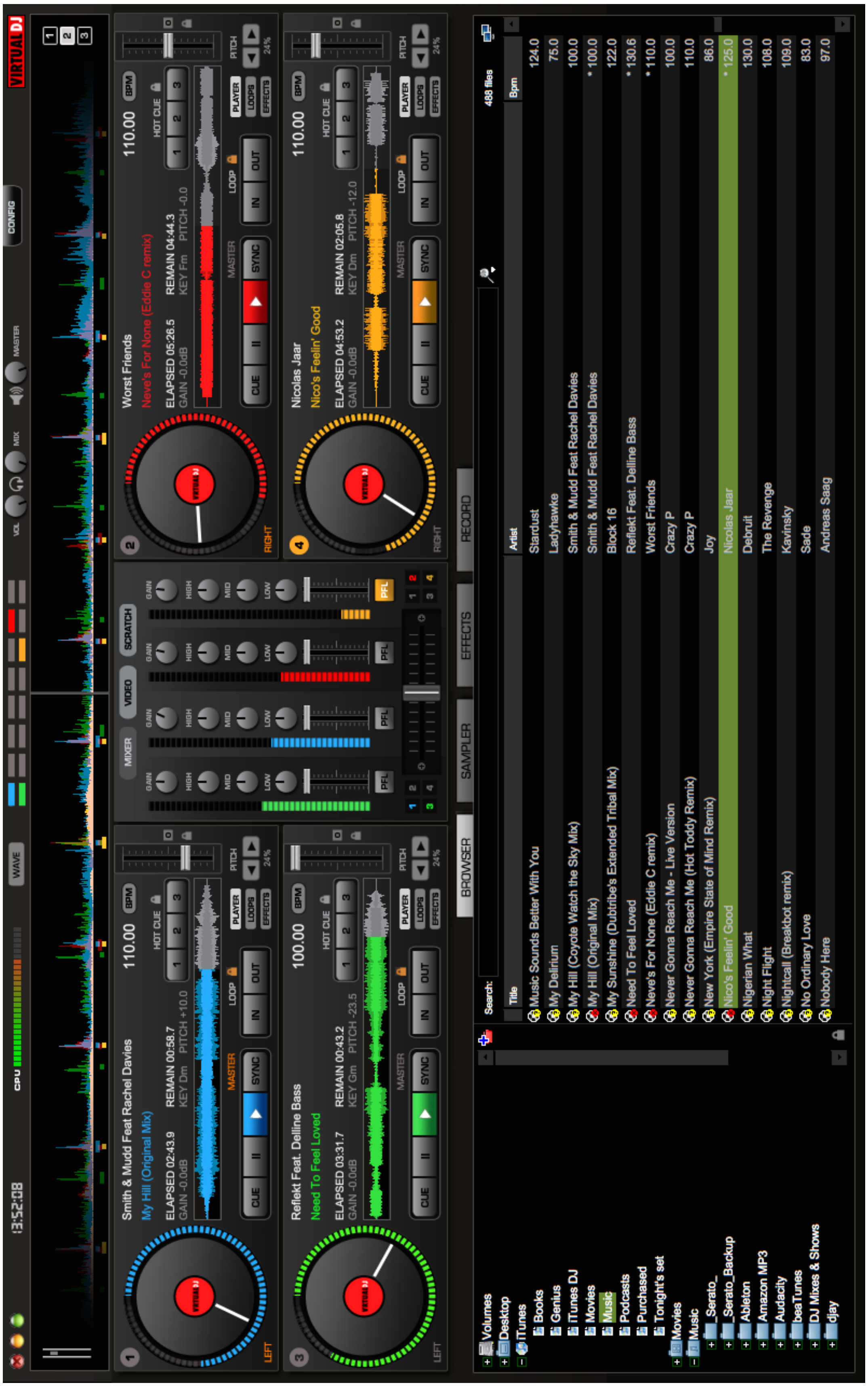
ID ↑	Vertex name	ID ↑	Vertex name
1	ChannelB.External Mixer Out	35	ChannelB.SamplePlayer0
2	Master Section.Cue Buffer	36	ChannelB.SamplePlayer1
3	ChannelC.Post Fader Effects	37	ChannelB.SamplePlayer2
4	ChannelD.ChannelD	38	ChannelB.SamplePlayer3
5	ChannelA.Audio In 1	39	.Send Effects In
6	.Audio Out 3	40	ChannelB.Click
7	Sampler.Audio Player	41	ChannelA.Click
8	ChannelA.Effects	42	ChannelB.TrackPlayer
9	ChannelD.TrackPlayer	43	Preview.TrackPlayer
10	ChannelA.SamplePlayer1	44	Sampler.Audio Sampler
11	ChannelA.SamplePlayer0	45	Master Section.Master Buffer
12	ChannelA.SamplePlayer3	46	.Aux In
13	ChannelA.SamplePlayer2	47	ChannelC.External Mixer Out
14	.Audio Out 4	48	Master Section.Microphone In
15	Master Section.Record Buffer	49	ChannelC.Audio In 3
16	ChannelD.Post Fader Effects	50	.Audio Out 6
17	ChannelB.Post Fader Effects	51	ChannelB.Effects
18	ChannelA.Effects Bypass	52	ChannelA.External Mixer Out
19	ChannelD.Effects Bypass	53	ChannelC.TrackPlayer
20	.Audio Out 1	54	Master Section.Monitor Buffer
21	Master Section.Master Click	55	ChannelD.Audio In 4
22	Master Section.Master PreSampler	56	ChannelB.Audio In 2
23	ChannelC.Click	57	Preview.Preview
24	.Audio Out 5	58	ChannelA.Post Fader Effects
25	ChannelB.Effects Bypass	59	ChannelA.ChannelA
26	ChannelD.External Mixer Out	60	ChannelD.SamplePlayer2
27	.Audio Out 2	61	ChannelD.SamplePlayer3
28	ChannelC.SamplePlayer3	62	ChannelD.SamplePlayer0
29	ChannelC.SamplePlayer2	63	ChannelD.SamplePlayer1
30	ChannelC.SamplePlayer1	64	ChannelA.TrackPlayer
31	ChannelC.SamplePlayer0	65	ChannelC.ChannelC
32	ChannelB.ChannelB	66	ChannelD.Click
33	ChannelD.Effects	67	ChannelC.Effects Bypass
34	ChannelC.Effects		

