

Audio Processing on a Multicore Platform

Daniel Sanz Ausin



Kongens Lyngby 2017
M.Sc.-2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk M.Sc.-2017

Abstract

The goal of this thesis is the design, implementation and evaluation of a real-time multicore audio processing platform. We propose a set of techniques and rules that allow multiple audio effect tasks distributed among the cores in the system to communicate and synchronize efficiently, given the constrained time requirements of real-time audio processing. The T-CREST platform has been used for the implementation. T-CREST is a time-predictable multi-processor platform for real-time embedded systems. The proposed solution allows multiple audio effects with different sample processing rates and communication requirements to be integrated in the same platform, using a network-on-chip for interconnection. We finally present the evaluation of the system, showing results that demonstrate its correct functionality under temporally constrained environments. A discussion on the implementation and results is also provided.


Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science of the Technical University of Denmark (DTU Compute) in fulfillment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

During my M.Sc. studies at DTU, I have followed the study lines of 'Embedded and Distributed Systems' and 'Digital Systems'. In the courses related to digital systems and computer architecture, such as 'Design of Digital Systems', 'Design of Asynchronous Circuits', 'Computer Architecture and Engineering' and 'Advanced Computer Architecture', is where I have acquired most of the knowledge on topics related to this thesis. Specially on the last one, where we designed the initial version of the audio interface for Patmos. Another DTU course that is related to this thesis is 'Audio Information Processing Systems', which I took as an elective course due to my great interest in digital audio systems, and has provided me with valuable knowledge in algorithms for audio signal processing.

This thesis presents and discusses the design and implementation of the real-time multicore audio processing platform. The report is structured with an ever increasing level of detail. First of all, an overview of the digital audio processing algorithms and the T-CREST platform is given, which is the one used for the implementation. After that, the improvements done to the audio interface are presented. The design and implementation are described next: first, the audio effects are treated individually, and then solutions are proposed for the integration and synchronization of multiple effects in the multi-processor platform. Afterwards, various aspects of the work are evaluated, showing numerical results to prove the correct functionality of the system in different ways. Finally, the results are discussed and the thesis is concluded.

Lyngby, 22-January-2017

A handwritten signature in blue ink, appearing to be 'Daniel Sanz Ausin', written in a cursive style.

Daniel Sanz Ausin

Acknowledgements

I would like to begin by thanking my supervisor, Martin Schoeberl, for accepting this project as my M.Sc. thesis, because it is a topic that I have a great interest in, and also for the help I have received from him during these months. I would also like to thank my co-supervisor, Luca Pezzarossa, for the constant help and feedback during the development of the project, and for the great ideas and discussions we have had on the topic. I would like to further thank all the members of the T-CREST group at the Technical University of Denmark, and specially Fabian Goerge, with whom I designed the first version of the audio interface for Patmos in the Advanced Computer Architecture course at DTU. Additionally, special thanks go to all my friends in Denmark for their help and the time spent together, and finally to my family, Maria Jesus, Jose Luis and Markel, who have been very supportive during the two years of my M.Sc.

Contents

Abstract	i
Preface	iii
Acknowledgements	v
1 Introduction	1
1.1 Multicore Platforms for Audio Processing	2
1.2 Network-on-Chip Based Multicore Platforms	2
1.3 Real-Time Audio Processing	3
1.4 Source Access	4
1.5 Thesis Outline	4
2 Digital Audio Signal Processing Algorithms	7
2.1 Fundamentals of Digital Audio	7
2.2 Digital Audio Effects	10
2.2.1 Classification	10
2.2.2 Filters and Delays	11
2.2.2.1 Basic EQ Filters	12
2.2.2.2 Comb Filters	15
2.2.3 Modulation Effects	16
2.2.3.1 Amplitude Modulation - Tremolo	16
2.2.3.2 Frequency Modulation - Vibrato	17
2.2.3.3 Time-Varying Filters	17
2.2.4 Non-Linear Processing Effects	19
2.2.4.1 Overdrive	20
2.2.4.2 Distortion	21
2.2.5 Spatial Effects	23
2.2.5.1 Reverberation	23

2.2.6	Connections between Effects	24
2.3	Architecture of DSP Processors	26
3	T-CREST Background	29
3.1	Overview of the T-CREST Platform	29
3.2	The Patmos Processor	31
3.2.1	Architecture	31
3.2.2	Memory System and I/O Devices	32
3.3	Compiler and Time-Analysis Tools	34
3.4	Argo NoC	35
3.4.1	TDM Scheduling	35
3.4.2	Network Interface	36
3.4.3	Reconfiguration	39
4	Architecture of the Audio Interface for Patmos	41
4.1	WM8731 Audio CODEC	41
4.2	Design of the Audio Interface	42
4.3	API of the Audio Interface	46
5	Design and Implementation of Audio Effects on Patmos	47
5.1	General Requirements for Real-Time Audio Processing	48
5.1.1	Communication Paradigms	49
5.1.2	Signal Latency	51
5.2	Fixed-Point v.s. Floating-Point Audio Processing	52
5.3	Object-Oriented Style Approach for Audio Effects Processing	54
5.4	Implemented Audio Effects	55
5.4.1	Tremolo	56
5.4.2	Vibrato	58
5.4.3	IIR Filters	58
5.4.4	Delay	59
5.4.5	Wah-Wah	60
5.4.6	Chorus	60
5.4.7	Overdrive	61
5.4.8	Distortion	61
6	Design of Multicore Audio Processing Platform	63
6.1	Static Task Allocation	64
6.2	Message Passing NoC v.s. Shared Memory Communication	66
6.3	Architecture and Synchronization of the Multicore Audio Processing Platform	67
6.3.1	Synchronous Data Flow	68
6.3.2	Rules for Multicore Audio Synchronization	69
6.3.2.1	Single Effect Cores	72
6.3.2.2	Multiple Independent Effect Cores	76

6.3.2.3	Multiple Effect-Chain cores	78
6.3.3	Reducing Send/Receive Overhead	79
6.4	Message Passing NoC Parameters	80
6.5	Audio Processing Latency on a Multicore Platform	81
6.5.1	Latency Added by the Effect Buffers	82
6.5.2	Latency Added by the NoC	83
7	Implementation and WCET Analysis of the Platform	87
7.1	Architecture and Technical Details	88
7.1.1	Master Core Latency	88
7.1.2	Architecture of the Implementation	89
7.1.2.1	The <code>AudioFX</code> Structure	90
7.1.2.2	The <code>alloc_audio_vars</code> Function	92
7.1.2.3	The <code>audio_process</code> Function	93
7.2	WCET Analysis and Static Effect Allocation	94
7.2.1	WCET Analysis and Execution Time Measurements	94
7.2.2	Static Effect Allocator	98
7.2.3	Main Audio Program	100
8	Evaluation and Discussion	101
8.1	Evaluation	101
8.1.1	Evaluation of the Task Allocation Algorithm	102
8.1.2	Evaluation of the Audio Processing Algorithms	103
8.1.3	Evaluation of the Synchronization of the System	104
8.1.4	Evaluation of Parallel Audio Effect Chains	108
8.2	Discussion	109
8.2.1	General Discussion	110
8.2.2	Task Allocation and Scheduling	111
9	Conclusion	113
9.1	Contributions and Results	113
9.2	Future work	114
	Bibliography	117
A	Audio Interface: Hardware Design & API	121
A.1	ADC Buffer	121
A.2	DAC Buffer	125
A.3	API Functions	130
B	Examples of some Audio Effects	135
B.1	Filter	135
B.2	Delay	137
B.3	WahWah	139
B.4	Overdrive	141

C	Audio Processing Function in the Multicore Platform	143
C.1	audio_process function	143
C.2	audioint.h	150
C.3	NoC Schedule XML file	151

CHAPTER 1

Introduction

This chapter introduces the work presented in this thesis. It provides an overview on the main topics that are related to the project and presents the outline of the thesis.

Multicore platforms are becoming more and more common for audio processing applications, due to the improvement in computation performance that they provide. Some examples of this are audio software environments that run on multi-processor computers, or embedded audio multicore Digital Signal Processors (DSP) the are found in many applications, such as hearing aids or portable mobile devices. In this work, we focus in real-time audio processing applications, which means that the processing must be applied within an interval of time that ensures that the delay of the signal is imperceptible for the human ear. This requires that the temporal behavior of the processing platform must be completely predictable in order to provide time guarantees.

The work presented in this thesis is addressed to network-on-chip based multicore platforms for real-time systems. An example of this is the T-CREST platform, which is under continuous development by the Technical University of Denmark. This is the platform chosen for the implementation of the audio processing system, currently running on an Altera DE2-115 FPGA board.

1.1 Multicore Platforms for Audio Processing

Multicore platforms appear to be a feasible way to increase computational power in many applications, due to the heat dissipation and clock rate limitations of single core processors. Multi-Processor System-On-Chips (MP-SoC) enable the integration of various Intellectual Property (IP) cores on the same chip. These IPs could be conventional processors, DSPs or hardware accelerators, as well as I/O devices. Some of these IP cores, such as DSPs or Graphics Processing Units (GPU), are very common in audio processing systems, because they provide a considerable speed-up in the typical operations required in audio computation, such as memory access instructions or arithmetic operations [1].

As the amount of computational resources in the system increases, more parallelism is available, which can be exploited by performing concurrent processing operations. Audio signals are processed in the digital domain as a stream of samples. In many cases, algorithms have sequential dependencies, which limit the amount of concurrent operations that can be performed. However, there are many ways to take advantage of the parallelism provided by multi-processor platforms.

One possible way to exploit this parallelism is to distribute the processing of an algorithm with high computational requirements into threads that can be concurrently executed on different cores. Another possible way is to use many processors to compute individual algorithms simultaneously, which is exactly what has been done in the work presented in this thesis. The individual processing algorithms correspond to audio effects that are very common in music applications, such as filters, delay lines, modulation effects or waveshaping techniques. These effects are connected to each other forming sequential or parallel chains, and the processing is distributed among the computational resources available in the platform.

1.2 Network-on-Chip Based Multicore Platforms

One of the main challenges of multi-processor systems is to accomplish optimal interconnection between the components in the platform. In some cases, the interconnection element can decrease the performance of the platform considerably. Traditionally, a shared bus has been used for communication by all the components in the system, which could represent a bottleneck when the communication requirements are high, due to the limited bandwidth and concurrency.

To overcome these restrictions, Networks-on-Chips (NoC) are used, which offer flexibility and parallelism in intercommunication, as individual channels are available between IP cores, depending on the requirements of the application. An example of a multicore platform based on a NoC is shown in Figure 1.1. Here, its basic components are shown, which are the Network Interfaces (NI), routers (R) and links. Packets of data can be transferred between cores through the NoC. The NIs allow the IP cores to send and receive data through the NoC. The routers exchange data between them through the links, depending on the path of packets from source to destination.

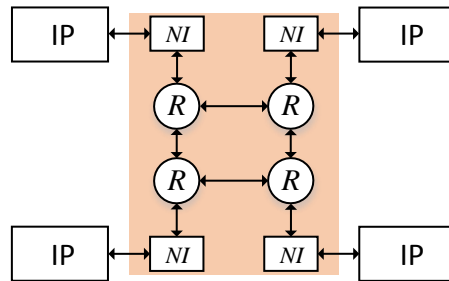


Figure 1.1: Overview of a multicore platform with a set of IP cores, which exchange data using a NoC (shown with a colored background). The NIs and the routers are the components of the NoC, together with the links between routers.

The usage of the NoC is essential in the implemented audio processing architecture, as the communication requirements of the system rely strictly on this component to achieve real-time processing.

1.3 Real-Time Audio Processing

Some audio applications use off-line processing: in this case, the full audio signal to be processed has been previously stored in some kind of memory system, and there are no strict requirements of the time it takes to process. This is not the case in real-time audio applications, where processing is done immediately as the stream of samples is input into the system, and the resulting stream must be output within a time interval that is perceived as instantaneous by the human ear. Some possible examples of real-time audio systems are hearing aids, digital audio communication systems such as streaming applications, or music effects. The presented work focuses on the latter.

In order to provide real-time guarantees, the system must have a predictable

temporal behavior. In this sense, the concept of Worst-Case Execution Time (WCET) becomes crucial, which is the maximum possible time taken for a task to execute. The platform used for processing must be designed in a way that WCET is predictable and within an acceptable interval of time. The multicore platform used here, T-CREST, is optimized for hard real-time systems, as it provides resources and tools for analysis and reduction of WCET.

1.4 Source Access

The full code related to this project can be found in the T-CREST¹ collection of GitHub repositories. In particular, the code is distributed in the Patmos² and Aegean³ repositories.

In the first one, an audio library, *libaudio*⁴, is found, which contains the C source code related to the audio effects. This library also contains a *README* file, which explains how to run audio applications on the FPGA board.

In the second one, a folder containing descriptions of some example audio applications is found, called *audio_apps*⁵. A *README* file is also found here, where the steps required to run these example applications in the board are explained.

1.5 Thesis Outline

The work presented in this thesis is the design, implementation and evaluation of a real-time multicore audio processing platform, based on a Network-on-Chip. For this, a set of audio effects has been implemented following conventional algorithms. The effects are then merged together in the multicore platform, forming chains of effects that are connected to each other. The thesis is structured as follows:

- Chapter 2 introduces some fundamental concepts of digital audio, and presents the main DSP algorithms for audio processing used in this project.

¹<https://github.com/t-crest>

²<https://github.com/t-crest/patmos>

³<https://github.com/t-crest/aegean>

⁴<https://github.com/t-crest/patmos/tree/master/c/libaudio>

⁵https://github.com/t-crest/aegean/tree/master/audio_apps

-
- Chapter 3 presents the T-CREST platform and overviews its tools and components, focusing on the most relevant ones for this work.
 - Chapter 4 recalls the audio interface for the Patmos processor that was previously designed, and explains the improvements done with the addition of input/output buffers.
 - Chapter 5 presents the implementation of the individual audio effects in the Patmos processor, discussing the main design considerations.
 - Chapter 6 explains the rules designed and followed in this project for the correct synchronization of multiple audio effects, which are mapped to different cores and form audio effect chains.
 - Chapter 7 describes the implementation of the multicore audio processing platform on T-CREST.
 - Chapter 8 verifies the different parts of the implementation, showing numerical results. It also provides discussion on some aspects of the system.
 - Chapter 9 concludes the thesis.
 - Appendices A, B and C contain some code listings related the audio interface (Chapter 4), the individual effects (Chapter 5) and the multicore implementation and evaluation (Chapters 7 and 8) respectively.

CHAPTER 2

Digital Audio Signal Processing Algorithms

This chapter provides background about the digital signal processing algorithms that have been used in this project to implement the audio processing effects. Section 2.1 briefly introduces the fundamentals of digital audio signal processing, and its most important parameters are explained. After that, Section 2.2 classifies and explains the algorithms used to create audio effects, showing signal-flow graphs for a better understanding. Finally, Section 2.3 presents the architecture of common DSP processors.

2.1 Fundamentals of Digital Audio

Sound can be described as a variation of pressure that propagates as a mechanical wave through a medium, typically air. Humans perceive these vibrations on their ears, and can hear them if the oscillation frequency is between 20 Hz and 20 kHz approximately. Sound waves are referred to as acoustic signals in the mechanical domain. Sound pressure level is typically measured in a logarithmic scale using the Decibel (dB) unit, considering a reference pressure level which is usually $20\ \mu P$ on air. This value is known to be the lower audible threshold

of the human ear. This is shown in Equation 2.1.

$$L_p = 20 \log_{10} \frac{p}{p_{ref}} \quad [dB] \quad (2.1)$$

In the electrical domain, however, sound waves are called audio signals. Therefore, a digital audio signal can be defined as a representation of sound in the digital domain.

The components that can be found in digital audio systems are the following:

- Acoustic-to-electric transducer, e.g. a microphone
- Analog-to-digital converter (ADC)
- Digital audio signal processing system
- Digital-to-analog converter (DAC)
- Electric-to-acoustic transducer, e.g. a loudspeaker

Not all audio systems need to contain all the parts mentioned above: for instance, a digital synthesizer might only contain the last 3 parts mentioned: a digital audio system which creates the sound, a DAC and a loudspeaker. Alternatively, a digital audio recorder will only contain the first 3 parts mentioned: a microphone, an ADC and a processing system to store the audio signal in a memory.

In order to treat signals in the digital domain, they need to be sampled. Two of the most important parameters of digital audio are the sampling frequency and the resolution.

- The **sampling frequency** sets the amount of audio samples used per second, represented in Hertz (Hz). In order for the audio signal to be represented correctly, the sampling frequency needs to satisfy the Nyquist theorem [2, Chapter 2.5], which specifies the minimum sampling frequency as double the bandwidth of the signal. As explained before, the maximum frequency of audio signals is 20 kHz , therefore the Nyquist frequency is 40 kHz . Standard sampling frequency values found in the industry are 44.1 kHz or 48 kHz , and the latter is used in this project. Some higher quality systems use values up to 192 kHz .

- The **resolution** specifies the amount of bits used to represent each sample. Depending on the resolution value, **quantization** might need to be done, which is the process of mapping each audio sample to the closest value that can be represented in a given resolution. The higher the resolution, the less quantization error when converting the signal from analog to digital. A standard value is 16-bit resolution, which is used in this project. If higher quality is needed, 24-bit or 32-bit resolutions can be used. The resolution is also directly related to the dynamic range of the digital audio signal, which will increase with a higher resolution value.

These two parameters are very important as they are directly related to the quality of the audio signal. Too low sampling frequencies will result in a loss of information contained in the higher frequencies of the audio spectrum. Low resolution values will lead to bigger quantization errors and will introduce audible noise. A clear example of this are old video game sounds, which used 8-bit to 12-bit audio. On the other hand, high sampling frequency or resolution values will improve the signal quality, with the drawback of needing more storage space and higher processing power.

Sampling frequency and resolution are also important parameters for the ADC and DAC, as they will be more complex and expensive if they need to operate at high values.

It is important to remind that most audio processing systems are stereo, which means they have left and right input and output channels. The system implemented in this project is also an stereo audio processor: that is why, in this document, when an audio sample is mentioned, it actually corresponds to two 16-bit samples, for the left and right channels.

The power of an audio signal in the digital domain is measured in a logarithmic scale as well, but the equation is different to the one used for mechanical sound pressure level: the difference is that the reference value is not the lower threshold of the audible range, but it is the maximum value that can be represented in the digital domain for a given resolution. The unit for this measurement is called *dBFS* (Decibels relative to Full Scale). For example, for a given resolution of n -bits, the audio signal can be have a maximum value of 2^{n-1} (for a signed signal). For the correct representation, the values must always be kept under this limit, which corresponds to 0 *dBFS* (so all values must be negative). Therefore, for a given sample i of an audio signal x , the amplitude level can be defined as shown in Equation 2.2.

$$L_{FS}(i) = 20 \log_{10} \frac{x_i}{2^{n-1}} \quad [dBFS] \quad (2.2)$$

The equation above gives the peak (instantaneous) value. However, it is very common to use RMS amplitude values instead, which are calculated over a window of samples, and give a much more realistic value of the loudness of an audio signal.

Finally, another parameter which acquires a great importance in real-time audio signal processing is the **latency**, which can be defined as the time measured between the instant when an audio sample is input to the system, and the point in time when it is output. The latency can be measured either in time units (usually *ms*) or in samples, for a given sampling frequency. In real-time audio, it is extremely important to keep this value within a certain time interval so that the output audio signal can be perceived as instantaneous at all times. This topic is further discussed in Subsection 5.1.2, and an estimation of a tolerable latency interval is given.

2.2 Digital Audio Effects

In this section, some digital audio signal processing algorithms are classified and explained. A high level of abstraction is used to describe them (not any specific software or programming language). The algorithms explained here are important because they provide a theoretical overview of the digital audio effects implemented in this project, which will be presented in Chapter 5. Most of the algorithms used follow different chapters of [3], so the reference to each corresponding chapter is provided in the beginning of each subsection. Subsection 2.2.6 comes at the end of this section, showing how the audio effects can be connected to each other.

2.2.1 Classification

Audio effects can be classified in many ways [3, Chapter 1]: for example, a perceptual classification can be used to describe how humans hear them in terms of rhythm, pitch, loudness, etc. In this project, however, a technical classification is more suitable, depending on the algorithms used for the implementation. This classification results in many different groups, but only some of them are used in this project. They are the following:

- **Filters and delays**
- **Modulation effects**

- **Non-linear processing effects**
- **Spatial effects**

The audio effects that belong to the mentioned groups are explained in Subsections 2.2.2 to 2.2.5.

2.2.2 Filters and Delays

Filter structures are very widely used in digital signal processing [3, Chapter 2]. They are also referred to as delay structures, because delayed samples of data are used for calculations. Two of the most common digital filter structures used are Finite Impulse-Response (FIR) and Infinite Impulse-Response (IIR) filters.

- **FIR** filters have a finite impulse response duration, because the output sequence is the result of a weighted sum of the last input samples ($N + 1$ samples for an N order filter). Each one of the samples is multiplied with the weighting filter coefficient b_i , as shown in Equation 2.3.

$$y(n) = \sum_{i=0}^N b_i \cdot x(n - i) \quad (2.3)$$

- **IIR** filters have an infinite impulse response duration, because the output sequence is the result of a weighted sum of both the last $N + 1$ input and N output samples, which results in feedback loops. The filter coefficients are b for the input samples and a for the output samples. Equation 2.4 shows this.

$$y(n) = \frac{1}{a_0} \left(\sum_{i=0}^N b_i \cdot x(n - i) - \sum_{j=1}^N a_j \cdot y(n - j) \right) \quad (2.4)$$

In this project, FIR filters have not been used at all. The reason is that, usually, much higher order FIR filters are needed to achieve similar audio effects than if IIR filters are used. This results in higher memory requirements and a longer computation time, which is an important drawback for real-time audio processing. That is why, in general, for digital audio processing, IIR filters are much more common than FIR filters. However, FIR filters become very useful

for audio applications when implemented as convolution filters, using the Short Time Fourier Transform (STFT) algorithm. This kind of digital filter is not used in this project, due to the processing limitations of the platform.

The IIR filter structure has been used to create many audio effects. Figure 2.1 shows the structure of a 2nd order IIR filter. As it can be appreciated, 5 multiplications need to be performed ($a_0 = 1$), and 4 audio samples need to be stored in memory (2 input, 2 output). This structure is the base of some of the effects explained in the following sections.

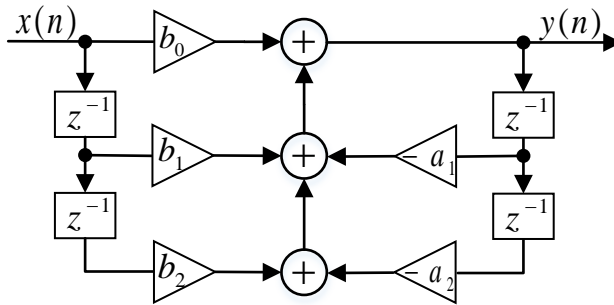


Figure 2.1: 2nd order IIR filter structure.

There are some more advanced digital implementations of IIR filters, such as the parallel second-order form, shown in [4], which reduces the quantization noise. This is useful when high filter orders are used, which is not the case of this project. That is why, here, the traditional direct form has been used, shown in Figure 2.1.

2.2.2.1 Basic EQ Filters

Equalization filters are very widely used in audio signal processing. They affect the frequency spectrum of the signal, removing some frequency components and possibly heightening others. The most common audio filters found, both on analog or digital domain, are the following:

- **Low-Pass (LP)** filters remove the higher frequency components of the signal. The most important parameters are the cut-off frequency (f_c) and the resonance or Q quality factor, which specifies the filter gain on the cut-off frequency.
- **High-Pass (HP)** filters remove the lower frequencies of the signal. The most important parameters are the same as for the low-pass filters.

- **Band-Pass (BP)** filters let only the frequencies between a lower and an upper limit to go through. These limits are given by the central or cut-off frequency (f_c) and the filter bandwidth (f_b).
- **Band-Reject (BR)** filters do exactly the opposite as the band-pass filters, removing the frequency components between the lower and upper limits.

Another very important parameter that is common for all filters is the filter order, which specifies the slope of the filter: in other words, how fast the gain decays outside the filter cut-off limits.

There are some other filter types that are also used for audio equalization, such as the shelving filters, but they have not been implemented in this project.

Although there are many different possibilities to implement the LP/HP/BP/BR filters in the digital domain, the chosen one uses the 2nd order IIR filter structure shown in Figure 2.1. The 2nd order IIR structure leads to the transfer function presented in Equation 2.5.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (2.5)$$

Modifying the values of the filter coefficients, certain frequencies can be attenuated. The coefficient values for the LP and HP filters are calculated from the desired f_c and Q parameters, following the equations shown in Table 2.1. The K parameter is proportional to the desired f_c value relative to the sampling frequency used, f_s , as shown in Equation 2.6.

$$K = \tan\left(\pi \frac{f_c}{f_s}\right) \quad (2.6)$$

The BP and BR filters can be implemented in many different ways using IIR filters. The implementation chosen in this project is based on a 2nd order IIR all-pass filter structure, which is given by the following transfer function presented in Equation 2.7.

$$A(z) = \frac{-c + d(1 - c)z^{-1} + z^{-2}}{1 + d(1 - c)z^{-1} - cz^{-2}} \quad (2.7)$$

	b_0	b_1	b_2	a_1	a_2
Low-pass	$\frac{K^2 Q}{K^2 Q + K + Q}$	$\frac{2K^2 Q}{K^2 Q + K + Q}$	$\frac{K^2 Q}{K^2 Q + K + Q}$	$\frac{2Q \cdot (K^2 - 1)}{K^2 Q + K + Q}$	$\frac{K^2 Q - K + Q}{K^2 Q + K + Q}$
High-pass	$\frac{Q}{K^2 Q + K + Q}$	$-\frac{2Q}{K^2 Q + K + Q}$	$\frac{Q}{K^2 Q + K + Q}$	$\frac{2Q \cdot (K^2 - 1)}{K^2 Q + K + Q}$	$\frac{K^2 Q - K + Q}{K^2 Q + K + Q}$

Table 2.1: 2nd order IIR Filter coefficients for low-pass and high-pass filters.

Filter parameters c and d are calculated from the desired values f_c and f_b , given the Equations 2.8 and 2.9.

$$c = \frac{\tan(\pi \frac{f_b}{f_s}) - 1}{\tan(\pi \frac{f_b}{f_s}) + 1} \quad (2.8)$$

$$d = -\cos(\pi \frac{f_c}{f_s}) \quad (2.9)$$

The all-pass filter equation shown does not affect the magnitude of the signal, but it affects the phase for different frequencies. When combining the all-pass filtered signal with the original input signal, BP and BR filters are achieved because the phase shift of the all-pass filtered signal attenuates or cancels some frequencies depending on the phase delay. This combination of the signals is shown in Figure 2.2, where the BP or BR filtering is determined by the sign of the combination.

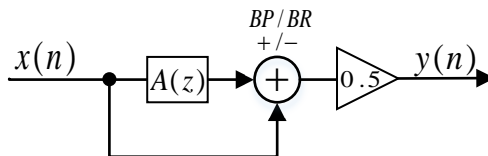


Figure 2.2: Band-Pass/Band-Reject filter structures using 2nd order IIR all-pass filter (sign indicates type of filter).

2.2.2.2 Comb Filters

Comb filters are also known as basic delay filters, because the input signal is combined with a delayed copy of it. The main difference between comb filters and IIR/FIR filters is that, in the latter, the order N indicates how many of the the last samples of the signal were combined, whereas in the former, the order indicates how many delayed copies of the signal are combined, and the delay is usually greater than a single sample. The computation of these filters is rather simple, but they usually require more storage space, proportional to the chosen delay length. The name comb filter refers to its transfer function, which in the frequency spectrum looks like a comb because certain frequencies are attenuated, which depend on the delay length.

Comb filters can be classified as FIR or IIR as well, depending on whether the delayed signal is the input or the output. These two filter structures are shown in Figures 2.3 and 2.4. These filters will change the timbre of the audio signal if the chosen delay is smaller than 50 ms (this value corresponds to the lowest audible frequency, 20 Hz). If instead, the delay is greater, the effect of the comb filter will be perceived as an echo. In the FIR filter, a single copy of the input signal will be heard, while in the IIR, multiple copies will be repeated due to the feedback loop.

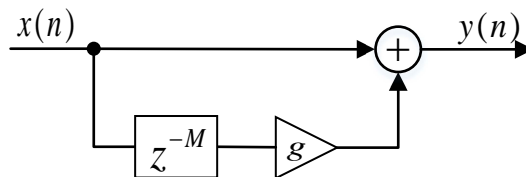


Figure 2.3: FIR comb filter structure.

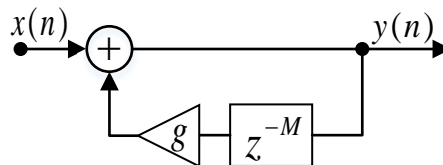


Figure 2.4: IIR comb filter structure.

An important consequence of processing sound using the presented filter structures is that the dynamic range of the signal gets altered. A simple way to demonstrate this is by analyzing Equation 2.3, for a 2nd order FIR filter. If all the coefficients, b_0 , b_1 and b_2 are 1, then the dynamic range of the output

signal can be up to 3 times that of the input signal. This is totally undesired because the signal cannot represent values outside its dynamic range (values over 0 *dBFS*), so overflow/underflow situations might happen, which would corrupt the output signal. To avoid this, normalization techniques are used to bring the output signal to an acceptable range. There are many different ways to do this: the most simple one is to reduce the amplitude of the output signal, which will also cause a change in loudness. Another simple normalization method is to saturate or clip the signal if it goes beyond the upper limit, but this can cause distortion. In general, dynamic range modification is something that happens in all kinds of signal processing, not only in the digital domain, and advanced methods have been developed to take care of this, which are outside the scope of this project.

2.2.3 Modulation Effects

Modulation is the temporal variation of certain parameters of a signal, which is called the carrier [3, Chapters 2, 3]. This process is commonly used in telecommunications, where the information to be transmitted is usually contained by the modulating signal.

In audio signal processing, however, modulation is used with a completely different purpose: the objective is to enhance certain properties of the carrier by adding some temporal variations to achieve different effects. The most common parameters to be modulated are the amplitude, the frequency or the phase of a signal, but more complex parameters can also be modified, as will be shown here. Depending on the modulating parameter and the properties of the modulation signal, many different audio effects can be achieved.

2.2.3.1 Amplitude Modulation - Tremolo

This is probably the most simple and straightforward modulation effect used in audio. The amplitude of the carrier signal is modulated using a Low Frequency Oscillator (LFO), which is perceived as a periodical change in the signal volume. LFO signals have a fundamental frequency that is under the audio range (lower than 20 *Hz*). If higher frequency signals are used, the amplitude modulation is perceived as a change in the timbre of the sound. The modulation signal is usually a sinusoid, but different shapes might also be used.

This effect works specially well when long duration notes or chords are played. It is usually found as a guitar effect, but can also be used in other instruments.

Figure 2.5 shows the signal flow of the tremolo effect, where an external LFO signal generator is required.

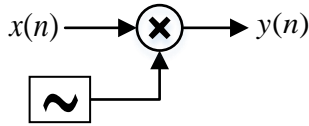


Figure 2.5: Tremolo effect.

2.2.3.2 Frequency Modulation - Vibrato

If, instead of the amplitude, the frequency of the signal is modified using an LFO, the resulting effect is called vibrato. It is perceived as a periodical change of the pitch of a signal, usually as a sinusoid. Again, if the frequency is within the audio range, the effect will affect the timbre of the original signal. This is called a ring modulator.

It is common to find vibrato effect pedals for guitars or synthesizers. In the digital domain, this effect can be implemented using a small sample delay array with the size of the modulation amplitude in samples, M in this case. The output sample index is determined by the modulation signal, which oscillates around $M/2$ with an amplitude of $M/2$ and with the desired frequency. The vibrato effect is shown in Figure 2.6, where the diagonal arrow indicates the sample index modulation.

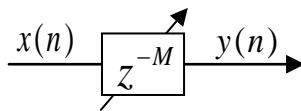


Figure 2.6: Vibrato effect.

2.2.3.3 Time-Varying Filters

Time-varying filters are the result of applying modulation to the filter effects shown in Subsection 2.2.2.1. Many different effects can be achieved doing this, and some of them have been implemented in this project. The parameters that are modulated are the filter coefficients for the IIR filters, and the delay sample index for the comb filters (done in a similar way as in the vibrato effect).

Two well-known effects that can be achieved by modulating the IIR filter coefficients are the wah-wah and the phaser. The first one is implemented as a time-varying band-pass filter. The central frequency (and possibly also the bandwidth) is modulated with a LFO signal, which results in time-varying filter coefficients. Usually, the wah-wah effect is used in the electric guitar, where the player can move the band-pass frequency using an expression foot pedal. However, a similar effect (also known as auto-wah) can be achieved if a LFO signal is the modulation source. Figure 2.7 shows this effect, where the filtered signal is combined with the original signal. g indicates the effect gain.

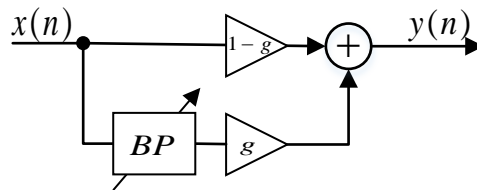


Figure 2.7: Wah-wah effect.

The phaser effect is implemented in a similar way to the wah-wah, but, in this case, the filter used is a band-reject filter (usually a series of filters are used, with different modulation parameters). The frequency variation of the band-reject filter causes different phases to be canceled, thus the name of the effect. The phaser is shown in Figure 2.8.

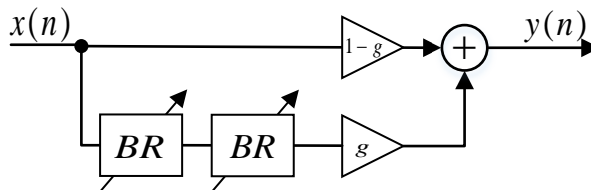


Figure 2.8: Phaser effect.

The last time-varying filter implemented in this project is the chorus, which can be achieved as a 2nd order time-varying FIR comb filter. Each one of the cascaded channels has a different delay length, and two LFO signals are used to determine the sample index of each channel, in the same way as in the vibrato.

The goal of this effect is to increment the amount of sound sources to simulate the behavior of many different musicians playing the same audio piece: these musicians will always be slightly unsynchronized in time and pitch, and this is emulated using delays with frequency modulations. The modulation signal

for the cascaded channels of the chorus can be a sinusoid, but sometimes some other sources are used, such as low frequency noise. In general, the choice of the modulation signal type and its parameters are a whole research topic itself to achieve the musically most pleasing results, but this is outside the scope of this project, so simple sinusoidal LFOs have been used here, which give satisfactory results. The chorus effect is shown in Figure 2.9.

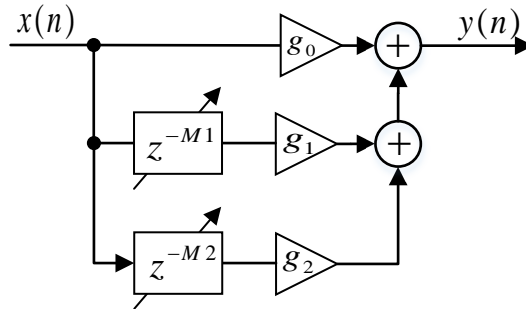


Figure 2.9: Chorus effect.

There are some other effects that are implemented as time-varying comb filters, some of which are the resonator, the slapback and the flanger. These are implemented in a similar way as the chorus effect, but with different delays and modulation sources.

2.2.4 Non-Linear Processing Effects

Most digital signal processing is mainly based on linear systems, such as filter structures as the ones explained in Subsection 2.2.2. However, lots of the analog audio gear used for music or other applications have non-linearities, which give a special character to sound [3, Chapter 4]. This gear includes valve amplifiers, tape recorders, analog mixers, distortion pedals, loudspeakers, and so on.

During the last decades, analog systems have been used by many musicians, performers, producers and sound engineers to enhance the audio signals in a non-linear way. These non-linearities are caused by the imperfections of analog components, but that does not mean that any analog component will improve the quality of sound or add some color to it: in fact, these components are chosen carefully by the engineers who develop these products, and lots of experience and knowledge is required to achieve musically pleasant results.

In digital audio signal processing, the non-linear behavior of the mentioned sys-

tems is emulated. To achieve results that are similar to analog components, these ones need to be modeled very precise and carefully, which requires high computational power. However, in most cases, more simple digital approximations are done which achieve acceptable results (this is a topic that generates discussion among experts). Finding a good balance between high-quality non-linear processing and minimizing computational requirements is not an easy task. A lot of listening and recording experience is required to adjust the non-linear parameters of a particular system: this is an art form itself that is outside the scope of this work. That is why, here, simple non-linear processing algorithms have been used.

Non-linear processing is also known as waveshaping, because the shape of the audio waveform is altered, thus modifying its frequency components as well. This means that high frequencies are added and harmonic distortion is introduced, which changes the character of the sound, possibly enhancing it or even destroying it completely.

One of the widely used non-linear effects is the dynamic range compression, where the amplitude level of the signal is measured (usually the RMS value of a certain window is taken), and the signal is attenuated if a threshold is exceeded. This is specially useful when mixing several audio signals in order to 'glue' them together (reducing the differences between the loudest and the softest parts). The waveshaping effect is clear, as only the peaks of the audio signals are modified and the softer parts remain unaffected.

Although compression has not been implemented in this project, there are some other non-linear effects that have, such as overdrive and distortion.

2.2.4.1 Overdrive

Overdrive is a mixture of linear and non-linear processing, because the signal gets linearly affected in the lower amplitude parts and is overdriven in the louder parts, with a smooth transition between these two regions. The aim is to give a warm and colorful characteristic to the sound in its loudest regions. This effect tries to emulate the behavior of the analog components in valve amplifiers, tape recorders, effects, and so on, where the signal gets slightly distorted on higher levels, resulting in a warm overdrive sound.

The implementation of overdrive used here is defined by 3 regions, depending on the amplitude of the input signal. The first 1/3 of the amplitude is the linear zone, where the output is equal to double the input. Between 1/3 and 2/3 of the amplitude, non-linear processing is applied. Finally, between 2/3 and 1, the

signal is simply clipped. This is shown in Equation 2.10.

$$f(x) = \begin{cases} 2x & \text{if } 0 \leq x \leq 1/3 \\ \frac{3-(2-3x)^2}{3} & \text{if } 1/3 \leq x \leq 2/3 \\ 1 & \text{if } 2/3 \leq x \leq 1 \end{cases} \quad (2.10)$$

This effect is shown in Figure 2.10, where the output values are shown as a function of the input. The linear and non-linear areas can be clearly distinguished.

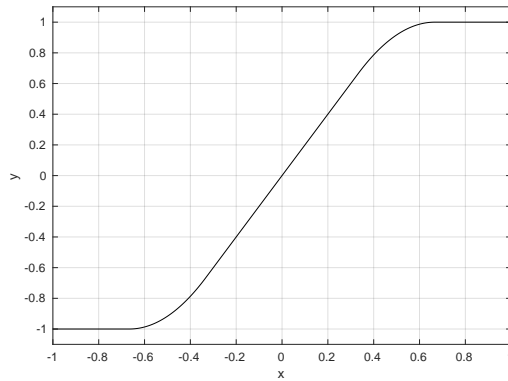


Figure 2.10: Overdrive effect: output signal y as a function of input x .

2.2.4.2 Distortion

This effect operates fully in the non-linear region, and the aim of it is to change the timbre of the input signal by adding strong harmonics to achieve a 'harder' sound. Distortion is a main characteristic of sound that has defined many new genres such as rock, punk or metal, and has changed the way an instrument like the electric guitar is approached. Another term that is also widely used is *fuzz*, which usually refers to an even harder distortion sound.

A very common way to emulate the behavior of distortion pedals in the digital domain is by the exponential function given in Equation 2.11.

$$f(x) = \text{sgn}(x) (1 - e^{-\alpha|x|}) \quad (2.11)$$

The α parameter sets the gain. In this project, this function has only been implemented with a gain of $\alpha = 1$ using the MacLaurin series. This creates a very soft distortion effect. Having a gain different than 1 makes the MacLaurin series diverge, so the implementation for a real-time system gets complex. That is why, in order to have a harder distortion, another function has been used from [5], where the distortion amount a is defined, then the parameter K is calculated as shown in Equation 2.12.

$$K = \frac{2a}{1 - a} \quad (2.12)$$

Then the distortion function depends just on K , and is calculated as in Equation 2.13.

$$f(x) = \frac{(1 + K) \cdot x}{1 + (K|x|)} \quad (2.13)$$

The distortion output as a function of the input is shown in Figure 2.11, where it is shown how non-linear processing is applied on the whole dynamic spectrum of the input signal, and how the output signal reaches saturation levels much faster than for the overdrive effect.

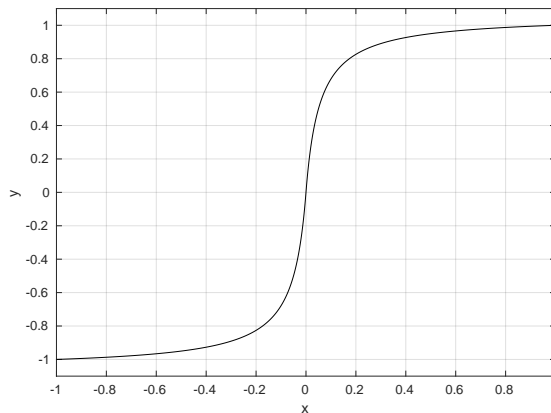


Figure 2.11: Distortion effect: output signal y as a function of input x .

2.2.5 Spatial Effects

The human ear is able to retrieve information from the physical surroundings just by listening to a sound: the position of the sound source can be approximately identified, both by the sound volume (which is inversely proportional to the distance to the source) and by the difference of perception between left and right ears. But not only does sound give information about the source, but also about the surroundings: the sound will be perceived in a completely different way depending on the environment. For example a small room, an open space or a cathedral will have completely different behaviors and the listener can detect this difference.

In order to model the behavior of human hearing in the digital domain, the concepts of *head-related transfer function* and *binaural techniques* are important [3, Chapter 5]. The first one tries to emulate the transfer function of the channel between the sound source and the human ears, which depends on the distance and position between source and receiver, and also on the human head shape. This is done by yielding temporal and spectral differences in each ear. The second concept, binaural techniques, are used to control the sound that is perceived in each ear, and use head-related transfer functions to do this. This effect is easy to perceive when listening to a sound with headphones: one has the impression of being on a physical space and distinguishing the positions of the sound sources.

The two concepts mentioned above are useful for creating digital simulations of physical spaces, and are widely used in audiovisual projects or for TV and cinema (for instance, to create an evolving sound and bring the listener more into the role). This is why these techniques are not so interesting for this project. But there is a very interesting spatial effect that is widely used in audio signal processing for music applications: it is called *reverberation*.

2.2.5.1 Reverberation

Reverberation is created with the reflections of sound in a physical space: these reflections cause the sound to be perceived even when the source is not producing it. Different rooms or spaces produce different reverberations, and can enhance the sound that is being played. This is the case of some auditoriums or theaters, where the sound is enhanced by the room shape. In musical recordings, this effect tries to be emulated by adding analog or digital reverberation to the audio signal to improve its quality.

The reverberation usually contains 3 different parts:

- The **direct sound**, which is what reaches the listener first.
- The **early reflections**, which are perceived as part of the direct sound, changing some characteristics of it.
- The **late reverberation**, which is the tail of the reflected signal and gives an idea of the size of the room.

The reverberation characteristic of a physical space is defined by its *impulse response (IR)*, which models the reflections of the objects and walls. When a 'dry' audio signal (no effect on it) is convolved with an IR, it seems to be played in the physical space defined by the IR. This can be achieved in the digital domain as an FIR filter where the order is as long as the length of the IR (in samples). This implies several thousands of samples (durations up to some seconds), so the FIR convolution becomes unpractical for real-time audio purposes due to the large amount of computations required. However, nowadays this is done using the Short-Time Fourier Transform (STFT), which is computationally cheaper but introduces some delay.

Another implementation of the reverb effect in the digital domain was proposed by J. A. Moorer [6], which is computationally more simple than the IR convolution, and has been used during decades to achieve satisfactory digital emulations of reverberation. Moorer's reverberator [7] is designed as shown in Figure 2.12, where two stages can be distinguished. The first stage corresponds to the mentioned early reflections, and it is implemented as a tap delay line where samples of different delays are added together to model the reflections on the walls. The second stage consists of a bank of parallel IIR comb filters that act as low-pass filters and simulate a smooth decay of the higher frequencies. After that, an all-pass filter is added to increase the density of the echo effect.

The 2nd stage of Moorer's reverb is based on Schroeder's work, who designed this structure that creates a dense impulse response. The all-pass filter used in this second stage is shown in Figure 2.13.

2.2.6 Connections between Effects

It is very common in audio applications to combine many of the presented effects. In music, this technique is widely used by many musicians and sound engineers to apply more than one effect to the audio signal, thus changing the character of the sound in many ways.

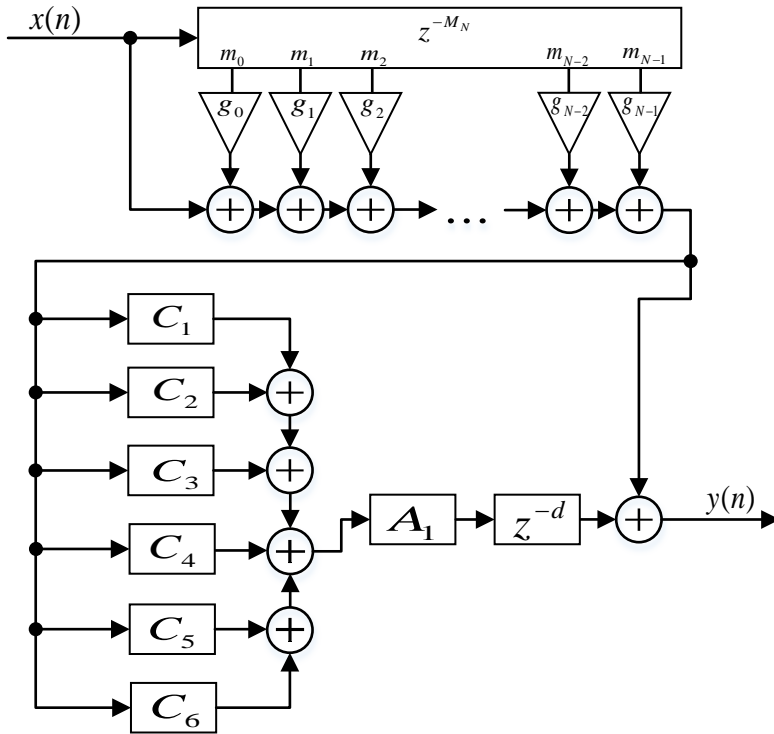


Figure 2.12: Moorer's reverberator.

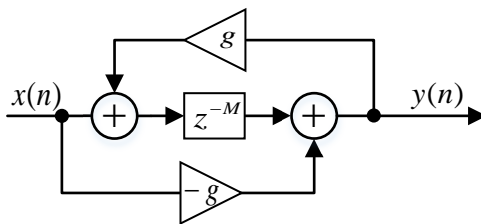


Figure 2.13: All-Pass filter structure used in Moorer's reverberator.

The effects are usually connected sequentially forming chains, where the stream of output samples of one of them is input to the next effect. The audio signal then flows through the effects found in the system. It is also very common to find parallel chains, where the audio signal is split into two or more branches, and separate processing is applied in each one of them. At some point, the signals are merged together again (their samples are added).

The type of effects found in the chain and the order in which they are placed defines the output sound of the system. If the same effects are combined in different orders, the resulting sound might change. A clear example of this could be a chain consisting of a low-pass filter and a distortion effect. If the distortion effect is placed last in the chain, it will create harmonics in higher frequencies. But if the low-pass filter is placed at the end, it will reduce the harmonics previously created by the distortion effect.

Figure 2.14 shows a possible connection between some effects. The first effect found in the chain is the wah-wah, and the signal gets then split into two parallel effects, the delay and the distortion. At the end, the two branches are added together again.

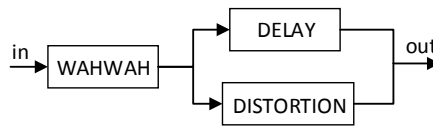


Figure 2.14: Possible setup of effects, forming sequential and parallel chains.

2.3 Architecture of DSP Processors

As it can be inferred from the presented algorithms, the most repeated operations in digital signal processing are the arithmetic addition and multiplication, and the memory access operations to access filter coefficients, sample buffers, modulation signals, and so on. The execution of a DSP algorithm is limited by the amount of these operations required. But obviously it also depends on the device used for computation.

Nowadays, there are many different types of processors optimized for each task. In the audio processing field, it is very common to use powerful DSPs for a wide variety of algorithms. But specialised devices for some tasks can also be found, such as FFT processors to compute convolution reverb. As it is shown in [4] and [8], Graphics Processing Units (GPU) are also widely used nowadays

for audio processing, and can reduce execution time considerably due to their high parallelism in data processing, for instance for high order IIR filtering. However, sometimes the speed-up provided by GPUs might be limited, due to sequential dependencies of audio signals. The work in [9] mentions that higher processing power is achieved when integrating multiple processors into the processing platform. Combinations of different types of processors into the same platform might be an optimal solution to cover a wide range of processing algorithms by distributing tasks.

Leaving some of those specialised processors aside, we focus on general purpose DSP processors now. Some of the main requirements to speed-up computation in these devices are listed here:

- **High memory-access bandwidth:** typical DSP operations, such as FIR filters, IIR filters or FFTs, require moving large groups of samples and coefficients from memory to arithmetic units. On multicore processors, bandwidth is also required to move data between cores. Having large buses allows moving data faster, for instance when high order filters need to be computed.
- **Local program and data memories,** which can be caches or SPMs. DSP algorithms generally spend most time in loops where they execute the same operations. Having local memories means faster access to instructions of the loop and the data needed, such as filter coefficients or multiplication products.
- **High computational power:** the main DSP arithmetic operations are the multiplication and the addition, but logical and bitwise operations are also needed, such as masking, bit-shifting and so on. The more resources available to do these operations in parallel, the faster the execution time. For instance, [10, Chapter 28] mentions that most powerful DSP units from the late 90's have separate ALUs, multipliers and barrel shifters in order to parallelize these operations.
- **Extended precision accumulators,** which are used to store the results of the multiplications without reducing the resolution, and thus minimizing the quantization noise added by the processing.
- **Available parallelism:** being able to execute many operations simultaneously reduces the execution time and allows execution of more complex algorithms in real-time. An example of this would be being able to access memory while performing a multiplication.