C. Screenshots Of The Sample Applications



Played	Artist	Title	Album	Year	Duration	Rating	Genre	Type	Track #	BPM	Date Ad
☐ (2)	Gui Boratto	Mr. Decay	Chromopho...	2007	6:58	★★★★·	Electronic	mp3	2	123.993	2011-02
☐ (2)	Danger	07:46	09/16 2007...	2009	5:38	★★★★·	Electronic	mp3	3	120	2011-02
☐ (1)	A-Trak	Say Whoa	Kitsuné Mai...	2008	5:57	★★★★·	Dance & DJ	mp3	12	126.883	2011-02
☐ (2)	Michna	Swiss Glide	Magic Mond...	2008	3:46	★★★★·	Electronic	mp3	2	103.318	2011-02
☐ (4)	Michna	Levitation	Magic Mond...	2008	5:02	★★★★★	Electronic	mp3	6	92.0186	2011-02
☐ (2)	Nightmares ...	Bringin It	Thought So...	2008	6:03	★★★··	Electronica/...	mp3	5	96.1159	2011-02
☐ (3)	Nosaj Thing	Lords	Drift	2009	3:37	★★★★·	Electronic	mp3	12	94.8272	2011-02
☐ (1)	Vitalic	Poison Lips	Flashmob	2009	3:52	★★★★★	Electronic	mp3	2	126.329	2011-02
☑ (4)	Glenn Morri...	No Sudden ...	No Sudden ...	2007	6:03	★★★★·	House	mp3	1	127.946	2011-02
☐ (3)	Sasha	Park it in th...	Park it in th...	2007	8:10	★★★★·	Electronic	mp3	1	125.95	2011-02
☑ (5)	Sydney Blu	Give It Up F...	Give It Up F...	2008	6:52	★★★★★	House	mp3	1	127.709	2011-02
☐ (1)	Wolfgang G...	Montezuma...	Montezuma...	2008	7:31	★★★★★	House	mp3	1	127.957	2011-02
☐ (1)	Wolfgang G...	Front To Ba...	Hot For Tea...	2008	7:55	★★★★·	House	mp3	2	127.963	2011-02
☐ (1)	Ace Ventura	Psychic Exp...	Rebirth	2007	9:16	★★★··	Psychedelic	mp3	3	135.155	2011-02

Figure C.1.: *Mixxx* [Mix]



Smith & Mudd Feat Rachel Davies
My Hill (Original Mix)
110.00 BPM
ELAPSED 02:43.9 REMAIN 00:58.7
GAIN -0.0dB KEY Dm PITCH +10.0