The processor used in this work to compute the presented DSP algorithms is Patmos, which will be described in Section 3.2. Patmos is not a DSP processor,

but a general-purpose real-time processor. Using Patmos to perform digital audio processing in real-time limits the complexity of the algorithms that can be implemented: in order to not exceed the execution time limits, the effects cannot have complex arithmetics, such as high order filters or a big amount of multiplication operations. For instance, real-time FFT processing is unfeasible in Patmos. That is why the audio effects implemented in this project are not complex or very high quality, but they are enough for building a multicore audio processing platform. The system has a high scalability, as it will be demonstrated in Chapters 6 and 7, so powerful DSPs or GPUs could be integrated into the network in the future to implement more complex algorithms.

T-CREST Background

This chapter presents the T-CREST platform, which is used in this project as the audio processing multicore platform. The chapter provides some aspects of the background and current state of the T-CREST project. In Section 3.1, a general overview is given. In the following Sections 3.2, 3.3, and 3.4, some parts of the T-CREST platform are explained, which are the most relevant ones for this project. They are the *Patmos* processor [11], the *time-analysis tools* [12] and the *Argo* Network-on-Chip [13], [14].

3.1 Overview of the T-CREST Platform

T-CREST¹ [15] is an open source research project that is continuously under development. The goal of the T-CREST project is to develop a general-purpose fully time-predictable multicore processor platform for embedded real-time applications. The T-CREST platform consists of a set of time-predictable resources: these include not only processors, memories and communication networks, but also tools for time-analysis and measurement. The goal of these resources and tools is both to reduce the Worst Case Execution Time (WCET)

¹<https://github.com/t-crest>

of any set of tasks executed in the platform and to achieve high predictability of the WCET to be able to provide timing guarantees.

Figure 3.1 shows the hardware side of the T-CREST platform, which consists of a set of IP cores (4 in this case, on a 2-by-2 topology) connected by a message-passing Network-on-Chip (NoC) to exchange data between them. Each of these cores is a statically-scheduled RISC-style processor called Patmos, which is equipped with a set of local memories (instruction and data caches and SPMs). The NoC is the time-predictable Argo NoC. Both Patmos and Argo are specially designed for the T-CREST platform, although theoretically the NoC can connect not only Patmos processors, but also other kinds of IPs with a compatible interface [16]. The platform is also equipped with an off-chip shared RAM memory, which has a memory controller that the cores can access by using a memory-tree NoC. This one is not shown in Figure 3.1.

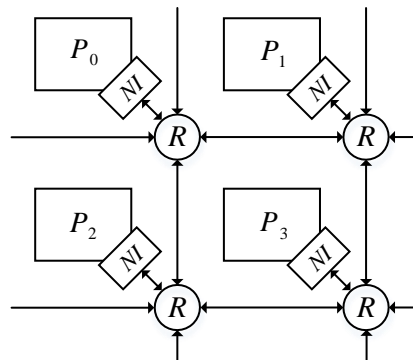


Figure 3.1: Overview of the 2-by-2 T-CREST platform, showing the cores connected by the NoC. The processors (P), Network Interfaces (NI) and Routers (R) are shown. Main memory is not shown.

Before going deeper into each of the parts that compose the T-CREST platform, one should know that there are different versions of it with different characteristics: for instance the Argo NoC has both a Globally-Asynchronous Locally-Synchronous (GALS) and a Globally-Synchronous version; for the Patmos processor, there is also an older version designed in VHDL, while the newest version uses the Chisel language. For this project, the T-CREST platform is built in the Altera DE2-115 FPGA board [17], and uses the Chisel version of Patmos with the Globally-Synchronous Argo NoC, synthesizable on FPGAs. The main memory is an off-chip SRAM, and some other off-chip I/O components of the board are used, such as the WM8731 audio CODEC presented in Section 4.1.

3.2 The Patmos Processor

Patmos is a time-predictable 32-bit Very-Long Instruction-Word (VLIW) RISC processor designed for embedded real-time applications [11]. In this work, Patmos has been used as the main computational resource to process the audio effects. Subsection 3.2.1 introduces the 5-stage pipeline architecture of Patmos, and Subsection 3.2.2 explains the local/global memories and IO devices that it has access to.

3.2.1 Architecture

Patmos consists of a classic RISC-style 5-stage pipeline, which is shown in Figure 3.2. For some instructions, some additional pipeline stages are used, which are not shown in Figure 3.2. An example of this is the multiplication instruction, which uses a parallel pipeline to the EX stage with a fixed length. Each one of the 5 stages is briefly explained here:

- **Instruction Fetch:** on this initial stage, the next instruction (or next two) are fetched from main memory or from the instruction cache. The program counter is also updated.
- **Instruction Decode:** here, the instruction is decoded and control signals are generated for the following stages. The operands are also read from the register file on this stage.
- **Execute:** the predicate registers are read and the ALU instructions are executed, if needed. Addresses for memory access are also calculated on this stage when needed.
- **Memory:** the memory is accessed, either by a load or store operation. This stage might cause a pipeline stall, if a cache miss happens.
- **Write Back:** on this final stage, the results are written into the destination registers.

As mentioned before, there is a separate stage for multiplication, which takes 3 cycles to execute in the current FPGA version (but it is still possible to issue one multiplication per cycle). This stage is repeatedly used in this project because, as it has been shown in Chapter 2, the two most common arithmetic operations for audio signal processing are the addition and the multiplication. This stage also represents an important limitation for the system: it can only perform fixed-point multiplications. The floating-point multiplication instruction is a software

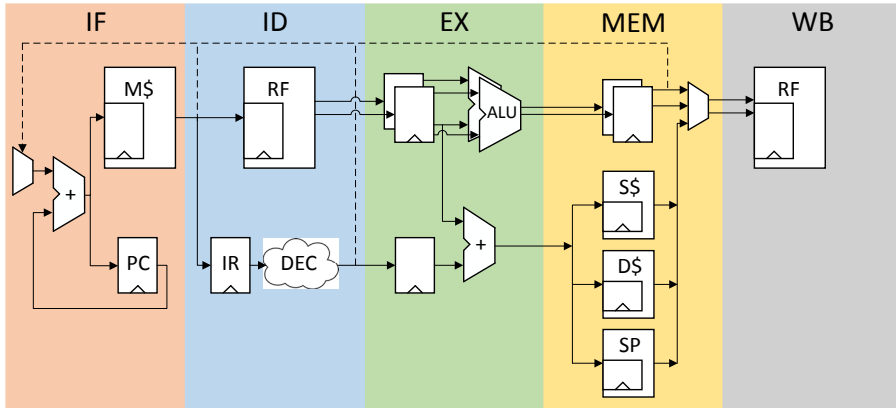


Figure 3.2: 5-stage pipeline of Patmos, showing the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write-Back (WB) stages.

routine which increases the execution time considerably. This has caused all the audio effects to be implemented in fixed-point arithmetic in this project. A deeper discussion on this is provided in Section 5.2.

It is also important to outline that, as said previously, Patmos is a dual-issue processor. This means that it can issue two instructions simultaneously, and this is because there are 2 copies of the 5-stage pipeline. One of them only features a subset of the functions of the other, which means that only some of the instructions are dual-issue. For instance, most ALU operations are dual-issue (Figure 3.2 indeed shows two parallel ALUs), but memory operations are single-issue, as data memory is only accessible from one of the pipelines.

3.2.2 Memory System and I/O Devices

Some of the most used operations for audio signal processing, together with the addition and multiplication, are the memory access operations. Patmos uses local and global address spaces to access different memories or devices. To access the external SRAM memory or the instruction ROM where the bootloader is located, the global memory space is used.

Patmos has different cache memories, which are used to hold copies of data that the processor might use in the near future due to temporal or spatial locality. These have single-cycle access, which prevent the processor from stalling due

to longer main memory access. Some parameters of the caches are configurable when building the platform, such as the size, the replacement policy and so on. The 3 types of caches that Patmos features are the following:

- The **stack cache** acts as a window to the current global memory address space. It acts as a circular buffer. The version of T-CREST used for this project has a 1 kB stack cache.
- The **data cache** is used in addition to the stack cache to hold copies of data, but this one is used for caching regular cached memory accesses. The data cache used in this project is 4 kB, one-way set associative (direct mapped), which means that each data block from main memory can only go to one address on the cache.
- The **instruction cache** holds copies of instructions that have either temporal or spatial proximity to the instruction that is currently under execution. In this project, a 8 kB method cache [18] has been used, which will often hold copies of entire functions in the cache (if they fit). With this type of caches, misses will only occur on `call` or `return` instructions. The associativity is set to 32, which means that up to 32 methods can fit in the cache. This value is quite high, but the instruction cache has proved to be the most important timing bottleneck of the system, because it can slow down the processing of the audio effects a lot due to instruction misses. A FIFO replacement policy has been used.

Additionally, each Patmos core has local single-cycle access Scratch Pad Memories (SPM) mapped to the local address space. The SPM is a true dual-port RAM, and Patmos has SPMs both for data (DSPM) and instructions (ISPM). These can be used by the programmer to explicitly place frequently used data or functions on them to avoid cache misses and to ensure to have constant read/write times, which is essential for constrained WCET requirements. The SPM is also used as a communication port for message passing, because it is connected to the Network Interface of the Argo NoC. Unfortunately, the ISPM could not be used in this project to reduce WCET in the multicore platform, as it will be explained in Section 5.4.

Finally, the I/O devices are also mapped to the local memory space. Some of the devices found in Patmos by default are the CPU info, the interrupt controller, the timer and the UART. In addition, the designed audio interface (Chapter 4) is also found as an I/O device on this project, and it is used for communication between Patmos and the WM8731 audio CODEC. Patmos interfaces these hardware components using a subset of the standard Open Core Protocol [19].

3.3 Compiler and Time-Analysis Tools

The *Patmos Compiler* [20] has been continuously used in this project. It is an adaptation of the LLVM compiler [21], and allows the Patmos processor to run C programs. Not only can it generate machine code for the Patmos ISA, but it also has high integration with the WCET analysis tools.

The main driver of the Patmos compiler used in this work is `patmos-clang`, which generates bitcode files from C source files and system libraries. One of the main tasks of a compiler for real-time systems is to reduce the WCET as much as possible.

An important software tool used mostly in the initial stages on this project is the *patmos-emulator*, which is a C++ based simulator generated from the hardware description of the processor. It behaves identically to the hardware processor, and it has been used to test the functionality of the digital hardware blocks designed in this project, which form the audio interface. Another simulator is also provided, called *pasim*. This one provides a high-level model of Patmos, and is therefore not so useful for hardware simulations.

Finally, the *platin tool kit* [12] is a key component of the T-CREST platform, as it features a set of tools for WCET analysis. This tool kit uses PML (*Platin Metainformation Language*) format files both for configuration and analysis.

The main driver used here is `platin wcet`, which performs static WCET analysis of a function specified by the user. The bounds for the `for` and `while` loops must also be set by the user. Apart from the C file, the tool also needs a PML file containing the configuration of Patmos (memory and cache sizes and parameters). The steps needed to perform WCET analysis are the following:

- Compilation of the C program, specifying the function to be analysed by the WCET tool.
- WCET analysis of the files generated by the compiler, using the `platin` tool with the specified configuration file.

The WCET analyser supports both the commercial *AbsInt aiT* tool and an internal `platin` tool. In this project, the `platin` tool has been used. When finished, it reports the result of the WCET analysis, where the WCET of the specified function is measured in clock cycles. However, the WCET analysis results given by the tool have proved to be quite pessimistic, as they differ from the experimental measurements, as shown in Subsection 7.2.1.

3.4 Argo NoC

Argo is a statically-scheduled NoC which, together with the SPM of Patmos, implements the message passing between the cores [13], [14]. Argo is used as the main communication resource in this project: the Patmos cores that are available in the platform take care of the computation of the audio effects, while the audio signal travels from one core to another through Virtual Channels (VC) on the NoC.

As Figure 3.1 shows, Argo is composed of Network Interfaces (NI) and a packet switching structure, which consists of routers and links. The links provide connections between the routers depending on the chosen topology: in this case, the *bitorus* topology has been chosen, which is the one shown in Figure 3.1.

The way the packets travel through the routers from source to destination is determined by the TDM schedule, which is explained in Subsection 3.4.1. After that, Subsection 3.4.2 briefly explains the NI, which stands between Patmos and the routers. Finally, Subsection 3.4.3 explains reconfiguration, one of the features of the newest version of Argo.

3.4.1 TDM Scheduling

The Argo NoC uses Time-Division Multiplexing (TDM) [13], which means that the resources of the NoC are shared over time between the VCs required by the application. To achieve this, time is divided into TDM periods, and a period is divided into time slots. The TDM period is determined by the off-line scheduler, according to a description of the NoC topology and the required VCs and their bandwidth. These parameters basically depend on the communication requirements expressed as a task communication graph (in this project, each audio effect corresponds to a task, and the communication requirements are the signal flow through the effects). The schedule is stored in the NIs. TDM avoids deadlocks and collisions, and ensures that all packets arrive in order, in a guaranteed time range. This is ideal to reduce WCET.

Each time slot corresponds to a *phit*, which is the finest granularity unit of the physical layer that the NoC can transfer. In this case, a phit is one 32-bit word. A NoC packet is composed by one or more phits, and the packet length is equal for all VCs in Argo. The TDM bandwidth for each channel is applied at a packet level. A common value is 3 phits per packet, where the first phit is always the packet header that contains information about the write address and the packet route. It also contains information about the type of packet: in this project,

the main type of packet used is the data packet to transfer audio samples, but reconfiguration packets are also used if reconfiguration of the NoC is enabled.

A full TDM period is the sum of packets that have been statically assigned to each VC by the scheduler (plus possibly some empty slots that avoid collisions), where each packet contains the same amount of phits. The packets are injected in the NoC by the NI, according to the static schedule table, in the order given by the TDM schedule. The routers in the structure continuously transfer these packets. When a TDM period finishes, the next period starts right away. If, at some point in time, there is no data on a given time slot, the router does not transfer any data. That could happen because *a)* the packet assigned to that time slot does not travel through that router (i.e. it takes another path); *b)* because there is no data being sent on that VC at that moment; or *c)* because that slot was left empty by the scheduler to avoid collisions between packets.

For better understanding, Figure 3.3 shows an example of a possible TDM period. There are 4 packets in the period, with 3 phits per packet (the first one is the header, H). There are 3 VCs in the channel: c_1 and c_3 with a bandwidth of 1, and c_4 with a bandwidth of 2 packets. There is also an empty time slot, which could have been assigned by the scheduler. The total TDM period is 13 time slots.

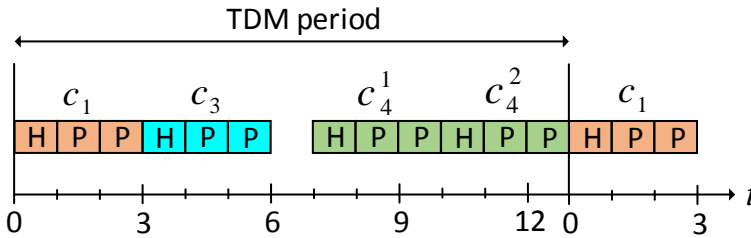


Figure 3.3: Example of a possible TDM schedule, where there are 4 packets of 3 phits corresponding to 3 VCs (c_4 has a bandwidth of 2 packets).

3.4.2 Network Interface

The NI offers standard read/write transactions between the NoC and each Patmos core in the platform [13]. It also takes care of the packet sending/reception to/from the NoC.

As Figure 3.4 shows, the NI contains a *TDM counter*, a *schedule table*, a *Direct Memory Access (DMA) table*, a *receive unit* and a *reconfiguration controller*

(the last one is explained in Subsection 3.4.3). The counter indexes an entry in the schedule table, which points to an entry in the DMA table. This entry contains parameters of a DMA controller, and also the route that the packet of that TDM slot should follow through the NoC. The TDM counters of all cores are synchronized, so all NIs read the same schedule table entry all the time.

For each packet in the TDM period, the indexed DMA controller of the corresponding source node reads the data from the local SPM and transfers the packet to the NoC router. This packet then travels through the routers and links given the route saved in its header. Finally, when the packet arrives at its destination NI, the receive unit takes care of writing the packet data into the SPM address of the destination core, specified in the packet header.

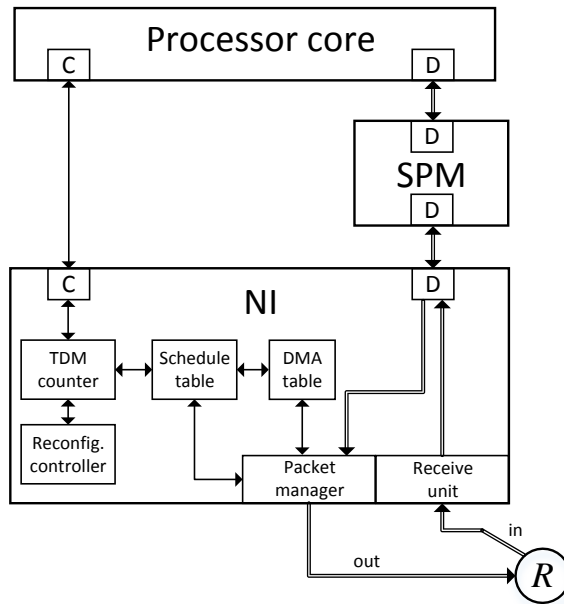


Figure 3.4: Network Interface of Argo, between Patmos and the router. Single lines indicate control signals, and double lines indicate data.

Now that the process of sending data packets between cores through the NoC has been explained, it is important to provide bounds of the time that it will take for a packet to travel from the local source core to the remote destination core. In this project, this is referred to as the *worst-case packet latency* and it is the time that passes from when the local core starts sending the first phit of a packet until the remote NI writes the last phit to its SPM. This value can vary depending on where in the TDM period the core performs the send operation: it will be faster if the sending starts right before the time slot assigned to that

VC than if that time slot has just passed, because the NI will have to wait for one full TDM period. The worst-case packet latency value, $L_{P_{wc}}$, can be defined as in Equation 3.1, where P_{TDM} is the TDM period. The value 8 depends on architecture details of the Argo NoC.

$$L_{P_{wc}} = P_{TDM} + 8 \quad [clock \ cycles] \quad (3.1)$$

It is also important to mention that the NI operates in an overlapping way with the processor, i.e. Patmos can perform other operations in parallel while the NI is sending data (it should just take care of not overwriting the send data on the SPM).

In this project, a message passing library called *libmp*² has been used, which is provided in the Patmos repository. It is built on top of the *libnoc* library that interfaces the hardware directly. The *libmp* library offers some of the following functions:

- `mp_create_qport` and `mp_create_sport` are used to create either a queuing or a sampling NoC channel port. The channel id, the direction, the buffer size and the number of buffers need to be specified. The queuing communication has been used in this project, which means that the data is stored in a queue of buffers in the receiver, instead of being overwritten.
- `mp_init_ports` needs to be called after creating all the ports on each core.
- `mp_nbSEND` and `mp_send` send data through the NoC on the specified channel. A time-out value might also be given, in case sending fails. The difference between the two functions is that the first one is non-blocking and the second one is blocking. The blocking one has been used, which waits until the send operation has been completed before performing the next send on the same buffer, to make sure no data is lost.
- `mp_nbrcv` and `mp_rcv` receive data from the NoC on the specified channel. Again, a time-out value can be set. The blocking function has also been used for receiving.
- `mp_nback` and `mp_ack` are used by the receiver to send the acknowledge signal and let the sender know that the data was received correctly. Again, the blocking function has been used.

²<https://github.com/t-crest/patmos/tree/master/c/libmp>

3.4.3 Reconfiguration

The Argo NoC supports reconfiguration, which allows performing mode changes in real-time [13]. A mode change is the change of software tasks that are being executed as a response to an external event. When this happens, a change in the state of the NoC might be needed because new channels and bandwidths are required, and old ones might disappear. This is the reconfiguration of the NoC. On an audio application, a mode change means switching the audio effect chain that processes the input sound. This could happen for instance when a guitar player presses a foot-switch pedal to shift from one set of effects to another, and the he/she should perceive the mode change as instantaneous. The implementation presented in this work supports mode changes among effect setups, and NoC reconfiguration has been used in some cases to update the communication channels according to the requirements of each effect configuration.

A set of available modes must be determined off-line by the scheduler: each mode has its own TDM schedule. During run-time, the reconfiguration controller in the NI holds the TDM schedule information of each mode. When the master core invokes reconfiguration, it broadcasts a reconfiguration packet to all slaves. The reconfiguration controllers in the NIs of the slaves will then update the schedule table. The reconfiguration packet latency is the packet latency defined in Equation 3.1. This means that the new mode is available at least 2 TDM periods after the master issues the mode change (i.e. the new mode will start with the beginning of the 3rd TDM period after mode change).

Finally, a slight drawback of supporting reconfiguration is that one VC with the bandwidth of the reconfiguration packet is always needed between the master core and each one of the slave cores. This means reserving some bandwidth for these VCs in the TDM period.

Architecture of the Audio Interface for Patmos

This chapter presents an audio interface for Patmos and the WM8731 audio CODEC included in the Altera DE2-115 board. The design and digital implementation of an older version of this interface was previously done as a project [22] for the course *02211 - Advanced Computer Architecture of DTU*. The main contribution done during this thesis is the addition of input and output audio buffers, which will be explained in this chapter. The rest of the components will be just overviewed.

In Section 4.1, the main characteristics of the WM8731 audio CODEC are briefly introduced. Section 4.2 explains the components of the audio interface, focusing on the input and output buffers, which are the latest addition. Finally, Section 4.3 describes the API for using the audio interface with Patmos.

4.1 WM8731 Audio CODEC

The WM8731 is a low-power stereo audio CODEC with integrated headphone driver and line and microphone inputs [23]. It has 24-bit sigma-delta ADCs and DACs. Sampling rates from $8kHz$ up to $96kHz$ are supported. The CODEC

has two main interfaces: the control interface and the digital audio interface.

The control interface is controlled via an I2C bus. Here, some of the configuration parameters are defined, such as the input and output selections (line in, microphone in, line out, headphone out), the audio resolution, the sampling rate, the output volume, and so on. This configuration is always done in the setup of each audio program. For this project, the 48 kHz 16-bit option has been chosen, which is a standard audio quality. However, the actual sampling rate is not exactly 48, but 52.083 kHz . This is because the WM8731 requires a source clock signal of 12.288 MHz to sample audio at 48 kHz . Instead, the provided clock is 13.33 MHz , which is the Patmos frequency of 80 MHz divided by 6. So the sampling rate is $\frac{48 \cdot 13.33}{12.288} = 52.083\text{ kHz}$. This does not represent any decrease in the performance or in the audio quality of the system because the sampling rate is even greater than 48 kHz , so the Nyquist theorem is fulfilled.

The digital audio interface is used for the input and output of the digital audio samples. This is done through 5 wires: DACDAT, DACLRC, ADCDAT, ADCLRC and BCLK. The LRC signals are used for sampling synchronization: they provide a pulse every sampling period. The DAT signals transfer the actual audio data, bit by bit. Finally, the BCLK signal is a synchronization clock for every sample bit.

The WM8731 supports many digital audio interfacing modes, and here the chosen one is the DSP mode A. Briefly, this means that a LRC pulse comes every 256 BCLK cycles. After each pulse, first the 16 bits of the left channel sample are transferred on every BCLK pulse, and then the 16-bits of the right channel. After that, there is no data for the next 223 cycles, until the next LRC pulse happens.

Both I2C signals and all the mentioned digital audio interfacing signals are connected from the Altera DE2-115 FPGA to the WM8731 in the board. The FPGA is intended to be used as a master, and the WM8731 is the slave.

4.2 Design of the Audio Interface

The designed audio interface [22] drives both the control signals and the digital audio signals of the WM8731. On the other side, it is connected to the Patmos processor as an I/O device. Patmos acts as the master and communicates via the OCP protocol. This makes it easy for the programmer to read input audio samples from registers and to write output audio data.

The Audio Interface and its internal components are shown in Figure 4.1. The HDL implementation of the input and output buffers can be found in appendix A, which are the main contribution to the interface done in this project. The rest of the components can be found at the Patmos GitHub repository¹. Each component does the following:

- **AudioInterface**: it is the top component, which connects the sub-components between them, to Patmos via OCP or to the WM8731 audio CODEC through the FPGA ports.
- **AudioClkGen**: it generates the 13.33 *MHz* XCLK and BCLK signals from the 80 *MHz* clock of Patmos.
- **AudioI2C**: when Patmos writes new data in the configuration data and address registers, this block transfers it to the WM8731 in the I2C format. It then waits for the acknowledge from the CODEC.
- **AudioADC** and **AudioDAC**: these two blocks exchange the input and output audio data respectively between Patmos and the audio CODEC, using the explained DSP mode A format. The **AudioADC** generates the ADCLRC pulse, and stores the data from ADCDAT in the left and right input audio registers connected to the input buffer. The **AudioDAC** block does the inverse process for the output.
- **AudioADCBuffer** and **AudioDACBuffer**: these are the input and output buffers respectively. They exchange data with Patmos via handshaking, and are essential to implement the flow control communication used in the system, explained in Subsection 5.1.1. However, the latency of the system is also increased by increasing the buffer sizes, and one must be careful with this so that the real-time perception of audio is not lost.

As mentioned before, the buffers are the main contribution to the interface done in this project (apart from minor changes in other components, to adapt to the buffers). Their hardware implementation is shown in appendix Sections A.1 and A.2. Some of the main characteristics of them are explained in the following points:

- Both the **AudioDACBuffer** and the **AudioADCBuffer** are implemented as circular FIFO buffers which wrap around, as explained in [10, Chapter 28]. For this, the buffer size must always be a power of 2, where the maximum size is 256 (this value could easily be increased in a bigger platform).

¹<https://github.com/t-crest/patmos/tree/master/hardware/src/io>

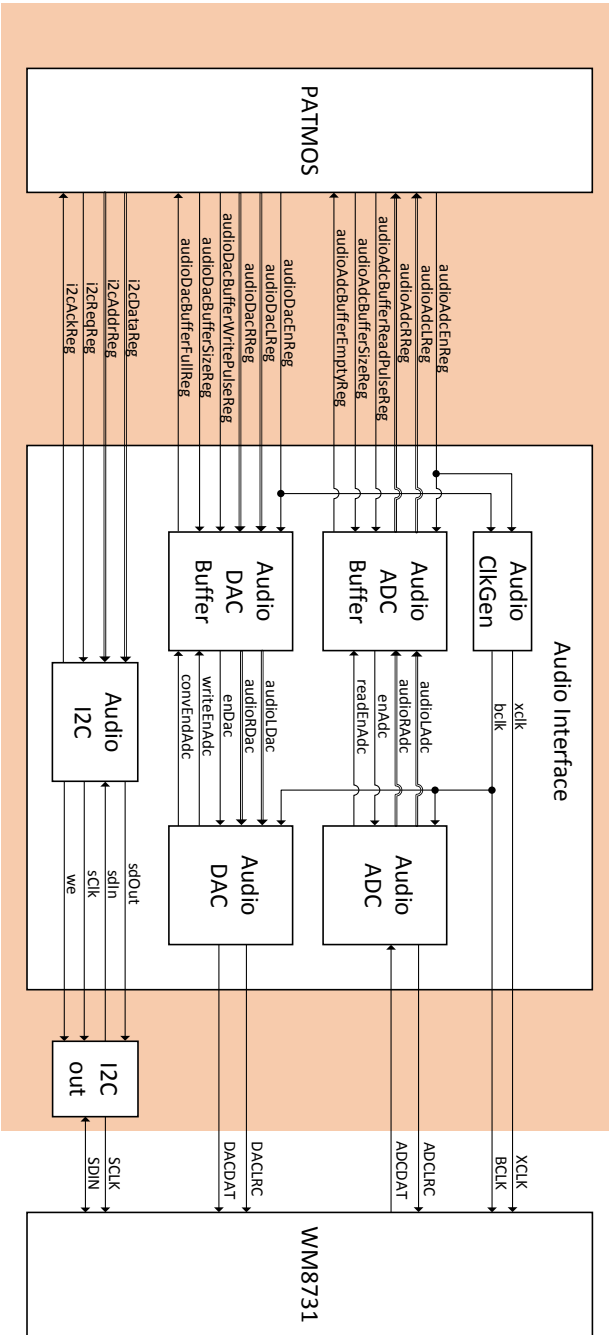


Figure 4.1: Architectural overview of the audio interface, showing its components and connections to Patmos (via OCP) and to the WM8731 audio CODEC. The colored background indicates that the components are on the same FPGA chip.

- They have read and write pointers, whose bitlength is equal to the amount of bits needed to represent the buffer length, for easy wrapping around (for instance, 8-bit pointers for a 256-sample buffer). The pointers are automatically incremented when a read or write is issued by Patmos or by the AudioADC/AudioDAC blocks.
- A state machine takes care of the state of the buffer, with states for *empty*, *almost-empty*, *idle*, *almost-full* and *full* situations. The state is idle when the buffer is not empty, almost empty, full or almost full. The *almost* cases are needed for when the buffer size is 2 samples, because a transition from idle to empty or to full cannot be distinguished without these states.
- The state machines also generate *empty* and *full* flags. Patmos can read audio data while the input buffer is not empty, and it can output data while the output buffer is not full. If the output buffer is empty during a time range longer than the sampling period, undesired interruptions will be heard in the processed audio output because some samples have been dropped out.
- Finally, the state machine of the AudioDACBuffer also drives the enable signal of the AudioDAC block. The audio output is enabled only when the DAC buffer is not empty (i.e. there is data to output). The AudioADC enable signal is always high, and if the input buffer is full, input data is overwritten in the buffer.

An important design decision has been to make the read/write pointers auto-incrementing. This means that Patmos can read an input sample only once. If a specific audio effect needs to buffer a group of samples, they need to be stored somewhere else when reading. This is exactly what happens on the filter effects, where a buffer is needed to store the newest input samples. These samples are stored in the SPM because it is a fast access memory. This is represented in Figure 4.2, showing that the samples for filter calculation are stored in the SPM.

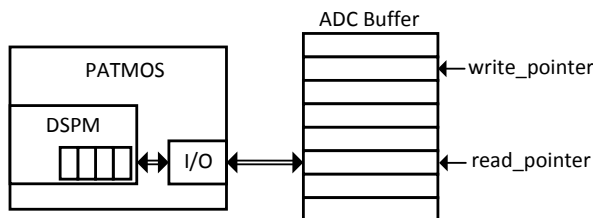


Figure 4.2: Representation of the input buffer and Patmos, showing how the latest samples needed for filter calculation are transferred to the data SPM.

Another possibility would have been to be able to change the read pointer of the buffer from Patmos, so that storing the input samples in the SPM would not be needed. But this requires a more complex buffer (it would not be a FIFO anymore) which is not needed for this project, as the data SPM is enough to store the buffers of each effect.

4.3 API of the Audio Interface

Some functions for configuration and audio input/output from/to the audio interface I/Os are provided in the *libaudio*² C library for Patmos. These functions are shown in the appendix Section A.3. The main function for configuration is the `setup` function, which sets standard configuration parameters for the WM8731 CODEC. The only choice is whether the input audio comes from the line input or from the microphone input (which has a built-in preamp and is useful for microphone or guitar signals). A function to change the volume has also been provided, although it has not been used in this project.

Before starting the audio processing, the input and output buffer sizes must be set. For this, the `setInputBufferSize` and `setOutputBufferSize` functions are provided, which take an argument with the size value.

Finally, the two main functions used during processing are `getInputBufferSPM` and `setOutputBufferSPM`: the first one reads data from the read pointer of the ADC buffer and places it in given SPM address. The second one does the opposite process. The way these functions work is the following:

- The `getInputBufferSPM` function first waits until the ADC buffer is not empty. When it is not, it provides a read pulse to the `AudioADCBuffer` block so that this one knows that Patmos is reading one sample and it can update the read pointer. Then, the left and right audio data is copied to the SPM.
- The `setOutputBufferSPM` function starts by copying the left and right audio output samples from the SPM into the intermediate registers of the `AudioDACBuffer`, and it then waits until the buffer is not full. When it is not, Patmos issues a write pulse so that the data can be copied from the intermediate register into the buffer.

²<https://github.com/t-crest/patmos/tree/master/c/libaudio>

CHAPTER 5

Design and Implementation of Audio Effects on Patmos

This chapter presents the considerations and decisions taken in the design and implementation process of the audio effects for the Patmos processor. Some given specifications are considered, such as the real-time approach to audio processing and the characteristics of the software and hardware resources available in the processor, as explained in Chapter 3. The design trade-offs are also explained and discussed. In this chapter, each effect is treated separately from other effects, as an individual processing unit that runs on a single-core platform.

These effects are based on the DSP algorithms for audio signal processing described in Chapter 2. Many references have been found which provide an implementation of those effects in different languages such as `C`, `Matlab` or `Python`. However, in all of them the processing is done off-line, not in real-time. That is why the real-time implementation of these effects is a contribution to the T-CREST project. The effects designed here will be connected to each other to form audio effect chains on a multicore platform, which will be explained in Chapters 6 and 7, also contributions to the T-CREST project.

First of all, Section 5.1 explains some of the main requirements that a real-time digital audio processing system must accomplish. Section 5.2 then explores the benefits and drawbacks about one of the main discussions in the

DSP world: fixed-point v.s. floating-point processing. Section 5.3 presents the object-oriented style approach used to implement the audio effects on Patmos. Finally, Section 5.4 lists and describes the main parameters and functions of the effects implemented in this project.

5.1 General Requirements for Real-Time Audio Processing

The design of real-time audio processing systems is a challenge compared to that of off-line systems: in off-line processing, the entire piece of audio that is to be processed is stored in the memory. This means that the signal has defined start and end points, and some characteristics of it can be analysed before starting to process, such as the dynamic range or the power of the signal. These characteristics might be used for tuning some processing parameters. Moreover, there are no strict requirements of the time needed for processing. However, this is not the case in real-time audio: the system has no knowledge of the audio signal except for what it can analyse from the current input. The processing needs to be done sample-by-sample (or block-by-block), in a way that it is possible to compute every sample within a certain time interval.

The most important requirement of a real-time audio system is that it must be powerful enough to process the audio data 'in time': for a given audio sampling rate, this basically means that **the processing time per sample must be smaller than the sampling period**. This is shown in Equation 5.1, and the value ϵ is the time margin left between two consecutive samples, where the processor is in idle state. F_s is the sampling rate, and t_{P_n} is the time required to process a sample for effect n .

$$t_{P_n} + \epsilon = \frac{1}{F_s} \quad [s] \quad (5.1)$$

Figure 5.1 shows an overview of the path done by the audio signal in the single core platform. As explained in Chapter 4, input and output buffers are used to hold previous and next samples of the audio signal while Patmos is processing. The picture in Figure 5.1 is important to understand the communication paradigms explained in Subsection 5.1.1, and how the latency of the audio signal for the single core platform is calculated in Subsection 5.1.2.

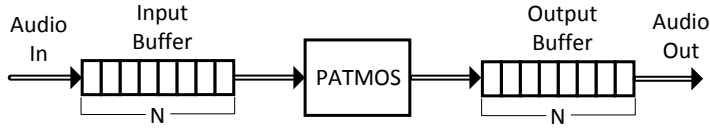


Figure 5.1: Representation of the audio signal flow on the single core T-CREST platform, with input and output buffers of the same size, N .

5.1.1 Communication Paradigms

The communication paradigm used to exchange data between the components of the system plays an essential role in the behavior of it. In [24, Chapter 2] different communication methods are discussed. The book does not focus in any specific application, but two interesting methods are explained which are valid for the T-CREST audio processing platform. The components that form this system are the audio input system (including the ADC and the input buffer), the processing system (as many as the amount of Patmos cores) and the output system (including the output buffer and the DAC).

- One of the paradigms is *time-triggered* communication, which is used when there is a periodic exchange of data between the components. This is the case of digital audio systems, because the samples arrive at a fixed frequency. Time-triggered communication can be achieved using an interrupt with a frequency equal to the sampling rate, indicating the arrival of a new sample. The processor then computes the sample and waits until the next interrupt happens: there is no need for handshaking with other components. In this case, the processing of each sample must be computed faster than the sampling period in absolutely all cases to avoid dropping out samples.
- The other possibility is to perform *flow control* communication, where there is no external synchronization signal. This is usually used when the exchange of data is non-periodic, and the data arrival times might be unconstrained. However, it is also useful for audio processing systems because it provides flexibility in the processing time, as it will be shown later. In this case, the processor computes one sample, and when it finishes it sends it out and continuously requests the next one. Each component needs to do some kind of handshaking with others.

The communication paradigm chosen for this project is the second one, *flow control*. To explain why, it is important to refer to Figure 5.1 and explain how the system works. Just before the processing starts, both the input and output buffers (of size N) are empty. When it begins, first the input buffer is allowed to load fully, so no processing is done during the first N samples. On the N -th sample, the processing starts, so the first sample that was input to the system is processed and sent to the output buffer, which will directly send it to the DAC. After this point, the input and output of audio data is done at a constant rate: one sample every $1/F_S$. That means that there are always N samples inside the system, distributed among the input buffer, the processing system and the output buffer.

As shown in Equation 5.1, the time it takes for the processor to process one sample is smaller than the sampling period. That means that some time after the processor starts processing the first sample when the input buffer is full, this one will get emptied completely and the output buffer will get full, because the processing rate is faster than the sampling rate. So in the 'stationary' situation, the input buffer will be empty, and the output buffer full.

Now, the equation presented in 5.1 is not true for absolutely all cases: in some minor cases, the processing time of an effect per sample might exceed the sampling period. This happens because the processor uses caches to have fast access to data and instructions, so the execution time will increase when cache misses occurs. This situation is given just once in a long while: when the first sample is processed, or when there is a mode change and new effects are loaded into the cache.

And this is actually the reason why flow control communication has been chosen: if time-triggered communication was used, interruptions would happen in the audio signal when cache misses occur, because the processing would not be done 'in time'. However, if this happens with flow-control communication, the output signal does not get interrupted as long as there are still samples in the output buffer. In this case, the input buffer will store samples until the processor is able to process in time again (i.e. fulfilling Equation 5.1). After some time, the stationary situation would be given again, where the input buffer is empty and the output buffer is full.

In general, it can be stated that the input and output buffers provide the system with elasticity against audio drop-outs when cache misses happen. The size of the buffers, N , must be chosen carefully to make sure that the output buffer is large enough and the system is still able to output data uninterruptedly when the WCET processing happens (on a cache miss, as explained). For a given effect n , the WCET processing will be the sampling frequency ($1/F_S$) multiplied by some overhead value T_{OH_n} . This value gives an idea of how many sampling

periods it takes to process a single sample. The output buffer must then be able to hold at least T_{OH_n} samples when a cache miss happens. This is shown in Equation 5.2.

$$N \geq \lceil T_{OH_n} \rceil \quad [\text{samples}] \quad (5.2)$$

If this is accomplished, then the audio processing system can be considered a hard real-time system, even if the processing of samples of each effect does not fulfill Equation 5.1 in absolutely all cases.

The stationary case execution time will be given by Equation 5.1, where ϵ_n is the time margin for effect n , and gives an idea of how long it will take for the system to go back to the stationary state after a cache miss happens. However, this is difficult to calculate, because for most effects ϵ_n is not a constant value.

5.1.2 Signal Latency

Another essential requirement for a real-time audio processing system is that the processing time must be perceived as instantaneous by the human ear. This means that signal latency from input to output must always be within a certain time interval.

There are still many open discussions about what is an acceptable latency for a digital audio processing system to be considered real-time. In [25] it is stated that typical audio processing system latencies range from 0.5 to 10 ms, some having up to 30 ms. It is also stated that, in one study, listeners were able to perceive latencies greater than 15 ms as a delay. In [26], a discussion of the latency of *CSound* is given. *CSound* is a sound computing system for major operating systems such as Android or iOS. In this reference, it is mentioned that a delay of 12 ms is acceptable. Many other resources also provide values that are within the same range. With all these values in mind, an upper limit of latency is decided for this project, which will be 15 ms. For the sampling frequency of 52.083 kHz used, the 15 ms latency corresponds to 781 samples.

In a single core platform processing system as the one in Figure 5.1, the latency depends solely on the size of the input and output buffers, N . As explained before, there are always N samples inside the system, distributed among the input buffer, the processing system and the output buffer. That means that the latency of the audio signal in a single core platform (L_{SC} , measured in samples)

is constant and can be calculated as shown in Equation 5.3. Alternatively, the latency can be converted from sample units to seconds simply dividing L_{SC} by the sampling frequency F_s .

$$L_{SC} = N \quad [samples] \tag{5.3}$$

The calculation of latency on a single core platform is therefore simple. Nevertheless, it gets more complex on a multicore platform. This will be explained in Section 6.5.

5.2 Fixed-Point v.s. Floating-Point Audio Processing

Fixed-point and floating-point representations are the two main ways used by DSP processors to store and process data. The fixed-point representation can be used to represent signed or unsigned integers and fractions. The values are equally spaced: the gap between them is fixed, thus the name. This does not happen in the floating-point representation, which usually uses a minimum of 32 bits and has a mantissa and an exponent. Here, the gap between values is non-constant: it is large between large numbers and small between small numbers. Fixed-Point and Floating-Point DSPs have different pros and cons, which are discussed in [10, Chapter 28], and also listed here:

- **Complexity:** the hardware architecture needed to perform floating-point arithmetic is more complex than the one for fixed-point DSP. This is an important drawback of floating-point processing.
- **Price:** as a consequence of a more complex architecture, floating-point DSPs are usually more expensive than fixed-point processors.
- **Dynamic range:** floating-point DSPs have a much higher dynamic range than fixed-point ones. In 32-bit representation, signed integers can represent values ranging from around $-2.15 \cdot 10^9$ to $2.15 \cdot 10^9$, while same bitlength floating-point values can represent values between $\pm 3.4 \cdot 10^{38}$ and $\pm 1.2 \cdot 10^{-38}$ (in the most used ANSI/IEEE standard). This is specially interesting in some audio effects, such as the filters, where the dynamic range of the output signal increases with the order of the filter. Floating-point makes it easier to represent these values.

- **Development time:** the development time of an audio processing algorithm implementation is usually greater when fixed-point arithmetic is used: this is because the programmer constantly needs to take care of possible overflow/underflow situations of an arithmetic operation, due to the limited dynamic range of fixed-point representation. The multiplication of two N -bit fixed-point values results in a $2N$ -bit fixed-point value, which causes having to scale the values up and down constantly. Moreover, the addition/subtraction of two N -bit values might result in a $N + 1$ -bit value, which again means that some scaling needs to be done. All of this is avoided when floating-point arithmetic is used.
- **Precision:** the problem of having a non-constant bitlength in fixed-point arithmetic operations means that resolution can be easily reduced when scaling or rounding off is done (for instance, when throwing out the least significant bit of the result of an addition). In each of these operations, quantization error is introduced and the signal-to-noise ratio gets reduced. The work presented in [27, Chapter 7] explains how the error introduced can be calculated when scaling or quantization rounding off is done. The error is not only introduced by the fixed-point representation of audio samples, but also by fixed-point representation of filter coefficients or other parameters that are multiplied to the samples.

The chosen representation for the audio effects implementation on T-CREST is the fixed-point representation. This is mainly because the hardware multiplier that is available in the processor pipeline is a fixed-point multiplier, and the floating-point multiplication takes much longer to execute, because it is a software library function. Some tests were done to take this decision, where the clock cycles that it takes for each of these operations to execute were measured. The fixed-point multiplication takes 3 clock cycles to finish, while the floating-point multiplication takes around 64. This difference is a key factor for this decision, due to the strict timing requirements for real-time processing.

The works in [28] and [29] provide conversions from floating-point algorithms to fixed-point, so that a programmer can easily write floating-point code and then run it on a fixed-point DSP. However, this is not usable in this project because these conversions are not optimized for real-time processing: they are only valid for off-line processing.

The chosen fixed-point representation is the 16-bit Q0.15 representation, where there is 1 sign bit and 15 fractional data bits. The precision is increased on intermediate operations when multiplications are done, because the results of multiplications are stored in 32-bit registers, which are used as DSP-accumulators. However, when the 32-bit products are added, downscaling is needed to avoid

overflow, so resolution is reduced in those cases. The amount of bits that a value needs to be scaled down depends on the amount of additions done. At the end, the output samples are scaled back to 16-bits.

5.3 Object-Oriented Style Approach for Audio Effects Processing

All the audio effects implemented in this project use an object-oriented style approach: the idea is that all the parameters (filter coefficients, buffers and so on) of each effect are stored as a data structure (similar to a class), so that many instances of it can be created as objects. This means that, for instance, if the user wants to use the delay effect twice in a chain, he/she just needs to invoke 2 instances of the delay structure, and their parameters will be mapped to different memory locations. However, this approach does not mean that all the features of object-oriented languages are available here. Some of them, such as inheritance or polymorphism, are not supported by the implemented audio effect classes. Object-oriented programming is just used as an inspiration.

The audio library for Patmos, *libaudio*¹, contains the data structures and the processing functions of each effect. In the *audio.h* file, the classes for each effect have been defined, and the C `struct` type has been used for this. Inside each `struct`, the parameters of each effect are found. When one of these effects is instantiated, it needs to be placed in the SPM for fast access to the data. The execution time of each effect strongly depends on this. The only data that is not placed in the SPM are the large arrays, needed in the following cases: *a*) when long audio buffers need to be stored (such as for the vibrato, delay or chorus effects), or *b*) when long modulation arrays need to be stored (such as the vibrato or the wah-wah effects). In those cases, only the offset address of the array is located in the SPM. The full array is placed in the external SRAM memory. Appendix B contains some examples of the data structures and processing functions of the effects. One of these structures, the one corresponding to the tremolo effect, is shown in Listing 5.1 as an example, and its parameters will be explained in Subsection 5.4.1.

¹<https://github.com/t-crest/patmos/tree/master/c/libaudio>

```
1 struct Tremolo {
2     int   pnt; //modulation pointer
3     int   pnt_n; //modulation pointer next
4     short frac; //fraction of modulation
5     short frac1Minus; //1 - frac
6     int   mod; //interpolated mod value
7     //SRAM Memory variables
8     int *mod_array; // mod_array[TREMOLO_P]
9     short *frac_array; //frac_array[TREMOLO_P]
10};
```

Listing 5.1: Structure of the tremolo effect.

To place data on the SPM, the `_SPM C` attribute has been used, which maps the specified variables to the local memory address space. The user must take care of mapping it to the correct address to avoid overwriting data. This can be done with the `mp_alloc` function of the message passing library, *libmp*. This function is valid because, in the multicore platform, the same SPM is used for data storage and for the message passing NoC. `mp_alloc` reserves the specified amount of bytes in the SPM and takes care of updating the offset address to the next available position.

5.4 Implemented Audio Effects

After explaining how the effects are structured, it is interesting to show how the setup and processing are done. Each one of the implemented effects has two main functions: an allocation function, which is done during setup time (when the application just starts running on the platform) and a processing function, executed during run-time. The first one has no timing requirements. Some memory space is reserved on the SPM for the effect parameters, and some of them are initialized. If the effect needs to use the external memory, some space is also allocated there using `malloc`. Modulation arrays are also initialized in this stage when needed. The last address available in the SPM is also updated in this function. On the second function, the input samples of the effect are processed with the parameters on the SPM, and output samples are sent to the next effect on the chain or output to the audio CODEC.

As explained previously, the amount of samples that each audio effect processes on a run might be different, as some effects operate on blocks of samples. However, all the effects implemented in this project process one sample per run. There is just one exception for this, which is an effect that processes 8 samples per run, but this one is explained in Chapter 7, as it is only used to show that the multicore platform also supports effects that operate on blocks of samples.

One of the main drawbacks of the implementation of the effects related to WCET is that it has not been possible to use the ISPM at all, because it is not available for the multicore version of the platform (it is only available for the single-core version), so the processing functions of each effect could not be stored there, as it was initially intended. This is also why this project relies so much on the instruction cache: being able to use the ISPM would allow freeing space in the instruction cache, and WCET would be reduced as instruction misses would be avoided. This would be specially noticeable in cold cache misses, which happens in the first execution of an effect, or just after a mode change.

The following sections explain the implementation of each effect, based on the DSP algorithms for audio effects explained in Chapter 2, so one must refer to this chapter to understand what each effect does. For each one of them, the main parameters are mentioned, and the allocation and processing functions are explained. The code of some of these effects is shown in appendix B. The rest can be found in the *audio.c*² file of the *libaudio* library. Here is a list of the implemented effects:

- Tremolo
- Vibrato
- IIR Filters
- Delay
- Wah-Wah
- Chorus
- Overdrive
- Distortion

5.4.1 Tremolo

The tremolo effect is perceived as a periodical volume oscillation. The amplitude modulation of this effect is done using a sinusoidal signal stored in two modulation arrays: one containing the integer part and another one containing the fractional part. This is done to use linear interpolation, as shown in Figure 5.2. This improves the precision of the computation, because if interpolation is not used, a great amount of noise is introduced to the signal. Equation 5.4

²<https://github.com/t-crest/patmos/tree/master/c/libaudio/audio.c>

shows how the value $f(x)$ is interpolated from its closest integer values, and the fractional part $frac$.

$$f(x) = f(i+1) \cdot frac + f(i) \cdot (1 - frac) \quad (5.4)$$

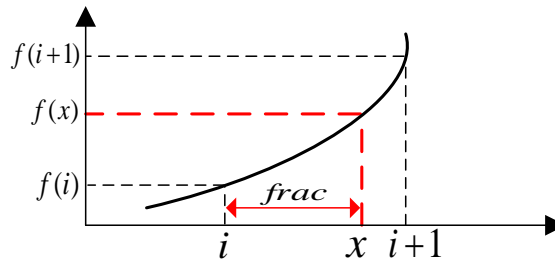


Figure 5.2: Linear interpolation, using integer and fractional parts.