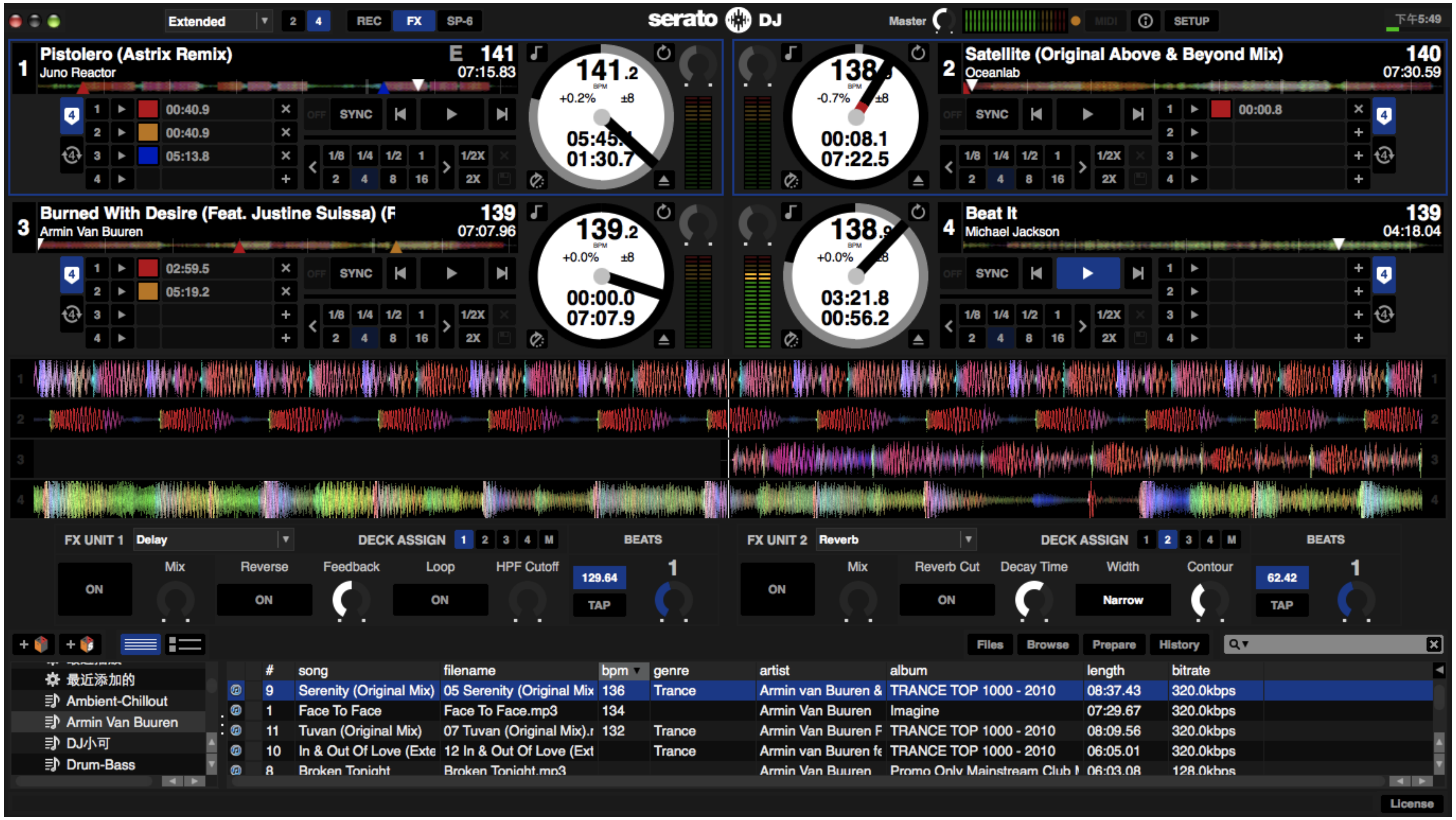
Worst Friends
Neve's For None (Eddie C remix)
110.00 BPM
ELAPSED 05:26.5 REMAIN 04:44.3
GAIN -0.0dB KEY Fm PITCH -0.0

Reflekt Feat. Delline Bass
Need To Feel Loved
100.00 BPM
ELAPSED 03:31.7 REMAIN 00:43.2
GAIN -0.0dB KEY Gm PITCH -23.5

Nicolas Jaar
Nico's Feelin' Good
110.00 BPM
ELAPSED 04:53.2 REMAIN 02:05.8
GAIN -0.0dB KEY Dm PITCH -12.0