The `struct Tremolo` was already presented as an example in Listing 5.1. The main parameters are the pointers to the integer and fractional modulation arrays, the calculated modulation value, and the offset addresses of the integer and fractional modulation arrays, which are stored in the external SRAM.

In the allocation function, `alloc_tremolo_vars`, some memory space is allocated in the SRAM for the integer and fractional modulation arrays. The length is given by the `TREMOLO_P` macro, which defines the period of the modulation signal. The `storeSinInterpol` function is called, which calculates the modulation array values given some parameters of the sinusoidal signal: the length, the offset and the amplitude. The modulation signal must oscillate between 0 and 1. The modulation pointers are also initialized.

The processing function is the `audio_tremolo` function, has three parts: on the first part, the modulation pointers are incremented. On the second one, the modulation value corresponding to the current sample is calculated by interpolating the current integer and fractional modulation values. Finally, the output sample is the result of multiplying the calculated modulation value ($[0, 1]$ range) with the input sample.

5.4.2 Vibrato

The vibrato effect is perceived as a periodical variation of the pitch of the sound. This effect has an audio buffer which is as long as the maximum vibrato length. As the tremolo, it uses linear interpolation, but in this case the modulation value is not multiplied to the sample: it is used as an oscillating pointer to the audio buffer, so the output sample comes with a small time-varying delay.

The main parameters the `struct Vibrato` are again integer and fractional modulation pointers and offsets. Additionally, an audio buffer is needed in the SRAM, which is as long as the maximum delay, which is defined by the `VIBRATO_L` (buffer length) macro. Finally, 2 accumulator registers are also needed, for the left and right audio channels.

The allocation function is called `alloc_vibrato_vars`, and it does similar steps to the tremolo allocation function, except that the audio buffer is also allocated in the SRAM and initialized to zero. The sinusoidal modulation signal oscillates between zero and the `VIBRATO_L` value.

The processing function is called `audio_vibrato`. It calculates the modulation value as in the tremolo processing function, but instead of interpolating between modulation values, interpolation is done between samples of the audio buffer. The 32-bit accumulator registers are used to hold the intermediate interpolation values.

5.4.3 IIR Filters

The IIR filters are used to implement equalization filters, which subtract some frequency components from the original sound. The C code of this effect is shown in appendix Section B.1. As explained in Chapter 2, 4 types of IIR filters have been implemented: HP, LP, BP and BR. All of them are 2nd order IIR filters and have the same structure.

The main parameters of the `struct Filter` are the input and output circular audio sample buffers (with a length of 3 samples for 2nd order filters), the `A` and `B` arrays containing the filter coefficients, a pointer to the buffer position and a type variable to know which one of the 4 filter types it is. Additionally, there is a `shiftLeft` variable: this one is used to know how many positions the output sample needs to be shifted left, due to possible overflow caused by fixed-point arithmetic operations when the coefficients are too large.

The `alloc_filter_vars` takes the cut-off frequency and Q (or bandwidth for BP/BR) values, and uses the `filter_coeff_hp_lp` or `filter_coeff_bp_br` to calculate the *A* and *B* coefficient arrays, depending on what type of filter it is. This calculation is done following equations in Table 2.1. The coefficients will usually be between -1 and 1, but if they are outside this range, all of them are scaled down until they are within the range, and the `shiftLeft` variable is also incremented on each scaling, indicating that output samples need to be scaled up after processing. Each incrementation of the `shiftLeft` value will result in a resolution loss due to the dynamic range, which wouldn't happen if floating-point processing was used, as explained before. In addition, the buffer pointer is also initialized to the maximum value (2) here.

The `audio_filter` function first places the input sample in the small audio buffer. It then calls the `filterIIR_2nd` function, which implements a 2nd order IIR filter. The way it does it is that it multiplies each input and output sample with its coefficient, and keeps storing the additions of each product in the accumulator register, shifting the result 2 positions to the right after each addition due to a maximum possible overflow of 2 bits. After that, the function checks if the resulting output sample is outside of the acceptable digital audio range, $[-1, 1]$. If this is the case, it simply clips the output to the maximum value. The output result is then stored as a 16-bit sample in the SPM, scaling correctly according to the `shiftLeft` value. Finally, for the BR/BP filters, the output value needs to be added/subtracted from the input value.

5.4.4 Delay

The delay effect is perceived as a repetition of the sound, similar to an echo. The C code of this effect is shown in appendix Section B.2. The delay effect is a 1st order IIR comb filter. It works in a similar way to the IIR filters, with some differences.

The main parameters of the `struct IIRdelay` are the accumulators, and the gains and delay values of each comb filter branches (2 for a 1st order filter). A circular audio buffer is also needed, which is stored in the SRAM, and its offset address and pointer are stored in the SPM.

The `alloc_delay_vars` function initializes the gains and delay values. The delay value is given by the `DELAY_L` value, which also sets the length of the audio buffer stored in the external memory. The audio buffer is initialized to zero, and its pointer is initialized to the maximum value.

The `audio_delay` function calls the `combFilter_1st` function, which takes the

input sample and the delay buffer to calculate the output sample. It uses the accumulator to store intermediate products. Clipping is also done, if necessary, as in the `filterIIR_2nd` function. The pointers are also decremented on each run.

5.4.5 Wah-Wah

This effect is basically a BP filter with time-varying cut-off and bandwidth values. The C code of this effect is shown in appendix Section B.3.

The `struct WahWah` has the same parameters as the `struct Filter`, with the addition of multi-dimensional A and B arrays of coefficients, which are stored in the external SRAM in the `alloc_wahwah_vars` allocation function. They are calculated from sinusoidal modulations of F_C and F_b . Pointers are also needed for the A and B positions corresponding to the current iteration.

The `audio_wahwah` function is very similar to `audio_filter`, with the exception that the values of the A and B arrays are updated on each iteration to achieve time-varying F_c and F_B values. The output signal is also mixed with the original input signal, achieving a combination of both.

5.4.6 Chorus

This effect is the result of adding time-varying delays to the original signal, which are perceived as multiple sources playing simultaneously with some time and pitch differences. The chorus effect is a 2nd order FIR comb filter with time-varying delay times on the cascaded signals. There are many possibilities to implement this variation: low-frequency noise can be used as a modulation signal. However, in this project a separate low-frequency sinusoidal signal has been used for each cascaded branches.

The `struct Chorus` is similar to the `struct IIRFilter`, except that the gain and delay arrays have one more value (3 for a 2nd order filter). Additionally, there are 2 sinusoidal modulation arrays, one for each cascaded branch, and the audio buffer has length `CHORUS_L` which needs to be as long as the maximum delay. These modulation signals and the rest of the parameters are initialized in the `alloc_chorus_vars` function.

The `audio_chorus` function is similar to the `audio_delay` function, except that the function called is `combFilter_2nd`. But before calling that function, the

delay values stored in the SPM are updated according to the current position in the modulation array.

5.4.7 Overdrive

The overdrive is the first one of the waveshaping effects implemented. The C code of this effect is shown in appendix Section B.4. The overdrive is quite a unique effect, as the processing done depends on the amplitude of the input signal, as Equation 2.10 shows. Therefore, the execution time also depends on this, as some processing parts are more complex than others.

The `struct Overdrive` is very simple, as it only requires stereo accumulators for intermediate calculations. The `alloc_overdrive_vars` is again extremely simple as there is no initialization or memory reservation required.

The `audio_overdrive` function is what makes this effect different, because the effect is different depending on the amplitude of the input sample. If the amplitude is in the $[0, 1/3]$ or $[2/3, 1]$ ranges, the processing is very simple as almost no computation is required. However if it is in the range $[1/3, 2/3]$, many arithmetic operations are needed, which increase WCET.

5.4.8 Distortion

The distortion is a non-linear effect, which adds more harmonic content and compression than the overdrive. The implementation of the distortion effect can be considered quite similar to that of the overdrive in the $[1/3, 2/3]$ range.

The `struct Distortion` is simple, it only contains accumulators and some parameters that specify the amount of distortion and the scaling required. The `alloc_distortion_vars` just calculates the distortion parameters given a distortion amount.

The `audio_distortion` simply performs the arithmetic operations given by Equation 2.13, and uses the accumulators for intermediate operations.

CHAPTER 6

Design of Multicore Audio Processing Platform

This chapter presents the considerations and steps taken to design the audio processing T-CREST platform, together with a latency estimation of the system. Here, the effects that were presented in Chapter 5 are put together in the multicore platform, combining the processing power of multiple Patmos processors with the communication resources provided by the Argo NoC. Therefore, this chapter represents a contribution both to the T-CREST project and to real-time multicore audio processing in general.

Multi-processor architectures are very common nowadays, but it is not always trivial to take advantage of the parallelism available in those architectures for audio processing, due to the sequential character of many algorithms [1]. Some of the most popular software environments for computer music have a mainly sequential behavior: parallelism is usually exploited by running copies of the algorithms on multiple threads distributed in the platform. This behavior is also given in this project, as multiple effects are connected one after another forming chains, so the sequential dependencies among them are clear. However, as it will be shown, **computational parallelism** is achieved with a pipeline-style approach.

The work presented in [30] discusses the use of local and shared memories for

multicore audio processing with their respective advantages, and mentions a message passing interface to transfer data explicitly between two cores. The message passing is implemented in this project using the Argo NoC, which provides faster communication than shared off-chip memory, and allows data transfers to be overlapped with processing.

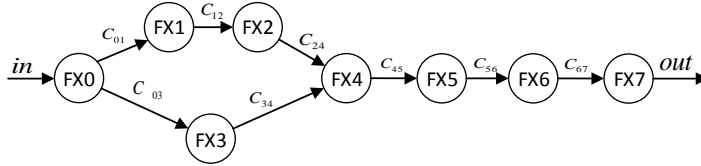
Section 6.1 briefly discusses task allocating for multicore audio applications, and presents the simple static task allocation algorithm used in this project. After that, Section 6.2 explains the advantages of using a NoC for message passing in real-time audio processing. 6.3 is the most important section of this chapter, as it presents the rules that must be followed to achieve correct synchronization and communication between cores in the multicore platform. Section 6.4 then discusses the main parameters of the Argo NoC, and finally, Section 6.5 explains how the overall latency of the system is calculated for the multicore platform.

6.1 Static Task Allocation

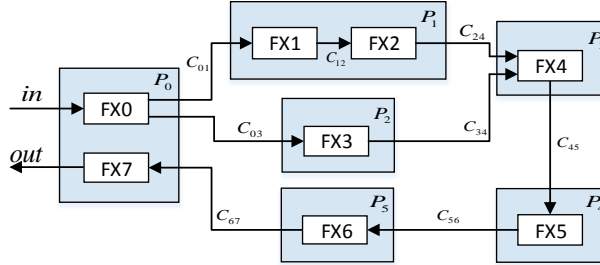
The computation of audio effect chains in a multicore processing platform can be done in various ways. In some cases, multiple cores are needed to process one single audio effect, due to the heavy computation required. This is not the case in this project, as all the effects are processed in real-time in a single core. It might even be possible to process more than one effect in the same core in some cases. Therefore, the problem faced here is about mapping audio effect units to cores efficiently: the effects must be distributed in the platform in a way that the use of computational resources (available Patmos cores) is optimized, and the exchange of data between them must be constrained in time so that the overall latency of the platform is kept within 15 ms, as discussed.

The distribution of effects into cores can be non-trivial in multiple effect setups with complex communication requirements. It is similar to the problem of assigning tasks in a Task Communication Graph to processing nodes in the multicore platform [13]. An example of this is shown in Figure 6.1, where each effect in the audio system corresponds to a task. The tasks are connected to each other forming chains, and parallel chains might appear, such as the ones in Figure 6.1 where two effects are processed in parallel and then merged together on FX4. Feedback loops will not happen between effects (they already appear in the internal structure of some of them). The assignment of the tasks found in the communication graph to the multicore platform must respect the communication requirements of all effects.

In this project, static task allocation has been used. This means that the map-



(a) Setup of effects (FX) and channels between them (C_{ij}) as a Task Communication Graph.



(b) Core communication graph, showing the effect distribution in processors (P_x) of the multicore platform.

Figure 6.1: Example of statically allocating a Task Communication Graph to a multicore platform.

ping of tasks into cores is done by an off-line allocator, so each effect is always processed in the same core. There are many ways to do static task allocation, some of them very advanced and complex, but what all have in common is that some previous knowledge of the tasks is needed. In the multicore audio processing system, the main parameter that the scheduler needs to know was introduced in Section 5.1: it is the time required to process a sample for an effect n , t_{P_n} . Knowing this value and the sampling frequency, F_s , the scheduler can calculate *utilization* (U) values each effect, which is the processing time of effect n relative to the sampling period (i.e. it corresponds to the amount of time that the processor is not idle when processing this effect), as Equation 6.1 shows.

$$U_n = t_{P_n} \cdot F_s \quad (6.1)$$

The utilization gives an idea of how much of the computational resources of the processor an effect uses, and this value is used by the allocator to decide how many effect it can place in a core. As a simple example, effects FX1 and FX2 in Figure 6.1b could have values of $U_1 = 50\%$ and $U_2 = 35\%$: this means that

they can be mapped to the same core, because the sum of their utilizations is smaller than 100%.

In this project, the static mapping of tasks is done in a simple way, as it is not the main point of focus of this work. Given the list of effects and their communications, the allocator takes each effect one by one in the same order as they appear in the chain, and places it on the next core, if the available utilization of that core is greater than the utilization of the effect (i.e. the effect 'fits' on the core). In this way, the amount of NoC channels required gets minimized, because the order of the effects in the chain is maintained. One important restriction is that the parallel chains must be placed in separate cores, due to simplifications explained in Subsection 6.3.2.

There are some other simple algorithms for static task allocation and scheduling, such as greedy algorithms [31] that maximize the usage of computational resources but create more NoC channels, as the effects are not placed in the same order as in the chain.

Complex multicore platforms use elaborate task scheduling methods to solve large graph theory problems. In the field of multicore audio processing, the work presented in [8] mentions many existing applications that use dynamic task scheduling (done in run-time) for resource optimization. In [9] and [32], complex algorithms for optimization of task distribution for multicore audio processing in real-time are also explained. A discussion on possible improvements in task allocation and scheduling is provided in Section 8.2.

6.2 Message Passing NoC v.s. Shared Memory Communication

The communication between cores in a multicore platform can be done in many ways, but two main ways have been considered here: message passing NoC communication and shared memory communication.

When shared memory communication is used, there is no need for a message passing NoC at all (the multicore platform uses the memory-tree NoC to access shared memory). The cores exchange data with each other through the shared memory. In order to do this, all the cores need to agree on which memory locations they will use for communication.

The main drawback of shared memory communication is that access times are long (the SRAM memory is an external chip, and data cannot be cached). More-

over, access time is not constant: it depends on how many cores are trying to access the memory simultaneously. For WCET analysis, the worst-case memory access time has to be considered, which assumes that the rest of the cores are trying to access memory at the same time.

In this project, shared memory communication has been used only for those signals with no strict time requirements: this includes the *ready* signals, which all the cores use to indicate that they have already finished the setup functions, and that they are ready to start processing audio. There is also an *exit* signal, which the master core sets to high when the user decides to finish the application. Additionally, other signals of this kind have been added to support mode changes: one of them is the *current mode* signal, which holds the global mode value. The others are *reconfiguration sync* signals, which are used by all cores to indicate that they are ready to start the new mode.

On the other hand, a statically-scheduled NoC that uses TDM scheduling is ideal for real-time applications with strict timing requirements. The Argo NoC provides timing predictability and communication guarantees, so WCET can be calculated to ensure that the signal latency from input to output of the platform is kept below 15 *ms*. This is why the message passing Argo NoC has been used as the communication resource to exchange the audio samples between cores.

6.3 Architecture and Synchronization of the Multicore Audio Processing Platform

This section explains the rules that need to be accomplished to achieve correct communication among multiple effects that are placed on different cores and connected to each other on a multicore platform. At this point, it is expected that the effects have already been allocated to the cores in the platform, so the starting point to define the rules is a system such as the one shown in Figure 6.1b, where the effect distribution is done but no parameters are assigned yet. The rules presented here use the data rates of each effect to *a)* design the communication channels and the send/receive buffers correctly, and *b)* define the steps done in the execution of each effect to ensure the correct functionality and synchronization of the system.

In Subsection 6.3.1, the Homogeneous Synchronous Data Flow is briefly explained, while Subsection 6.3.2 uses the basic concepts of 6.3.1 and defines general rules to design the multicore audio processing platform.

6.3.1 Synchronous Data Flow

Processing of audio effects on a multicore platform is an example of a *Synchronous Data Flow* (SDF) [33] application. The work presented in [34, Chapter 3] explains SDF among some other models of computation. In SDF, the actors in the system have an static order of execution. The actors in the audio processing platform are the effects, and the order of execution is set by the signal flow through the effect chain.

An interesting SDF type for this project is *Homogeneous SDF*, in which the data rate of each actor is static (i.e. each effect always processes the same amount of samples per iteration). Here, it is said that an actor *fires* when there is a *token* on each of its inputs, so it produces a token on each output. In this project, firing means starting the effect computation, and one token is equal to one audio sample.

Audio processing is a *multi-rate SDF model*, where the firing rates of the actors might not be identical: some actors require more than one input token to fire once and produce multiple output tokens. In the audio field, these are the effects that operate on blocks of samples. Taking this into account, *Balance Equations* need to be defined to ensure that each actor can fire and communicate with other actors correctly so that the system can work as intended. Figure 6.2 shows a simple example of this, where actor *A* produces *M* tokens on its output each time it fires, while actor *B* needs *N* tokens on its input to fire.

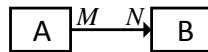


Figure 6.2: Simple example of two actors with different data rates on an SDF model.

The balance equation for this simple situation is shown in 6.2, where q_X indicates how many times actor *X* fires in order to have a constant data rate in the overall system.

$$q_A M = q_B N \tag{6.2}$$

The balance equation for the example of Figure 6.2 looks rather simple, but the complexity grows when there are many actors in the system with multiple data rates, because the equation needs to be fulfilled for each connection between

actors. Moreover, the equation does not have a single solution: there are multiple possible solutions that will have an influence in the design of the system: for the audio processing example, the memory requirements (buffer sizes), the total latency and the sending/receiving overhead will be directly affected by the solution chosen, as it will be shown in Subsection 6.3.2. For a general case, [34, Chapter 3] suggests to use the least positive integer solution (i.e. each actor fires as soon as it has enough input tokens).

6.3.2 Rules for Multicore Audio Synchronization

This subsection takes the basic concepts of multi-rate SDF explained in Subsection 6.3.1 and presents the rules followed in this project for the implementation of the multicore audio processing platform. In audio applications, the parallelism that can be achieved in processing is sometimes limited due to sequential dependencies and feedback loops of the algorithms. A clear case when parallelism can easily be achieved is when processing parallel audio chains that are independent to each other. On the other hand, if the effects are sequential (i.e. the results of effect $i - 1$ are needed to process effect i), computational parallelism is achieved by **pipelines** of processor cores, which provide an increase in the number of effects that can be processed in real-time compared to a single core platform. The audio processing on the T-CREST platform uses a similar approach to the *flow decomposition* approach presented in [32], where a sequence of tasks is executed concurrently by chaining their inputs and outputs through buffers, forming pipelines and parallel chains.

In the T-CREST multicore platform, each one of the effects that is mapped to a core has receive and send buffers of different sizes on its input and output ports, to exchange data with other effects on the same or different cores. The size of these buffers and the amount of times each effect should run depends on the other effects and their data rates. In this section, the minimum required buffer sizes and the execution order are explained for 3 different types of effect setups. It is very important to understand that this solution minimizes the buffer sizes of each effect and the latency of the signal from input to output of the platform, but it maximizes the overhead caused by sending/receiving audio samples. This will be understood better after explaining the rules, in Subsection 6.3.3.

Figure 6.3 shows the most relevant parameters of an effect for synchronization, which are independent of the type of processing done. They are the following:

- S_i : amount of samples that effect i processes on each run (i.e. data rate, or the amount of samples it needs to fire). This value is 1 in most of the

effects implemented in this project. However, the rules presented here are not limited only to those effects, as more complex effects with higher data rates are also considered, to design a scalable and flexible system in terms of data flow.

- $B_{i,j}^t$: size of the buffer at effect i , connected to effect j , measured in samples. t sets if it is a receive or send buffer (i.e. sends to j or receives from j): $t = x, y$ (x and y mean input and output respectively, following the typical DSP notation used in Chapter 2). The size of any buffer of effect i must always be equal or greater than its data rate: $B_{i,j}^t \geq S_i$. This is because the buffer needs to hold at least as many samples as the effect will process once it fires.

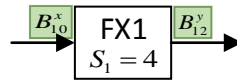


Figure 6.3: Effect $FX1$ with its send and receive buffer sizes, and a data rate of 4 samples per iteration.

Some effects have one send and one receive channel (i.e. are connected to just one effect on their ends), but some others might have more than one:

- **Join** effects have multiple receive channels (Figure 6.4). They connect two or more audio signals together: before processing, the samples of each input buffer are added. In order to simplify the platform, all the receive buffers have the same size, equal to the maximum one required. Also, the join effect requires that the relative latencies of all the chains that it connects are the same.



Figure 6.4: Join effect with 3 receive channels.

- **Fork** effects have multiple send channels (Figure 6.5). They split the signal into different chains. As in the join effect, all the send buffers must have the same size, equal to the maximum one required.

We look at the cores that form the multicore platform now. Each core might process one or more effects, so there are 3 possible setups for a core:

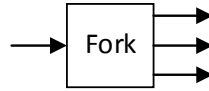


Figure 6.5: Fork effect with 3 send channels.

- **Single effect cores** (6.3.2.1), where the core processes just one effect, so this effect receives and sends data through the NoC.
- **Multiple independent effect cores** (6.3.2.2), where the core processes two or more independent effects, which are not connected to each other: in this case, all effects exchange data using different NoC channels.
- **Multiple effect-chain cores** (6.3.2.3), where the core processes two or more sequential effects which are connected on a chain: in this case, only the first and last effects of the chain inside the core are connected to NoC channels.

The receive and send buffer sizes of each effect depends not only on the effect itself, but also on other effects it is connected to. This way, we can distinguish among 3 possible **Processing Types** of effect i (PT_i), which can be:

- **XeY** if the send and receive buffer sizes of effect i are equal: $B_{i,j}^x = B_{i,k}^y$.
- **XgY** if the receive buffer size of effect i is greater than that of the send buffer: $B_{i,j}^x > B_{i,k}^y$.
- **XlY** if the receive buffer size of effect i is less than that of the send buffer: $B_{i,j}^x < B_{i,k}^y$.

The processing type of an effect is independent of whether the effect is or not a join or fork, because, in those cases, the sizes of all receive or send buffers are the same.

Depending on the data rate (S_i), buffer sizes ($B_{i,j}^t$) and processing type (PT_i) values of an effect, there are 3 parameters which need to be set, because they define how many times each effect receives, sends or fires on a run. A run is a complete iteration of any core in the system: the whole computation piece that is constantly repeated during execution, where all the effects in that core are processed. We therefore say that the effects in a core fire once or many times in every run of the processor. The parameters are listed here, and how they can be calculated will be explained in the next subsections:

- **Receives per run** (N_R) indicates the amount of receive operations (receiving data from the previous effect(s) on the chain) executed on each run of the effect.
- **Sends per run** (N_S) indicates the amount of send operations (sending data to the next effect(s) on the chain) executed on each run of the effect.
- **Firings per send or receive** (N_F) indicates the amount of firings done (execution of the algorithm of the effect):
 - per each send, in effects of type XgY .
 - per each receive, in effects if type XlY .
 - per send and receive, in effects of type XeY (the amount of send and receives done per run is the same).

In the following sections, a set of rules is defined, which explain how these important parameters are calculated for each effect, depending on the core setup.

6.3.2.1 Single Effect Cores

In this setup, each effect is mapped to a different core (i.e. each core processes just one effect). An example of this case is shown in Figure 6.6, which shows a possible setup of audio effects on a multicore platform. The system is composed of a chain of audio effects FX0 to FX5, and each effect is assumed to be processed on a separate core (a to f), so each core has just one effect. Core a corresponds to the master core on the T-CREST platform, which is connected to the audio interface and takes care of the audio input and output. Effects FX1 to FX5 are placed on slave cores. In the following lines, the calculation of the buffer sizes and the N_S , N_R and N_F parameters for effects that are individually placed on each core is shown, and numerical examples corresponding to the S values shown in Figure 6.6 are given for better understanding, interleaved with the rules and shown in blue background.

The rules that apply for single core effects are the following:

1. S_i and $B_{i,j}^t$ must be a power of 2 always (to ensure that any division between them is an integer number).
2. $B_{i,j}^y = B_{j,i}^x$ always. This means that the two buffers located between two sequential effects (i.e. connected to the same channel) must have the same size.