Title	Artist	Bpm
Music Sounds Better With You	Stardust	124.0
My Delirium	Ladyhawke	75.0
My Hill (Coyote Watch the Sky Mix)	Smith & Mudd Feat Rachel Davies	100.0
My Hill (Original Mix)	Smith & Mudd Feat Rachel Davies	* 100.0
My Sunshine (Dubtribe's Extended Tribal Mix)	Block 16	122.0
Need To Feel Loved	Reflekt Feat. Delline Bass	* 130.6
Neve's For None (Eddie C remix)	Worst Friends	* 110.0
Never Gonna Reach Me - Live Version	Crazy P	100.0
Never Gonna Reach Me (Hot Toddy Remix)	Crazy P	110.0
New York (Empire State of Mind Remix)	Joy	86.0
Nico's Feelin' Good	Nicolas Jaar	* 125.0
Nigerian What	Debruit	130.0
Night Flight	The Revenge	108.0
Nightcall (Breakbot remix)	Kavinsky	109.0
No Ordinary Love	Sade	83.0
Nobody Here	Andreas Saag	97.0

Figure C.2.: *Virtual DJ* by *Atomix Productions* [Ato]



#	song	filename	bpm	genre	artist	album	length	bitrate
9	Serenity (Original Mix)	05 Serenity (Original Mix	136	Trance	Armin van Buuren &	TRANCE TOP 1000 - 2010	08:37.43	320.0kbps
1	Face To Face	Face To Face.mp3	134		Armin Van Buuren	Imagine	07:29.67	320.0kbps
11	Tuvan (Original Mix)	07 Tuvan (Original Mix).r	132	Trance	Armin Van Buuren F	TRANCE TOP 1000 - 2010	08:09.56	320.0kbps
10	In & Out Of Love (Exte	12 In & Out Of Love (Ext		Trance	Armin van Buuren fe	TRANCE TOP 1000 - 2010	06:05.01	320.0kbps
8	Broken Tonight	Broken Tonight.mp3			Armin Van Buuren	Promo Only Mainstream Club I	06:03.08	128.0kbps

Figure C.3.: *Serato DJ* by *Serato* [Ser]