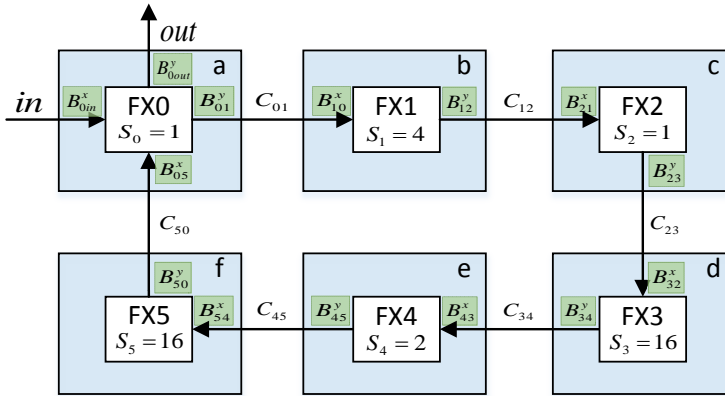


Figure 6.6: Example of a multicore platform processing an audio effects chain, with some given S (data rate) values. All the cores are single effect cores.

3. $B^y_{i,j} = B^x_{i,j} = \max(S_i, S_j)$. This is because effect i must have send and receive buffer sizes at least as large as the amount of samples it processes on each firing.

In Figure 6.6: $B^y_{0,1} = B^x_{1,0} = 4$, $B^y_{1,2} = B^x_{2,1} = 4$, $B^y_{2,3} = B^x_{3,2} = 16$, $B^y_{3,4} = B^x_{4,3} = 16$, $B^y_{4,5} = B^x_{5,4} = 16$, $B^y_{5,0} = B^x_{0,5} = 16$

4. Depending on the processing type of effect i (PT_i) and the buffer sizes of this effect (calculated following rules 1, 2 and 3), which receives audio from j and sends audio to k ($B^x_{i,j}, B^y_{i,k}$):

- If $PT_i = XeY$ ($B^x_{i,j} = B^y_{i,k}$):

- $N_{R_i} = N_{S_i} = 1$
- $N_{F_i} = \frac{B^x_{i,j}}{S_i} = \frac{B^y_{i,k}}{S_i}$

Effects of type XeY are: FX1, FX3, FX4, FX5:

- $i = 1, 3, 5$: $N_{R_i} = N_{S_i} = N_{F_i} = 1$
- $N_{S_4} = N_{R_4} = 1$, but $N_{F_4} = 16/2 = 8$

- If $PT_i = XgY$ ($B^x_{i,j} > B^y_{i,k}$):

- $N_{R_i} = 1$
- $N_{S_i} = \frac{B^x_{i,j}}{B^y_{i,k}}$
- $N_{F_i} = \frac{B^y_{i,k}}{S_i}$, which in this case indicates firings per send.

The only effect of type XgY is FX0:

- $N_{R_0} = 1$
- $N_{S_0} = 16/4 = 4$
- $N_{F_0} = 4/1 = 4$
- In total, the FX0 will execute 16 firings per run: 4 firings per send with 4 sends per receive.

- If $PT_i = XlY$ ($B_{i,j}^x < B_{i,k}^y$):

- $N_{R_i} = \frac{B_{i,k}^y}{B_{i,j}^x}$
- $N_{S_i} = 1$
- $N_{F_i} = \frac{B_{i,j}^x}{S_i}$, which in this case indicates firings per receive.

The only effect of type XlY is FX2:

- $N_{R_2} = 16/4 = 4$
- $N_{S_2} = 1$
- $N_{F_2} = 4/1 = 4$
- In total, the FX2 will execute 16 firings per run: 4 firings per receive, and 4 receives per send.

5. Finally, the steps for each run of effect i are the following (including the buffer read/write position updating instructions, which are not shown here), depending on the processing type (PT_i) and the N_S , N_R and N_F parameters calculated in rule 4:

- If $PT_i = XeY$:
 - Receive once ($N_{R_i} = 1$)
 - Fire N_{F_i} times.
 - Send once ($N_{S_i} = 1$)

Effects 1, 3 and 5 will receive once, fire once and send once.
Effect 4 will receive once, fire 8 times and send once.

- If $PT_i = XgY$:
 - Receive once ($N_{R_i} = 1$)
 - Repeat N_{S_i} times:
 - * Fire N_{F_i} times.
 - * Send once

Effect 0 will receive once, and then { fire 4 times, send } 4 times.

- If $PT_i = XIY$:
 - Repeat N_{R_i} times:
 - * Receive once
 - * Fire N_{F_i} times.
 - Send once ($N_{S_i} = 1$)

Effect 2 will { receive, fire 4 times } 4 times, and then send.

The steps for each run are quite simple for the XeY effect type, as receiving and sending happens just once per run, so the send/receive rate is constant. This does not happen for the XgY and XlY types, so the steps that need to be executed are not so straightforward. These types of effects could be avoided if all the effects in the system were designed as XeY types, as they would all process the same amount of samples per run. This is feasible when the data-rates of the effects in the system are similar, as the latency increment introduced by increasing some buffers is not much. However, when there are big differences in the data-rates of each effect, this might be unfeasible, because considering all the effects as XeY will increase the latency notably. One can imagine a setup of effects, where one of them needs 512 samples to fire. If the buffers of the rest of effects are increased to match this value, the latency of the system will increase and the real-time perception might be lost. Nevertheless, the proposed solution using XgY and XlY effect types would keep the latency within an acceptable time interval on those cases. Another possible solution would be to reduce the amount of buffers of all the effects, and have larger internal buffers in those effects with higher samples per firing. In that case, the latency would be the same as in the proposed solution, but the overhead introduced in the system due to receiving and sending would be larger for those effects.

In Subsection 7.1.1, it will be shown how the master core (a in the example) is quite special (it is the first and last of the chain and must take care of the signal latency), but for now it is easier to assume it works the same way as the others. The rules shown in this section were applied to an example where there is just one audio signal chain, but the same rules apply if there are multiple parallel audio chains on single effect cores. In that case, there will be join and fork effects. For a join effect, a receive operation means receiving on all its receive channels, and for a fork effect, a send operation means sending to all its send channels.

6.3.2.2 Multiple Independent Effect Cores

Sometimes, for resource optimization purposes, many effects might be allocated on the same core, as explained in Section 6.1. In this case, the effects might be connected to each other or not. In multiple independent effect cores, the effects mapped to this core are not connected to each other: they both receive and send from/to different channels of the NoC. Figure 6.7 provides an example of this case.

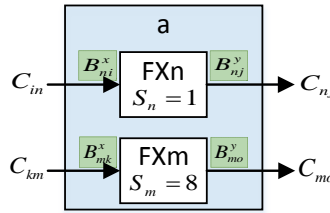


Figure 6.7: Example of core a processing two independent effects n and m , with some given S values.

The setup presented in Figure 6.7 is the following: effects n and m are mapped to the same core, a . Effect n receives audio samples from effect i through NoC channel $C_{i,n}$ and sends to effect j through $C_{n,j}$, while effect m receives from effect k through NoC channel $C_{m,k}$ and sends to o through $C_{m,o}$. If these two effects n and m were implemented on separate cores, the rules defined in Subsection 6.3.2.1 would be considered, so the following would apply:

- $B_{n,i}^x = \max(S_n, S_i)$
- $B_{n,j}^y = \max(S_n, S_j)$
- $B_{m,k}^x = \max(S_m, S_k)$
- $B_{m,o}^y = \max(S_m, S_o)$

In Figure 6.7, we assume $S_i = 4$, $S_j = 2$, $S_k = 1$ and $S_o = 32$. This gives:

- $B_{n,i}^x = 4$
- $B_{n,j}^y = 2$
- $B_{m,k}^x = 8$
- $B_{m,o}^y = 32$

The amount of samples that are processed on each effect on a run of the processor is always given by the largest value between its receive and send buffers. In this case, the values is 4 for n and 32 for m . Placing effects on the same core means that, for each run, each one of them needs to process the same amount of samples: otherwise, the data rates of the effects would be unbalanced, and this is unwanted (it would only make sense if the effects had different sampling rates). Therefore, if two or more effects are processed on the same core independently, in order to calculate their receive and send buffer sizes, first the largest one of them needs to be identified, and then the size of it might be modified: it must take into account the S values of all the effects in the core and all the effects connected to the core. This means that the largest buffer of both effects must have the same size, to achieve the same data rate per run.

Going back to the setup of Figure 6.7, for core a which processes effects n and m , the following rules can be defined:

- $\max(B_{n,i}^x, B_{n,j}^y) = \max(S_i, S_k, S_n, S_m, S_j, S_o)$
- $\max(B_{m,k}^x, B_{m,o}^y) = \max(S_i, S_k, S_n, S_m, S_j, S_o)$

For the S values given before:

- $\max(B_{n,i}^x, B_{n,j}^y) = B_{n,i}^x = 32$
- $B_{n,j}^y = 2$, as before
- $\max(B_{m,k}^x, B_{m,o}^y) = B_{m,o}^y = 32$
- $B_{m,k}^x = 8$, as before

Now, the amount of samples processed by each effect on a run is the same, 32.

It is important to mention that this will also affect the buffer sizes of effects that are connected to this core, which makes the buffer size calculation an iterative process.

Following with the example, the send buffer of core i must increase: $B_{i,n}^y = 32$, instead of 4.

Once the buffer sizes are defined, the same rules as before apply: the N_{R_i} , N_{S_i} and N_{F_i} parameters of each effect and the execution order can be calculated in the same way as in rules 4 and 5 of Subsection 6.3.2.1.

6.3.2.3 Multiple Effect-Chain cores

Another possible setup is to have multiple effects connected to each other on a single core: this is desired when possible, because it avoids sending data through the NoC, and therefore reduces the overall latency of the audio processing system.

To explain this case, Figure 6.8 is presented, where effects n and m are connected on a chain inside core a .

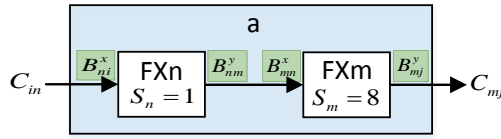


Figure 6.8: Example of a core processing two effects n and m belonging to the same audio chain, with some given S values.

To proceed in the same way as before, first of all the buffer sizes will be calculated assuming the effects were placed in different cores, which would mean:

- $B_{n,i}^x = \max(S_n, S_i)$
- $B_{n,m}^y = B_{m,n}^x = \max(S_n, S_m)$
- $B_{m,j}^y = \max(S_m, S_j)$

In Figure 6.8, we assume the following values: $S_i = 4$ and $S_j = 16$. This gives:

- $B_{n,i}^x = 4$
- $B_{n,m}^y = B_{m,n}^x = 8$
- $B_{m,j}^y = 16$

But, as explained in Subsection 6.3.2.2, this would break the balance as the effects would have different data rates per run. Moreover, if the effects on a setup like this are of processing type XgY or XlY (i.e. have different receive and send buffer sizes), the execution of them needs to be interleaved, otherwise some data might be overwritten in the intermediate channel. This is too complex

to be implemented in the audio processing T-CREST platform: in this project, when there are multiple effects mapped to the same core, the execution of them is sequential (i.e. the full processing of one must finish before the next one can begin). So, to simplify the platform, all the effects on an effect-chain core are of type XeY (same send and receive buffer sizes). This means that all effects perform send and receive operations just once per run.

To match all the buffer size values, the largest data rate of the effects in the core and the effects connected to it are taken:

- $B_{n,i}^x, B_{n,m}^y, B_{m,n}^x, B_{m,j}^y = \max(S_i, S_n, S_m, S_j)$

This results in:

- $B_{n,i}^x, B_{n,m}^y, B_{m,n}^x, B_{m,j}^y = \max(4, 1, 8, 16) = 16$

So now it is clear that each one of the effects will process 16 audio samples on each run. The N_{R_i} , N_{S_i} and N_{F_i} values and the execution order can be then determined as explained in Subsection 6.3.2.1 for effects of type XeY .

6.3.3 Reducing Send/Receive Overhead

As stated before, the rules proposed in Subsection 6.3.2 minimize the buffer sizes and the latency of the audio signal. However, the overhead caused by transferring data through the channels can be high in this case. The effects on different cores use message passing for communication, implemented in the T-CREST platform as the *mplib* library with the `mp_send`, `mp_recv` and `mp_ack` functions. These functions are part of the processing of each effect, so the more often they are called, the more overhead is introduced.

Small buffer sizes mean high communication overhead (i.e. less samples are transferred, thus sending needs to be done more often). An easy solution to reduce this overhead is to increase the send and receive buffers, which will also increase the latency of the signal, as it will be demonstrated in Section 6.5. Therefore, a balanced compromise needs to be found between the buffer sizes and the communication overhead introduced. The buffer sizes shall then be increased if there is enough memory available for buffering (SPM in Patmos), always taking into account that the overall latency of the system must be kept under 15 ms.

As a simple example, if, after following the rules, a given input buffer size is $B_{i,j}^x = 8$, it can be multiplied with an overhead reducing factor, for instance 4, so $B_{i,j}^x = 32$. This means that the amount of receive operations done gets divided by 4, but the latency introduced by this effect is 4 times larger.

6.4 Message Passing NoC Parameters

The buffer sizes of each NoC channel are given by the rules in 6.3.2, with some possible overhead reducing factors applied to increase them. The hardware resources of the NoC need to be shared among all the NoC channels according to the communication requirements of the application. In the Argo NoC, this is done by creating a custom NoC schedule to assign bandwidth only to those channels needed by a given effect setup.

The NoC channel bandwidth is defined as the amount of packets per TDM period, and the *phits* parameter determines the amount of words in a packet. These concepts were introduced in Subsection 3.4.1. In this case the *phits* value has been set to 3, which is the recommended value. This means that each packet is composed of the header, one audio sample (2x16-bit) (or one acknowledge signal) and one flag (used by the message passing library to indicate if the message was read or not). So, one audio sample travels on each packet.

If the bandwidth value is increased, there are more packets in a TDM period, which make the period longer. It is clear for the audio processing T-CREST platform that the bandwidth values assigned to each channel need to be equal, because the sampling rate of the system is constant, which means that, on average, one sample travels on a NoC channel every sampling period. This is independent of the buffer sizes: if a channel has a buffer of one sample, it will transfer one sample every sampling period, while if it has a buffer of 8 samples, it will transfer 8 samples every 8 periods. The difference is on how the samples will be distributed in the NoC packets (which will affect the NoC latency), but the bandwidth required is the same. It is also clear that the bandwidth value should never be greater than the smallest buffer size in the system: if this happened, empty slots would be found in the TDM period, as the channel will never have as many samples as the bandwidth assigned to it. So the smallest buffer size is an upper limitation when choosing the bandwidth value.

Additionally, NoC channels are also required in the inverse direction of the audio signal flow, used for acknowledge signals, but these ones can have the smallest possible bandwidth value, 1, as they do not transfer any data. Finally, if NoC reconfiguration is enabled, the NoC scheduler creates additional channels

between the master and the slave cores, to issue the reconfiguration signal.

The Argo NoC also supports multiple buffers on each channel. These buffers act as a FIFO, allowing a sender to send data again to the next available buffer, even if the receiver did not receive the previous data yet. In the audio processing platform, the Argo NoC transfers data very fast compared to the time between send operations of each core: this means that, when a sender core computes some data and sends it to the NoC channel, the previous data will have already arrived at its destination. However, there are some cases when data can get accumulated on a buffer, for instance on a cache miss. That is why having multiple send and receive buffers enhances the elasticity of the system against cache misses even more, together with the effect of input and output buffers as explained in Subsection 5.1.1.

A situation which requires a channel to have more send and receive buffers is when the effect connected to it has different send and receive buffer sizes (i.e. is of type XgY or XlY). This is because this type of effect will have different send and receive rates, so more buffering is required to prevent processors from blocking due to not having enough send/receive buffers available.

6.5 Audio Processing Latency on a Multicore Platform

The audio signal latency from input to output in the audio processing T-CREST platform is proportional to the amount of processing applied to it. It can be calculated by knowing the buffer sizes of each effect, and how the effects are connected to each other (through the NoC or in the same core). In Subsection 5.1.2, the latency of the single-core processing system was calculated, which was equal to its input and output buffer sizes, N . For multicore setups as the ones described, the overall latency L measured in samples is the sum of three values: the latency added by the input/output audio buffers (L_{IO}), which is the same as for the single core platform, equal to N ; the latency due to the effect buffers (L_{FX}) (6.5.1) and the latency of transferring data through the NoC (L_{NoC}) (6.5.2). This is shown in Equation 6.3. We recall the tolerable latency range given in Subsection 5.1.2, which was 781 samples, corresponding to 15 ms. The total latency of the system must then stay below 781 samples.

$$L = L_{IO} + L_{FX} + L_{NoC} \quad [samples] \quad (6.3)$$

It is important to point out that the latency value calculated, L , is the latency of the system, and does not consider latencies possibly introduced by some effects, as it could happen for instance in a big FFT. The effects designed for the T-CREST platform have zero or almost-zero latency, therefore this one is considered negligible.

6.5.1 Latency Added by the Effect Buffers

Buffering samples in send/receive ports of an effect adds latency to the audio signal, and the more effects are chained, the longer the latency. When parallel audio signal chains are implemented, the latency is equal to the longest one of the chains. However, as previously explained, the implemented audio processing platform requires that all the parallel chains have equal buffer size values, so it does not matter which one is considered.

The latency does not depend on the execution time of each effect because it is assumed that this one is always under the sampling period (except for the cache miss cases, as explained). That means that the processing of each sample gets synchronized with the sampling frequency, which allows being able to measure latency in sample units.

For a chain (or chains) of effects with different data rates and receive and send buffer sizes, it has been found that the latency is the sum of send buffer sizes of all the effects (except for effect-chain cores, where only one send buffer needs to be considered for all the effects in the core, and the size of it is the same for all of them). This is because each effect can send data as soon as its send buffer is full, which means that it has processed as many samples as the size of the send buffer. At this point, the next effect in the chain receives the samples and can start processing. The processing might be done faster in some cases, but the system always gets synchronized to the sampling rate. This latency L_{FX} is shown in Equation 6.4, for an audio effects chain of N effects (effect-chain cores are taken as a single effect, as explained), indexed $0, 1, 2, \dots, N - 1$.

$$L_{FX} = B_{0,1}^y + B_{1,2}^y + B_{2,3}^y + \dots + B_{N-2,N-1}^y \quad [\textit{samples}] \quad (6.4)$$

The relation between the buffer sizes and the latency can be better understood now. It is also helpful to provide a numerical example in this case. Figure 6.9 is based on the effects setup found in 6.1b, and the buffer sizes have been calculated following the rules explained in 6.3.2.

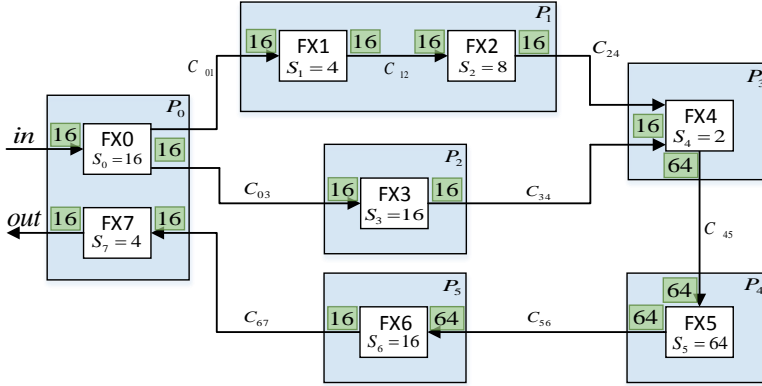


Figure 6.9: Effect setup of Figure 6.1b, with some given S data rate values for each effect, and buffer sizes as calculated following the rules.

The latency added by the effect buffers in Figure 6.9 is the sum of the output buffers of each effect: $16 + 16 + 64 + 64 + 16 + 16 = 192$ samples (in the parallel chains of P_1 and P_2 just one of them was considered). This is easy to check if the path of the samples through the system is analysed: P_0 will send 16 samples to P_1 and P_2 when it has processed them, so the latency added by P_0 is 16 samples; in the same way, P_1 and P_2 need to process 16 samples to fill in their send buffers, so the accumulated amount is 32 samples; the latency added by P_3 is even greater, as it needs to process 64 samples before its send buffer is full and it is ready to send; Following the chain, FX7 in P_0 will output 16 samples after processing. The accumulated latency is 192 samples.

6.5.2 Latency Added by the NoC

The TDM period (P_{TDM}) in the Argo NoC is proportional to the amount of channels and bandwidth values required. At this point, it is interesting to recall the worst-case packet latency term introduced in 3.4.2, $L_{P_{wc}} = P_{TDM} + 8$, measured in clock cycles. The NoC latency is proportional to $L_{P_{wc}}$.

The latency of a NoC channel $C_{i,j}$ between effects i and j on different cores, $L_{C_{i,j}}$, is calculated shown in Equation 6.5, where $BW_{i,j}$ is the bandwidth of the channel measured in samples and $B_{i,j}$ is the buffer size assigned to the channel. The equation gives an idea of how many TDM periods are required for the full buffer to get transferred from source to destination. This latency value is measured in the same unit as the packet latency, clock cycles. To convert it to

samples, the amount of clock cycles per sample period must be known.

$$L_{C_{i,j}} = \frac{B_{i,j}}{BW_{i,j}} \cdot L_{P_{wc}} \quad [\text{clock cycles}] \quad (6.5)$$

To provide a simple example, if a channel has a buffer of 16 samples and a bandwidth of 2 samples, it means that 2 samples are assigned to each TDM period, so it will take 8 periods for the full buffer to arrive at the destination. To calculate the overall latency added by the NoC in a chain of audio effects, the latencies of all the NoC channels need to be added. This is shown in Equation 6.6 for a set of N effects ($0, 1, 2, \dots, N-1$) that form a chain of effects, and all the connections between them are NoC channels. The value T_{CC} is the sampling period measured in clock cycles, which allows converting the NoC latency value from clock cycles to sample units. If there are parallel chains, just one of them needs to be considered, as the buffer sizes are equal in all chains.

$$L_{NoC} = \left\lceil \frac{L_{C_{0,1}} + L_{C_{1,2}} + L_{C_{2,3}} + \dots + L_{C_{N-2,N-1}}}{T_{CC}} \right\rceil \quad [\text{samples}] \quad (6.6)$$

The L_{NoC} value shown is a worst-case value, because the worst-case packet latency was considered to calculate it. In many cases, this value will be under one sample (because the sum of NoC channel latencies is below one sampling period). But even in this case, a latency of 1 sample should be considered. In general, the L_{NoC} value should be quantized to its closest upper integer.

As an example, the setup presented in Figure 6.9 can be considered again. The values of the NoC parameters can be assumed, for instance, as a bandwidth of 1 sample per packet, with a worst-case packet latency of 18 clock cycles. Following Equation 6.5, the worst-case latency added by each channel of the system can be determined. The values are calculated for all the channels in the system shown in Figure 6.9:

$$L_{C_{01}} = L_{C_{24}} = L_{C_{67}} = (16/1) \cdot 18 = 288 \text{ clock cycles}$$

$$L_{C_{45}} = L_{C_{56}} = (64/1) \cdot 18 = 1152 \text{ clock cycles}$$

Channels C_{03} and C_{34} are not considered as one of the parallel chains has already been considered. Channel C_{12} is neither considered as it is not a NoC channel. If the latencies of all channels are added, we get the total NoC latency in clock cycles, according to Equation 6.6:

$$L_{NoC} = 288 \cdot 3 + 1152 \cdot 2 = 3168 \text{ clock cycles}$$

This value can be then divided with the sampling period measured in clock cycles, which in the current platform is calculated by dividing the processor clock frequency, 80 *MHz*, with the sampling frequency of 52.083 *kHz*. The value is 1536 cycles per sample. So the NoC latency measured in samples is only $\lceil 3168/1536 \rceil = 3$ samples.

CHAPTER 7

Implementation and WCET Analysis of the Platform

This chapter shows the implementation of the audio processing platform and the WCET analysis performed to guarantee its real-time functionality. Here, the individual effects presented in Chapter 5 are combined in the multi-processor system, forming sequential and parallel audio effect chains. The effects are allocated into cores following the concepts and rules explained in Chapter 6. As previously explained, the platform used for implementation is T-CREST, that is why this chapter represents a contribution to the T-CREST project.

Section 7.1 presents the architecture and technical details of the chosen audio processing platform implementation. First, how the system takes care of the signal latency is explained, and then the software architecture is shown, in the form of C data structures and functions. Section 7.2 shows both WCET analysis and experimental execution time measurements done for each effect. The results of the experimental measurements are used by the implemented effect allocator to decide how to map audio effects to cores. The allocator is also presented here.

7.1 Architecture and Technical Details

The chosen T-CREST platform implementation is the 2-by-2 bitorus topology with 4 Patmos processors, as the one previously shown in Figure 3.1, running on the Altera DE2-115 FPGA board. There is a master core, which is a Patmos processor connected to the audio interface component as an I/O device, and 3 slaves, standard Patmos cores. Therefore, the master core is in charge of audio input/output, and the slaves are only used to process effects. All the cores have the same cache and SPM sizes. The instruction cache has proved to be the main bottleneck of the system, as its size and associativity values needed to be increased to be able to compute all the effects in real-time. This is the main reason why a platform with more cores has not been used: the large associativity value of the instruction cache is a limitation for meeting the timing requirements in the FPGA platform.

Subsection 7.1.1 first explains how the master core takes care of the signal latency, a concept already discussed in Section 6.5. After that, the architecture of the chosen implementation is described in Subsection 7.1.2, where the general data structure of the effects is shown, and the main setup and audio processing functions are overviewed.