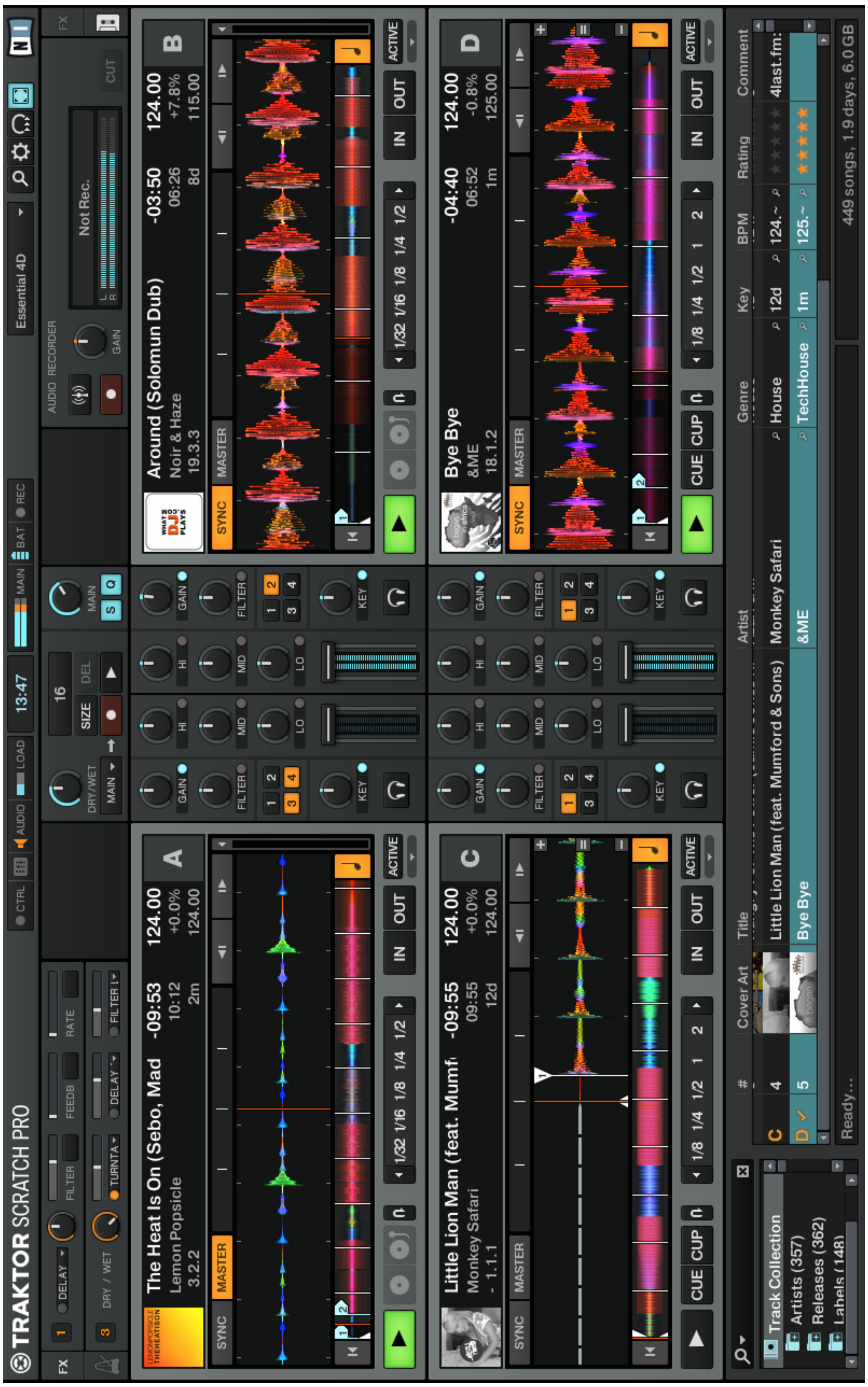


Figure C.4.: *Traktor Pro 2* by *Native Instruments* [Nat]

Bibliography

[Ato] Atomix Productions, “Virtual DJ Product Website.” [Online]. Available: http://www.virtualdj.com

[Ble11] T. Blechmann, “Supernova - A Multiprocessor Aware Real-Time Audio Synthesis Engine For SuperCollider,” Ph.D. dissertation, 2011.

[Bus98] F. Buschmann, **Pattern-orientierte Software-Architektur: ein Pattern-System**, ser. Professionelle Softwareentwicklung. Bonn: Addison-Wesley, 1998.

[CSB$^+$11] J. A. Colmenares, I. Saxton, E. Battenberg, R. Avizienis, N. Peters, K. Asanovi, J. D. Kubiatowicz, and D. Wessel, “Real-time musical applications on an experimental operating system for multi-core processors,” in **Proceedings of the International Computer Music Conference 2011**, no. August, 2011, pp. 216–223.

[DDH11] F. Deblaere, E. Demeulemeester, and W. Herroelen, “RESCON: Educational project scheduling software,” **Computer Applications in Engineering Education**, vol. 19, no. 2, pp. 327–336, Jun. 2011.

[HGL12] H.-M. Huang, C. Gill, and C. Lu, “MCFlow: A Real-Time Multi-core Aware Middleware for Dependent Task Graphs,” **2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**, no. 3, pp. 104–113, Aug. 2012.

[Mic] Microsoft Corporation, “Understanding Profiling Methods (in Visual Studio 2010).” [Online]. Available: http://msdn.microsoft.com/en-us/library/dd264994(v=vs.100).aspx

[Mix] Mixxx Development Team, “Mixxx Product Website.” [Online]. Available: http://www.mixxx.org/

[MSM04] T. G. Mattson, B. A. Sanders, and B. L. Massingill, **Patterns for Parallel Programming**, ser. Software Patterns Series. Munich: Addison-Wesley, 2004.

[Nat] Native Instruments, “Traktor Pro 2 Product Website.” [Online]. Available: http://www.native-instruments.com/products/traktor/

[Pin12] M. L. Pinedo, **Scheduling : theory, algorithms, and systems**, 4th ed. New York: Springer, 2012.

[Ran] Rane Corporation, “Rane TTM56S Mixer Product Page.” [Online]. Available: http://dj.rane.com/products/ttm56s-mixer/

[Ser] Serato, “Serato DJ Product Website.” [Online]. Available: http://serato.com/dj

[Wik] Wikipedia, “Disk Jockey.” [Online]. Available: http://en.wikipedia.org/wiki/Disc_jockey

[ZA11] U. Zölzer and X. Amatriain, **DAFX: digital audio effects**, 2nd ed. Chichester: John Wiley & Sons, Ltd, 2011, vol. 4.