7.1.1 Master Core Latency

In the current implementation of the platform, the master core needs to have prior knowledge of the latency of the signal before audio processing starts. The first and last effects of the chain are always placed in the master core, because it is here where audio is input and output to and from the processing system. This means that the audio signal needs to travel through the whole platform between its entry and exit points, so it is important to show how the master core handles this situation.

The master core can be seen as a multiple independent effect core (Subsection 6.3.2.2), because it contains at least two effects (first and last) independent to each other, connected to components that are external to the core. The amount of samples processed in each iteration of the core is then equal for all of its effects. Additionally, all the effects mapped to this core are simplified as XeY effects, which means that they all have the same send and receive buffer sizes. This makes taking care of the signal latency easier.

Obviously, the master core cannot start outputting audio at the same time as the first sample is input into the system, because there is nothing to receive on

the last effect of the chain yet, so it would get stuck for a long time after calling the `mp_recv` function. Therefore, the master must wait until the signal has travelled through all the cores in the chain until the last effect can receive and output. In the current implementation, the master retrieves the latency value from one of the header files generated by the effect allocator, which contains the latency due to the effect buffers and the NoC, calculated using equations shown in Subsections 6.5.1 and 6.5.2 (the latency added due to the I/O buffers is not part of the core system, but of the audio interface component). But the latency stored in the header is not measured in samples, but in iterations or runs of the master core (amount of times it fully processes all the effects on it), so that it knows after which iteration the audio signal can be output. This is shown in Equation 7.1, where the latencies are added and divided by the amount of samples processed per run by the master core, which is equal to any buffer on it, B_M , as all the effects are XeY . The value is quantized to its closest upper integer.

$$L_M = \left\lceil \frac{L_{FX} + L_{NoC}}{B_M} \right\rceil \quad [iter.] \quad (7.1)$$

As a simple example, we take a chain of N effects, where 0 (first) and $N - 1$ (last) are located in the master core. We assume a buffer size for these two effects of 16 samples; the total latency of the effects is 48 samples and the one of the NoC is 3 samples. According to Equation 7.1, $L_M = \lceil (48 + 3)/16 \rceil = 4$ iterations. This means that the master core will process only effect 0 for the first 3 iterations, and it will start processing effect $N - 1$ on the 4th iteration. Therefore, the master core will add a latency of a few samples to the system: even if the samples are received with a latency of 51, it needs to wait for a value proportional to its buffer size, in this case $16 \cdot 4 = 64$, to start outputting data. This is due to the technical characteristics of the chosen implementation for the platform, where the master core cannot start outputting samples in the middle of an iteration. This means that the actual latency of the system in the example is 64 samples, and not 51. Those extra 13 samples are buffered between effects.

7.1.2 Architecture of the Implementation

The individual effect structures and functions explained in Sections 5.3 and 5.4 are the base for the multicore platform implementation: here, a data structure that is general for all effects is created on top of the structures previously mentioned, which is called `struct AudioFX` and explained in Subsection

7.1.2.1. The same object-oriented style approach has been used for this implementation. Similarly, the effect setup and audio processing functions, called `alloc_audio_vars` and `audio_process` respectively, are also built on top of the functions of each individual effect. These two are overviewed in Subsections 7.1.2.2 and 7.1.2.3. The full C implementation of these structure and functions can be found in the `audio.c` and `audio.h` files of the `libaudio` library of Patmos¹.

7.1.2.1 The AudioFX Structure

The `struct AudioFX` uses an object-oriented style approach, which allows instantiating audio effects as objects. It stores generic parameters of the effect (ID, effect type, core number, buffer sizes, connections...), which need to be set following the rules explained in Section 6.3. The structure can be found in Listing 7.1.

```

1 //type of connection: first last, to NoC, or to same core
2 typedef enum {FIRST, LAST, NOC, SAME} con_t;
3 // comparison of receive/send buffer sizes
4 typedef enum {XeY, XgY, XlY} pt_t;
5 // possible effects:
6 typedef enum {DRY, DRY_8S, DELAY, OVERDRIVE, WAHWAH,
7              CHORUS, DISTORTION, HP, LP, BP, BR,
8              VIBRATO, TREMOLO} fx_t;
9
10 struct AudioFX {
11     //effect ID
12     _SPM int *fx_id;
13     //core number
14     _SPM int *cpuid;
15     //connection type
16     _SPM con_t *in_con;
17     _SPM con_t *out_con;
18     //amount of send and receive channels (fork or join effects)
19     _SPM unsigned int *send_am;
20     _SPM unsigned int *recv_am;
21     //pointers to SPM data
22     _SPM unsigned int *x_pnt; //pointer to x location
23     _SPM unsigned int *y_pnt; //pointer to y location
24     //receive and send NoC channel pointers
25     _SPM unsigned int *recvChanP;
26     _SPM unsigned int *sendChanP;
27     //processing type
28     _SPM pt_t *pt;
29     //parameters: S, Nr, Ns, Nf
30     _SPM unsigned int *s;
31     _SPM unsigned int *Nr;
32     _SPM unsigned int *Ns;
33     _SPM unsigned int *Nf;

```

¹<https://github.com/t-crest/patmos/tree/master/c/libaudio>

```

34     //in and out buffer size ( both for NoC or same core, in
        samples)
35     _SPM unsigned int *xb_size; //x buffer
36     _SPM unsigned int *yb_size; //y buffer
37     //audio data
38     volatile _SPM short *x; //input audio x[2]
39     volatile _SPM short *y; //output audio y[2]
40     //Audio effect implemented
41     _SPM fx_t *fx;
42     //Pointer to effect struct
43     _SPM unsigned int *fx_pnt;
44     //Boolean variable for last types: checks need to wait for
        output
45     _SPM int *last_init;
46     //Latency counter (from input to output)
47     _SPM unsigned int *last_count;
48     _SPM unsigned int *latency;
49 };
50

```

Listing 7.1: Parameters of the AudioFX structure.

Some of the parameters of Listing 7.1 are trivial. The ones that are not are explained here:

- The `send_am` and `recv_am` parameters store information about how many send or receive channels this effect is connected to (i.e. if it is a fork or a join effect).
- The `x_pnt` and `y_pnt` parameters point to the location where the audio data in the receive and send buffers is. This will be the buffer corresponding to a NoC channel, if the effect is connected to the NoC. Otherwise, it will be some location in the local SPM.
- The `pt` parameter defines the processing type of the effect: XeY , XgY or XlY (these terms were introduced in Subsection 6.3.2). This is needed by the `audio_process` function to know which steps it should execute during processing (sending, firing, receiving...) and how often.
- The `S`, `Nr`, `Ns` and `Nf` parameters were also introduced in Subsection 6.3.2.
- The `x` and `y` locations hold the audio samples, but are only used if the effect is not connected to the NoC on its input or its output, respectively. When it is connected, the audio samples are stored in the send and receive buffers of the NoC channels, handled by the functions of the message passing *libmp* library, and accessed by the `x_pnt` and `y_pnt` pointers.

- The `fx_pnt` parameter is a pointer to the actual audio effect structure (delay, filter, distortion, and so on). It can point to any of the effects presented in Section 5.4. That is why, at the beginning of this section, it has been stated that the `struct AudioFX` is implemented on top of the individual processing structures.
- Finally, the `last_init`, `last_count` and `latency` parameters are instantiated only when the effect is the last one of the chain, so it needs to take care of the audio signal latency, as explained. The `latency` parameter contains the latency value in iterations. The `last_count` is incremented in each run at the beginning, and when it reaches the latency value, the `last_init` boolean value is set to true, indicating that the output of audio data can begin.

7.1.2.2 The `alloc_audio_vars` Function

The `alloc_audio_vars` function can be found in the `audio.c` file of the `libaudio` library. It takes care of the audio effect allocation and initialization, and needs to be executed during setup time, before processing. It has no strict time requirements. The main argument it takes is `struct AudioFX *audioP`, which is a pointer to the effect object. The rest are values of the effect's parameters.

The function takes care of storing each parameter in the local SPM, using the `mp_alloc()` function of the `libmp` library to keep track of the next available address. As explained before, it does not store all the parameters of the `struct AudioFX`, but only the relevant ones for the given effects (for instance, if the effect is not the last of the chain, it does not make sense to store the parameters related to latency). It also initializes some parameters.

The audio effect that is processed (delay, distortion...) is also given as an argument. This function calls the `alloc_<FX>_vars` function to allocate the effect `<FX>` in the SPM. It can be any effect of the ones listed in Section 5.4. An important addition is an effect called `DRY_8SAMPLES`, which is unique in the sense that it processes a block of 8 samples, instead of just one, as the rest of the effects do. It does not make any actual processing, as it simply copies 8 input samples to its output buffer each time it fires. However, this effect has been created to show that the implemented multicore platform also supports effects that process more than a single sample, and that combinations of effects with different data-rates are synchronized correctly.

Finally, there is a function related to this one, named `free_audio_vars`, and it is called just before exiting the program, to free the space that has been dynam-

ically allocated in the external memory (such as audio buffers or modulation arrays of the effects).

7.1.2.3 The `audio_process` Function

The `audio_process` function is shown in the appendix Section C.1. It is the main function to process each effect, so it has strict real-time requirements. Its only input argument is a pointer to the effect structure, `struct AudioFX *audioP`. As stated before, this function is the same for all the effects in all cores, but the effect object passed as an argument will have its own parameters, so the function will act differently on each case. This function calls a different processing function depending on the effect type (distortion, delay...). The function called is `audio_<FX>`, where again, `<FX>` can be any of the effects listed in Section 5.4. The steps done by this function are briefly described here.

First of all, input and output audio data pointers `xP` and `yP` are created, which will point to the location specified by `x_pnt` and `y_pnt`. The location can be a NoC channel buffer or another SPM location. Then, the receiving, firing (processing) and sending steps are executed. For this, the function checks the processing type `pt` of the effect, and executes the steps in the correct order, as many times as needed, depending on the `Nr`, `Ns`, `Nf` and `S` values. These steps were defined in Subsection 6.3.2 for each processing type. The receiving and sending process is executed on every channel, if the effect is a join or a fork. If the effect is the first or last of the chain, the `audioIn` and `audioOut` functions are called respectively, to exchange data with the audio interface I/O device. In each step, the `xP` and `yP` pointers need to be incremented correctly.

If the effect is connected to the NoC, it calls the `mp_recv`, `mp_send` and `mp_ack` functions (the last one as many times as the receive). The timeout argument is used to prevent the platform from getting stuck when there is any problem. In the case of the `mp_send` function, the sending process through the NI and the Argo NoC will be overlapped with the computation: this means that the core can continue processing after sending, as long as it does not get stuck because there are no available send buffers. If the next effect in the chain is located in the same core, then data is simply placed in an SPM location, where it can be read by the next effect.

7.2 WCET Analysis and Static Effect Allocation

The static allocation algorithm used in this project is strictly related to the WCET analysis of each one of the implemented effects: in order to decide how to distribute the effects through the multicore platform, the allocator must have prior knowledge of the utilization of each effect, a concept introduced in Section 6.1. This way, it can decide which effects 'fit' together in a core.

Subsection 7.2.1 shows the WCET analysis and execution time measurements of each effect, and discusses how the values are interpreted. After that, Subsection 7.2.2 briefly explains some characteristics of the implemented task allocator. Finally, 7.2.3 overviews the main audio processing program, which processes the audio effects according to the effect distribution done by the allocator.

7.2.1 WCET Analysis and Execution Time Measurements

WCET analysis is essential for any real-time application. As explained in Chapter 3, the goal of the T-CREST platform is to develop a multicore processor for embedded real-time applications, so all the resources and tools of T-CREST focus on prediction and reduction of WCET. One of those tools is the platin WCET analysis tool, introduced in Section 3.3, which has been used in this project for WCET analysis of each effect. Apart from the software analysis, execution time has also been measured experimentally in the platform, simply reading the CPU cycles just before and after the function to be analysed. All the resulting values of the analysis and measurements must be compared to the sampling period, which is 1536 clock cycles for the 80 *MHz* Patmos processor with a sampling rate of 52.083 *kHz*. The execution time of each effect compared to this value gives an idea of whether real-time processing is possible or not. If this value is kept under 1536 cycles, the correct real-time functionality of the system can be guaranteed, avoiding audio interruptions.

Table 7.1 shows both WCET analysis done using the platin tool (two columns on the left) and experimental execution time measurements, as explained (two columns on the right). In both cases, measurements with cold and warm caches have been taken: on the first one, the caches are initially empty, so all the data and instructions need to be loaded from main memory; on the second one, the caches already hold the data and instructions of those functions, so execution time is smaller. On the experimental measurements with warm caches, a range of execution time values is found, so the bold font is used to mark the longest ones. The functions analysed here are each one of the individual effects processing functions. This means that the processing time related to NoC message passing

Effect	WCET <i>platin</i> cold cache	WCET <i>platin</i> warm cache	Experimental measurements cold cache	Experimental measurements warm cache
Filter (LP/HP)	21819	4728	7812	372
Filter (BP/BR)	21819	12600	7812	379
Tremolo	5981	2228	3444	201- 307
Vibrato	10393	5146	4872	373- 556
Delay	22745	12172	5208	458- 603
Wah-wah	24910	13921	8904	422- 588
Chorus	27942	15626	6804	464- 668
Overdrive	11806	6144	3444	178- 1657
Distortion	15221	8718	5040	1108

Table 7.1: WCET analysis and experimental execution time measurements of audio effect functions, using the *platin* tool for the analysis. The values indicate the amount of clock cycles needed to process one audio sample. The bold values show the worst-case values among all the measurements done with warm caches.

is not taken into account here (i.e. only the firing part of the `audio_process` function is considered).

The measurements of Table 7.1 need to be compared to the sampling period of 1536 clock cycles. Clearly, the WCET analysis done with *platin* reports that none of the effects can be processed in real-time (all the values are above 1536), neither with cold caches nor with warm ones. However, when measured experimentally, the values are much smaller, which means that the *platin* tool analyses WCET in a very pessimistic way (for instance, it guesses worst-case memory access times for every load/store instruction, which in reality is very non probabilistic). That is why this tool has been set aside on this work.

Before interpreting the results, it must be stated that the overdrive is a special effect, because the processing applied to it depends on the amplitude of the input signal, so the execution time can vary a lot. Execution takes longest for the $[1/3, 2/3]$ amplitude range, but this does not happen very often, due to the oscillating character of audio signals. That is why the measurements show a wide range of values. In the last column of Table 7.1, the values of the rest of the effects are within a range of around 200 clock cycles, while in the overdrive there is a difference of up to 1469 cycles.

The experimental measurements of Table 7.1 show big differences between cold and warm cache values: the speed-up is up to 21 (filter) with the warm ones, and all the values corresponding to cold caches are over 1536, meaning that execution-time per sample will be greater than the sampling period in this case.

The warm cache values, instead, are all within the acceptable range (except for the special overdrive effect, as explained). This means real-time processing is possible when the cache is warm, so in order to define the utilization values of the effects, experimental values corresponding to warm caches need to be considered. Therefore, it can be stated that the WCET corresponds to cold cache values, which should reside somewhere between the cold cache measurements and the results of WCET analysis. In Subsection 5.1.1 the cold cache WCET situation was faced, showing that misses are not perceived as interruptions, due to the flow-control communication paradigm used among the components of the system.

Table 7.2 shows the experimental execution time measurements taken for the whole `audio_process` function, that is, including the time taken for message passing on the NoC, buffer read/write position updating, and so on. The values here are all taken with warm caches, as real-time processing relies on it. The setup used for these measurements is a 2 core setup, where the master core just exchanges audio samples with the interface, and the slave core receives data from the master, processes and sends it back through the NoC. Execution of each effect running in this slave core is measured, reading the CPU cycles on every iteration of the processor. The values correspond to the initial executions, after caches get filled but before all samples in the input buffer of the audio interface are transferred to the output buffer (after they are transferred, the execution time of a full iteration synchronises with the sampling period). Different buffer sizes have been used, ranging from 1 to 16 samples. This shows how the execution time per sample gets reduced when increasing the buffer sizes, due to the reduction of NoC sending/receiving overhead.

If the values of Table 7.2 are analysed, this reduction in the overhead can be clearly appreciated. If any of the effects is taken, the tremolo for instance, the worst measured value relative to the sampling period is $610/1536 = 39.7\%$ for a buffer of 1 sample, but it is only $2962/24576 = 12\%$ for a buffer of 16 samples. This proves that execution time is reduced by increasing the buffer sizes, although this will also increase the latency of the signal. The percentage values calculated, 39.7% and 12%, are actually utilization values. Another term that was presented in Subsection 6.3.3 is the *overhead reducing factor (ORF)*. When an ORF of 16 is applied to the tremolo effect, the utilization decreases from 39.7% to 12%.

Using the results of the measurements, a table has been created, where utilization and ORF values have been assigned to each effect. This table is stored in JSON format, and can be found in the `audio_apps` folder of the aegean repository, called `FX_List.json`². The information on this file is used by the static allocator

²https://github.com/t-crest/aegean/tree/master/audio_apps/FX_List.json

Effect	Buffer size: 1 sample (1536 CCs)	Buffer size: 2 samples (3072 CCs)	Buffer size: 4 samples (6144 CCs)	Buffer size: 8 samples (12288 CCs)	Buffer size: 16 samples (24576 CCs)
Filter (LP/HP)	744	1080	1752	3096	5784
Filter (BP/BR)	751	1087	1759	3103	5791
Tremolo	527- 610	609- 694	841- 946	1366- 2290	2290- 2962
Vibrato	661- 829	913- 1264	1501- 1753	2593- 2776	5029- 5113
Delay	891- 1138	1311- 1642	2151- 2650	3831- 4430	7191- 8696
Wahwah	763- 931	1099- 1351	1771- 2107	3451- 3535	6475- 6643
Chorus	752- 1060	1172- 1340	1928- 2143	3571- 3907	6679- 7435
Overdrive	504- 573	569- 2027	790- 879	959- 1319	1479- 2928
Distortion	1433	2421	4397	8349	16253

Table 7.2: Execution time measurements of the general `audio_process` function, measured in clock cycles for different amounts of samples processed, using the experimental results with warm cache. Values taken before audio input buffer gets full (before processing period synchronizes with audio sampling period).

to decide which effects can be combined into a single core, and how big the buffer sizes shall be to reduce overhead.

The values in the `FX_List.json` table are based on the worst case measurements of Table 7.2, but they need to be oversized in most cases, in order to be conservative and ensure that audio is processed 'in time' in all cases. This is because the utilization values measured could change when effects are combined in a single core, for instance if the cache miss rate changes. Therefore, the utilizations given to the effects are either 0.5 for the 'lightest' effects, or 1 for the 'heaviest' ones. This means that the lightest effects with an utilization of 0.5 can be combined in the same core, while the ones with an utilization of 1 need to be processed individually. The ORFs range from 4 to 16. These values have been assigned taking into account results of Table 7.2 and have been oversized following further experimental tests. They have proved to effectively reduce overhead, but keep the signal latency in an acceptable range. Finally, it has been also proved experimentally that it is possible to process the overdrive effect in real-time when an ORF value of 16 is assigned to it, even with the wide range of execution times that it has.

Due to the difficulty of setting a WCET value for each effect, the input and output buffer sizes have been oversized. A value of 128 samples seems to be a good compromise between keeping latency low and avoiding interruptions on WCET situations such as cache misses. The relation between WCET and the

audio I/O buffer size was shown in Subsection 5.1.1.

7.2.2 Static Effect Allocator

The static allocator implemented in this project is simple but has proved to be effective. The reason for this is that the main focus on this project is not on task allocation, but on how the effects in the real-time audio processing system synchronize and exchange data with each other effectively. The static task allocation method used here was already presented in Section 6.1, explaining how it minimizes the amount of communication channels required.

The designed allocator is the *audioFXGen.py*³ Python file, found in the Aegean repository of T-CREST. It uses two JSON files: one of them is the *FX_List.json* file explained before, containing the utilization and ORF values of each effect, and the other one is a description of the audio application to be executed in the platform. Many of these files describing different audio effect setups have been stored in the *audio_apps*⁴ folder of the aegean repository, to be used as examples. One of them is shown in Listing 7.2. It contains two modes, which means there are two effect setups, allowing the player to switch between them. Each one of the modes contains the names of the effects (the naming order defines their position in the chain). Instead of an effect name, the list can contain an object called “chains”: in this case, the signal is split in parallel chains, and then merged together. In this example, parallel chains with delay and distortion effects would be created in the first mode. Signal flow representations of the two modes of this example are also shown in Figure 7.1.

```

1 { "modes" : [
2   [
3     "WAHWAH", { "chains" : [
4       [ "DELAY" ],
5       [ "DISTORTION" ] ]}
6   ],
7   [
8     "OVERDRIVE", "VIBRATO", "HP"
9   ]
10 ]}

```

Listing 7.2: Example of a possible audio effect setup in JSON format. There are two modes, and the first one has two parallel audio chains.

The steps performed by the allocator for a given audio application are the following:

³<https://github.com/t-crest/aegean/tree/master/python/audioFXGen.py>

⁴https://github.com/t-crest/aegean/tree/master/audio_apps/

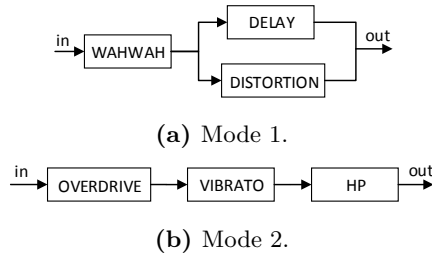


Figure 7.1: Example of an audio application with two modes, corresponding to the description in Listing 7.2.

- First of all, it assigns the effects of the application to the cores in the multicore platform for each mode. It might report errors, for instance when not all the effects could be placed due to space reasons, or because the latencies of parallel chains cannot be balanced correctly. Initially, all the cores have 100% free utilization (i.e. they are totally empty), except for the master, which has an utilization of 50%, because it has to exchange data with the audio interface.
- Then, the allocator connects the effects according to the application requirements and to the placement of the effects in the platform. It assigns IDs to each channel: some of them are NoC channels, some others are virtual channels inside the same core.
- It calculates the buffer sizes for each effect. It does it as an iterative process, following the rules explained in Subsection 6.3.2, and uses the ORF values of each effect given in *FX_List.json* to increase the buffer sizes. The sizes are also increased when two or more effects are placed on the same core in order to reduce overhead even more, but sizes are currently limited to 32 samples as a measure to keep the latency low.
- It does an initial calculation of the latency. The final calculation is done by another script, called *audioLatencyCalc.py*⁵, which uses the worst-case packet latency value according to the TDM period of the NoC schedule.
- It creates a C header file, called *audioinit.h*, and stores it in the *libaudio* library of the Patmos repository. An example of this file is shown in the appendix Section C.2, which corresponds to the setup in Figure 7.1.
- It extracts the NoC channels and creates a NoC schedule XML file with the custom setup for each mode described in the application. An example of this file is shown in the appendix Section C.3, also from the setup of Figure 7.1.

⁵<https://github.com/t-crest/aegean/tree/master/python/audioLatencyCalc.py>

Finally, as explained, the *audioLatencyCalc.py* script reports the latency of the system, so that the user can check that it is within an acceptable range, under 15 ms. This script also generates the *latencyinit.h* file, located in the *libaudio* library of Patmos as well, which contains the latency of each mode, measured in iterations of the master core, as explained.

7.2.3 Main Audio Program

The main audio program implemented in this project is called `audio_main.c`⁶. It can be found in the *audio_apps* directory of the Patmos repository, and has been implemented in a flexible way so that it can be used to run different audio applications described in the *audio_apps* folder of Aegean. This C program reads the schedule and latency generated by the allocator as *audioinit.h* and *latencyinit.h* files. Here are the main steps it executes:

- First of all, during the setup stage, the thread functions are created in the slave cores, as many as needed for the application.
- Then, on each core, the `struct AudioFX` objects are instantiated and initialized for all the modes in the audio application. This is done by reading the information about the effect distribution and the parameters in the *audioinit.h* file, setting the buffer sizes of the effects and calling the `alloc_audio_vars` function as explained in Subsection 7.1.2.2. NoC channels are also created in this stage. When finished, the cores must wait until all others have finished the setup stage before processing can start. When this happens, the NoC channels are initialized. If there are no errors during setup and initialization, processing can start.
- The cores execute the `audio_process` function on all the effects assigned to them in the current mode. This is repeated until the user presses a button in the FPGA board, to finish the program.
- Additionally, the user might press another button of the FPGA to trigger a mode change. In that case, the master core waits until all threads finish processing their current functions, and enables a mode change using shared memory communication. After this happens, all the cores in the system process the effects according to the new mode. In a mode change, NoC reconfiguration will also happen, if enabled, calling the `noc_sched_set()` function of the *libmp* library to update the channels and bandwidth values of the Argo NoC according to the new mode.

⁶https://github.com/t-crest/patmos/tree/master/c/audio_apps/audio_main.c

Evaluation and Discussion

This chapter presents the evaluation of the audio processing multicore platform, showing different types of verifications. It also provides discussion on some of the main topics related to this project.

The evaluation is found in Section 8.1, where 4 different types of verifications are presented. The discussion is in Section 8.2, where some aspects and design decisions of the implemented platform are reviewed.

8.1 Evaluation

Part of the evaluation of the implemented audio processing platform has already been presented in Subsection 7.2.1, when it was shown how the processors are able to process the effects in real-time with warm caches. However, further verification of the system is done here in 4 parts. First of all, Subsection 8.1.1 verifies that the effect allocation process is done correctly, according to the design rules presented in Chapter 6 and utilization values derived from execution time measurements shown in Chapter 7. For this, an example application is used, and the generated C header files and NoC schedule file explained, which are shown in appendix C. Then, Subsection 8.1.2 evaluates the correctness of the audio DSP algorithms used, focusing on the low-pass filter effect. After

that, Subsection 8.1.3 verifies the synchronization and communication of the effects in the system, as it shows that the stream of audio data flows through the effects as expected, keeping the right order. Additionally, the verification of the latency estimation is done, which ensures the real-time perception of audio processing. Finally, in Subsection 8.1.4 audio applications with parallel effect chains are tested, showing a numerical example where the samples of the concurrent chains are added in the join effect, meaning that the parallel signals are merged together.

8.1.1 Evaluation of the Task Allocation Algorithm

As stated before, the implemented static task allocation algorithm is light but effective, and its behavior has proved to be correct for all the tests done. To show this, an example application is used, which corresponds to the two modes of effect setups shown in Figure 7.1 and Listing 7.2. The *audioinit.h* header file created by the static allocator (*audioFXGen.py*, Subsection 7.2.2) for this application is shown in the appendix Section C.2, as it is too long to be displayed here. Most of the parameters shown in this file are trivial, but there are two which are not:

- **FX_SCHED_<X>** is an array containing information about the effects setup on mode <X> (one array is created for each mode). Each row in the array corresponds to an effect, and the values found from left to right are the effect ID, the core it must be placed in, the type of effect that it implements (delay, tremolo...), the input and output buffer sizes, the amount of samples processed per firing (S) and its input and output connection types (NoC, same core...). The effects in this array match the setups presented in Figure 7.1, and their buffer sizes fulfill the synchronization rules as expected.
- The **SEND_ARRAY_<X>** and **RECV_ARRAY_<X>** arrays contain information about how the effects are connected to each channel in the platform (NoC or same core channels). The rows are the effects in the system, and when a value of 1 is found in the array, it means that the effect on that row is connected to the channel ID assigned by the column. More than one 1's might appear in a row in the case of fork and join effects. This is what happens in the first mode of the example shown in the appendix Section C.2. It can be checked that the connections between the effects match the signal flow of Figure 7.1 on each mode, showing that the allocator keeps the right order of effects in the chain.

As explained, the other file generated by the allocator is the NoC schedule file, shown in the appendix Section C.3. The correctness of this one is easy to test, as two communication fields are found, each one corresponding to a mode, and the NoC channels created match the communication requirements shown in the *audioinit.c* file of appendix Section C.2. In this example, the audio data NoC channels have been given a bandwidth of 4 packets (this value has been found to perform fairly well in most cases), while the channels in the inverse direction have a bandwidth of 1, used only for acknowledge signals.

Our static task allocator has been tested in a similar way for many other effect setups, even with more modes and more complex configurations, and the algorithm was done correctly in all cases, even reporting errors when creating faulty applications on purpose (with too many effects, or unbalanced parallel chains).

8.1.2 Evaluation of the Audio Processing Algorithms

The numerical evaluation of the DSP algorithms used for processing effects has been done throughout all the development stages of the project. The UART has been used as a debugging tool to print input, intermediate and output audio sample values and ensure that all the calculations done corresponding to the algorithms used are correct. As an example, part of the terminal console is shown in Listing 8.1, which corresponds to printing the values of all the coefficients and all the samples of an IIR filter. In this case, it is a low-pass filter.

```
1 A [0]=14789 , Y [0]=-531 , B [0]=20 , X [0]=-1004
2 A [1]=-31091 , Y [1]=-565 , B [1]=40 , X [1]=-1004
3 A [2]=0 , Y [2]=-496 , B [2]=20 , X [2]=-1003
4 RESULT=-2448438
5 A [0]=14789 , Y [0]=-565 , B [0]=20 , X [0]=-1004
6 A [1]=-31091 , Y [1]=-598 , B [1]=40 , X [1]=-1003
7 A [2]=0 , Y [2]=-531 , B [2]=20 , X [2]=-1003
8 RESULT=-2579222
9 A [0]=14789 , Y [0]=-598 , B [0]=20 , X [0]=-1003
10 A [1]=-31091 , Y [1]=-630 , B [1]=40 , X [1]=-1003
11 A [2]=0 , Y [2]=-565 , B [2]=20 , X [2]=-1003
12 RESULT=-2705936
```

Listing 8.1: Coefficients, buffered samples and results for 3 executions of a 2nd order IIR filter, corresponding to a low-pass filter effect.

The A and B arrays shown in Listing 8.1 have constant values, which correspond to the 3 a_n and b_n coefficients found in a 2nd order IIR filter, as explained in Subsection 2.2.2. The values of the X and Y input and output audio sample buffers are also shown, and it can be seen how some values get shifted from

one iteration to the next. The **RESULT** value is the output sample, calculated following the IIR filter Equation 2.4. The calculation of that equation can be done manually for one of the executions shown in Listing 8.1. For instance, if the middle execution is taken:

$$y(n) = (20 \cdot (-1004)) + (40 \cdot (-1003)) + (20 \cdot (-1003)) - (14789 \cdot (-565)) - ((-31091) \cdot (-598)) - 0 = -10316893$$

This value needs to be divided by 4, because the samples in the 2nd order IIR filter implementation are shifted 2 positions to the right, to avoid overflow situations. When this is done, the resulting value is -2579223.25 , which is close to the -2579222 reported by the platform. The small differences are due to the loss of precision of fixed-point arithmetic calculations because of the constant scaling and round-off operations that need to be done. This verifies that the algorithm is computed correctly. Similar numerical tests have been done with all the algorithms implemented and all of them have proved to work as expected, with some differences in the resolution loss.

In audio effects processing, a perceptual evaluation is also very important: it is a way to ensure that the implemented algorithm actually generates the audible results that are expected. This type of verification has also been used constantly during this project, for tuning the parameters of the effects, such as filter coefficients, delay values, and so on. Sometimes, a time domain or frequency domain visualization of the effect is helpful to verify that the behavior is the expected one. The visual verification of the low-pass filter is shown in Figure 8.1, and has been created by inputting a white noise signal with constant gain all frequencies to the system and plotting the processed output with an FFT visualizer. It can be clearly appreciated how the amplitude of the upper frequencies of the spectrum is reduced.

8.1.3 Evaluation of the Synchronization of the System

In this subsection, the functionality of the multicore audio processing platform is evaluated in terms of synchronization and data flow. On the first part of the evaluation, the focus is on ensuring that all the steps are executed correctly when processing (not the audio DSP algorithms themselves, but rather the NoC sending and receiving operations, the read/write position updates and so on), and that the audio signal travels through the cores uninterruptedly and in the correct order, following the path specified in the JSON application description file. The next verification consists of measuring the actual signal latency in the platform, to confirm that it matches the expected value.



Figure 8.1: Spectrum of the implemented low-pass filter effect for a white noise input signal with constant gain all over the spectrum, showing the decay of frequencies above the cut-off frequency.

To do this, a simple effect setup has been created, consisting only of dry audio effects. These effects do not apply any processing to the signal, but just copy their data from input to output. This means that the audio signal out of the system should be exactly the same as the input signal. The application used for the test is shown in Listing 8.2, and it can be appreciated how effects of different firing rates have been mixed in the platform (the `DRY` effect fires on every sample, while the `DRY_8SAMPLES` fires every 8 samples). The next Listing, 8.3, shows part of the header files generated by the allocator. As the 4th and 5th columns of the `FX_SCHED_0` array show, the input and output buffer sizes of all effects in the system is 16, and some of them are combined in the same core (for instance, effects 1 and 2 are mapped both to core 1, as the 2nd column shows). The latency of the master core is also shown, which is 5 iterations.

```
1 { "modes" : [  
2   [  
3     "DRY", "DRY", "DRY", "DRY_8SAMPLES", "DRY", "DRY"  
4   ]  
5 ]}
```

Listing 8.2: Audio application of chained dry effects.

```

1 // FX_ID | CORE | FX_TYPE | XB_SIZE | YB_SIZE | S | IN_TYPE |
  OUT_TYPE //
2 const int FX_SCHED_0[7][8] = {
3     { 0, 0, 0, 16, 16, 1, 0, 2 },
4     { 1, 1, 0, 16, 16, 1, 2, 3 },
5     { 2, 1, 0, 16, 16, 1, 3, 2 },
6     { 3, 2, 1, 16, 16, 8, 2, 3 },
7     { 4, 2, 0, 16, 16, 1, 3, 2 },
8     { 5, 3, 0, 16, 16, 1, 2, 2 },
9     { 6, 0, 0, 16, 16, 1, 2, 1 },
10 };
11
12 //latency from input to output, measured in iterations
13 const unsigned int LATENCY[MODES] = {5, };

```

Listing 8.3: Part of the header files generated by the allocator, corresponding to the application shown in Listing 8.2.

When generating this header file, the allocator also reports the latency of the system in the terminal window. The values reported for the example application of Listing 8.2 are shown in Listing 8.4. First of all, the overall latency is 3.99 ms, which is under 15 ms, considered acceptable. For other more complex applications tested, it has always been under 10 ms. The latency that the master core sees is the addition of the FX and NoC latencies, 65 samples. As explained in Subsection 7.1.1, the master measures the latency in runs or iterations, which means that the amount of samples needs to be rounded up to $5 \cdot 16 = 80$ (this was shown in Equation 7.1), corresponding to the 5 iterations of the master core. This means that 80 samples will be input to the master core from the audio interface before there is any audio output. This is exactly the value that the allocator reports, which, added to the 128 samples of the I/O buffer latency, give the overall one, 208 samples.

```

1 ***** MODE 0 *****
2 IO Latency: 2.46 ms (128 samples), FX Latency: 1.23 ms (64 samples)
  , NoC Latency: 0.02 ms (1 samples)
3 Total Latency of FX and NoC:1.54 ms (80 samples)
4 TOTAL LATENCY: 3.99 ms (208 samples)

```

Listing 8.4: Latency report of the allocator, corresponding to the application shown in Listing 8.2.

To prove the functionality of the platform for the example application shown in Listing 8.2, The UART has been used in the master core, to print the input and output data from and to the audio interface. Furthermore, the iteration number is also printed. Two separate parts of this are shown in Listing 8.5, corresponding to the iterations 11 and 15 of the master core.

```
1 ...
2 audio OUT: -4357, -4150
3 audio OUT: 2093, 2134
4 ***** ITERATION 11 *****
5 audio IN: 453, 535
6 audio IN: -942, -825
7 audio IN: -2484, -2327
8 audio IN: -3356, -3176
9 audio IN: -3078, -2905
10 audio IN: -1944, -1798
11 audio IN: -417, -309
12 audio IN: 669, 748
13 audio IN: 2421, 2454
14 audio IN: 1884, 1929
15 audio IN: -13211, -12768
16 audio IN: -1642, -1474
17 audio IN: 14430, 14163
18 audio IN: 10890, 10692
19 audio IN: 7646, 7518
20 audio IN: -1703, -1606
21 audio OUT: -2441, -2283
22 audio OUT: 601, 680
23 ...
24 ...
25 ...
26 ...
27 audio OUT: 4382, 4366
28 audio OUT: 3362, 3368
29 ***** ITERATION 15 *****
30 audio IN: 316, 402
31 audio IN: -523, -415
32 audio IN: -2220, -2066
33 ...
34 audio IN: -9205, -8855
35 audio IN: 3739, 3760
36 audio IN: 11522, 11331
37 audio OUT: 453, 535
38 audio OUT: -942, -825
39 audio OUT: -2484, -2327
40 audio OUT: -3356, -3176
41 audio OUT: -3078, -2905
42 audio OUT: -1944, -1798
43 audio OUT: -417, -309
44 audio OUT: 669, 748
45 audio OUT: 2421, 2454
46 audio OUT: 1884, 1929
47 audio OUT: -13211, -12768
48 audio OUT: -1642, -1474
49 audio OUT: 14430, 14163
50 audio OUT: 10890, 10692
51 audio OUT: 7646, 7518
52 audio OUT: -1703, -1606
53 ***** ITERATION 16 *****
54 audio IN: -1088, -973
55 audio IN: -5521, -5290
```

Listing 8.5: Input and output audio data values printed by the master core in the platform, corresponding to the setup of Listing 8.2.

What Listing 8.5 shows is that the 16 audio samples that are input to the platform in iteration 11 are output in iteration 15. This verifies two things: one of them is that data is correctly processed in all effects (the values are unaffected, as they are all dry effects), as the order of the samples in the stream is respected and no data is lost, which means that communication and synchronization happen as expected; the second one is the correct estimation of the latency, which can be checked taking one of the input samples, for instance the (453, 535) sample (two values because it is a stereo sample) that is the first input on iteration 11. The same sample is output in the first position in iteration 15, proving that 80 samples have been input in between: the 64 ones of iterations 11 to 14, and the 16 ones of iteration 15 (because audio input happens before output on each iteration). This value, 80 samples, is the value that was previously calculated.

Similar verifications have been done with multiple possible setups and effects of different processing types (XeY , XgY and XlY), and all of them have shown to work as expected, with no data loss or unwanted reordering of the samples. In all cases, the multicore platform processes the audio signal with the expected behavior. It has also been constantly verified that the master core takes care of the latency of the system correctly, even with parallel effect chains, as they have the same latencies.

Additionally, mode changes without NoC reconfiguration have also been tested with a wide range of applications and they show correct behavior. On a mode change, the platform switches to the new setup correctly, and all the cores start processing the effects corresponding to the new mode. The usage of NoC reconfiguration in mode changes has only been tested for simple application setups, and it has been verified that it functions as expected in those cases.

8.1.4 Evaluation of Parallel Audio Effect Chains

The last evaluation presented verifies that the implemented system supports parallel audio effect chains. To show this, a simple audio application has been created, with 3 modes, as presented in Listing 8.6. The first and second modes consist of a single effect, the distortion and the overdrive respectively. On the last mode, these two effects are placed in parallel chains. The results of this verification are shown in Table 8.1. For the 3 modes, the same input samples have been used, which are shown in the first column of the table. The next columns show the output values of each mode.


```

1 { "modes" : [
2   [
3     "DISTORTION"
4   ],
5   [
6     "OVERDRIVE"
7   ],
8   [
9     {"chains" : [
10      [ "DISTORTION" ],
11      [ "OVERDRIVE" ]
12    ]}
13 ]
14 ]}

```

Listing 8.6: Application to verify the functionality of parallel audio chains.

Input	Distortion	Overdrive	Parallel Chains
4376, 4308	24428, 24313	8752, 8616	16590, 16464
-15224, -15288	-30894, -30908	-28754, -28832	-29824, -29870
24327, 24261	32179, 32173	32767, 32767	32472, 32469

Table 8.1: Results of the verification of parallel audio chains, corresponding to the modes presented in Listing 8.6 for 3 different input samples.

The values of the 2nd and 3rd columns of Table 8.1 are not of interest here, as they are just the result of the DSP algorithms corresponding to the distortion and overdrive effects for the given samples. The focus here is in comparing those values with the last column. Before that, it must be explained that the join effect found after parallel chains reduces the amplitude of each one of them, otherwise the addition of two signals could result in an overflow. For two parallel chains, the amplitude of each signal is reduced to a half. Therefore, the correct functionality of the 3rd mode of Listing 8.6 is verified in the last column of Table 8.1: if the values of the 2nd and 3rd column are divided by 2 and added, the resulting values match the ones in the last column. This shows that the audio signals of parallel chains are actually added correctly, and that the relative latencies of each chain does not get modified in an undesired way. Further evaluation has been carried out with more numerical examples and longer signals, and perceptual evaluation has also given correct results.

8.2 Discussion

In this section, some aspects of the presented design and implementation are discussed. In many chapters of this work some discussion has already been

provided, as it was needed to justify some design decisions. Among others, discussion has been provided on topics such as communication paradigms, WCET analysis and reduction, methods for synchronization and communication of effects, and compromise between parameters such as buffer sizes and latency. Some of these topics are briefly recalled here, and some new points of discussion are introduced.

This section is divided in two subsections. The first one, 8.2.1, focuses on some general aspects of the audio processing platform, and discusses strengths and possible improvements of the system. The second Subsection, 8.2.2, briefly recalls the allocation algorithm used in this project, and discusses some state of the art allocation and scheduling solutions.

8.2.1 General Discussion

Real-time audio applications are challenging due to the constrained time requirements for processing. The complexity of the algorithms to be computed is limited by the resources of the chosen platform. In multicore platforms, these resources are mainly distributed among processing units (IP cores) and intercommunication resources, which exploit parallelism as computation and communication are overlapped.

The T-CREST platform chosen for implementation is optimized for general-purpose hard real-time applications. The implementation of audio DSP algorithms in such a platform is possible, as it has been demonstrated, but the complexity of them is limited by the computational resources available. During the development of this project, some decisions have been taken to find a balance between the complexity of the implemented effects and the time required for execution. Due to this, some of the algorithms slightly decrease the resolution of the signal, which might introduce some unwanted noise artifacts. We do not consider that this affects the quality of the work presented here, as the goal of the project is not to optimize the quality of the effects, but rather to design a system where chains of effects are processed and synchronized efficiently and with strict time guarantees.

In order to process complex audio algorithms with high resolution and minimal error in real-time, multi-processor platforms are usually equipped with a set of different IP cores that are specialized for each operation. The work presented in [8] discusses the use of General-Purpose Graphics Processing Units (GPGPU) as part of a multicore system to take care of the computationally most expensive algorithms. As it is stated in [32], it is also common to find multi-processors with DSPs or FPGAs interconnected with NoCs.

In the current implementation, all the IP cores found in the system are Patmos processors. However, the platform has been designed with high scalability, which means that in theory not only Patmos processors are supported: other IP cores shall also be integrated in the system if these can interface the Argo NoC, and if the rules explained in Subsection 6.3.2 are accomplished. For instance, a more powerful platform can be designed if hardware blocks optimized for audio processing algorithms are included in the platform, such as high order IIR filter blocks or FFT blocks. This kind of hardware implementation would not only increase the computational power available in the platform, but would also reduce WCET and improve its predictability, as the execution time would not depend on the compiler or instruction cache hits anymore.

The main strengths of the T-CREST platform for audio processing are the local SPM and the Argo NoC. The first one allows storing most of the effect parameters in a fast access memory, to prevent the processor from stalling on data cache misses. Moreover, it improves WCET predictability, as SPM access time is constant.

As far as the Argo NoC is concerned, its TDM behavior is excellent for real-time audio applications. Custom NoC schedules optimize the usage of the available bandwidth to create only those channels that are required for a given application, depending on the effect distribution on the platform, which in this case is constant due to static task allocation. Furthermore, in the current implementation the data-rate of all the effects is the same and constant (i.e. each effect processes one sample per sampling period) and so it is for the NoC channels, so the available bandwidth is optimized. In general, all multicore audio applications require large amount of data to be transferred between the IP cores of the system, so a TDM scheduled NoC seems like a very good solution for real-time applications when off-line allocation or scheduling are used.

8.2.2 Task Allocation and Scheduling

The performance of an audio application relies on how the resources provided in a multi-processor are employed. In this sense, correct distribution of tasks among cores becomes essential to maximize the usage of available computational resources. Moreover, the communication requirements of the application must also be considered, as large amounts of overhead might be introduced to the system if the allocation is not done efficiently.

Our static allocation algorithm, presented in Subsection 7.2.2, is simple but has proved to perform an effective task distribution depending on their utilization values, as the amount of NoC channels needed for an application gets mini-

mized, therefore reducing message passing overhead and signal latency due to data transfers through the NoC. This is achieved by mapping the effects to the cores according to their order in the audio signal chain. However, the computational resources are not optimized in this way. For a given audio application, it might happen that our allocation method is not able to place all the effects in the 4-core platform. On the other hand, an scheduler which optimizes the processing resources could find a way to distribute all of them efficiently, although this would increase the communication requirements. In general, the algorithm used must find a balance between optimizing the usage of computational and intercommunication resources. But, as explained before, task allocation is not the main focus of this work, so the performance of the algorithm used is considered to be enough for this work.

The dependency between WCET analysis and task allocation has been experienced in this project. If utilization rates of the effects are not estimated correctly, real-time processing might fail as one of the cores in the system is not able to process in time, thus limiting the execution of the whole system. To avoid this, the utilization values and overhead reducing factors have been overestimated, due to the unpredictability of combining effects on the same core, as the compiler might change the order of the instructions, or the cache hit rates might decrease. With more precise WCET values for each effect in every possible combination, allocation could be done more precisely, without so much overestimation required. In this case, longer chains of effects could fit in the platform.

Task allocation and scheduling for audio applications is an advanced topic which is constantly under development, as there is currently much research going on in this field. The work presented in [9] approaches task scheduling as a graph theory problem, where components or nodes are connected between each other through edges. It then proposes solutions for the problem of task scheduling in worker threads. In [32], a dynamic-scheduling solution is proposed, based on events. Here, the scheduler is part of the platform and dynamically assigns tasks to the available resources. These ones generate events when they are ready to receive new tasks. An advantage of dynamic scheduling is that the available resources can be optimized: the scheduler can minimize the amount of cores needed to process tasks, freeing computation on other cores. When WCET situations happen on some tasks, the scheduler will increase the amount of cores used for computation.

As a future improvement, many of these algorithms could be integrated in the implemented platform to maximize the usage of resources and schedule the computation of audio effects efficiently.

Conclusion

This chapter concludes the thesis. First of all, Section 9.1 briefly lists the contributions made in this work and the results. After that, Section 9.2 proposes the future work.

9.1 Contributions and Results

In this thesis we have contributed to real-time multicore audio processing, proposing a solution which allows effects to communicate and synchronize effectively, and using a TDM scheduled NoC to provide communication guarantees within a time interval for all processors in the system. In addition, this work also contributes to the T-CREST project, as audio processing has been used as a test application for the multicore platform and the Argo NoC.

The following is a list of the steps that have been followed during the development of this project, and the results obtained:

- We have improved the design of the audio interface for Patmos, integrating circular input and output buffers which provide the processor with more flexibility for real-time processing.

- We have acquired knowledge on the main DSP algorithms that can be applied for audio signal processing, and on how the different parameters affect the sound.
- We have developed the integration of such DSP algorithms into the world of real-time audio processing, implementing them in an efficient way, balancing the computation requirements with the complexity of the algorithm, and finally optimizing WCET by making use of local memories.
- We have implemented a set of audio effects in C that run on a Patmos processor, using the designed audio interface for audio input/output.
- We have designed a set of rules that allow using a multicore platform to process different audio effects that are connected between each other forming chains, taking care of balancing overhead associated to data transfers with the latency of the signal, to ensure that real-time perception is accomplished in all cases.
- We have implemented the multicore processing system on T-CREST platform, which allows processing sequential and parallel chains of effects in real-time.
- We have implemented audio mode changes, which allows having more than one effect setup in a single application to switch among them at run-time.
- We have implemented a software tool which performs the allocation of audio effect tasks, following the mentioned rules to distribute the effects in the multicore platform, minimizing the usage of communication channels.
- Finally, we have verified the correct functionality of different aspects of the implementation, such as the communication and processing on the platform and the performance of the allocation algorithm. We have also discussed the high scalability of the design, which allows integration of other IP cores in the system.

9.2 Future work

The following improvements and extensions of the project are proposed as future work, which would increase the performance of the system in many senses:

- Up to 12 different audio effects have been implemented in the T-CREST platform. All of them are either filters (possibly with temporal variations) or non-linear effects. As a future extension, the implementation of more

and more complex effects is proposed, such as spatial or frequency-domain effects.

- In relation to the previous point, the integration of hardware blocks that implement audio processing algorithms is proposed, which would allow more complex implementations of some effects, and would also reduce WCET and improve its predictability, as explained.
- The usage of the instruction SPM is also recommended, as this would reduce WCET considerably. In the current implementation, data cache misses are minimized using a local data SPM, but instruction misses seem to be an important limitation for real-time processing.
- Finally, a more complex static task allocation algorithm would maximize the computational and communication resources of the platform, finding a correct balance between these two.

Bibliography

- [1] Eric Battenberg, Adrian Freed, and David Wessel. Advances in the Parallelization of Music and Audio Applications. *Proceedings of the International Computer Music Conference*, pages 349–352, 2010.
- [2] Richard Boulanger and Victor Lazzarini. *The Audio Programming Book*. The MIT Press, 2011.
- [3] Udo Zöler. *DAFX : Digital Audio Effects*. 2 edition, 2011.
- [4] Jose A. Belloch, Balázs Bank, Lauri Savioja, Alberto Gonzalez, and Vesa Välimäki. Multi-Channel IIR Filtering Of Audio Signals Using a GPU. *Proceedings of the IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP)*, 2014.
- [5] Music-DSP Source Code Archive. Available online at <http://musicdsp.org/archive.php?classid=4#41> (last accessed: Jan. 2017).
- [6] James A. Moorer. About This Reverberation Business. *Computer Music Journal*, 3(2), 1979.
- [7] Hudson Giesbrecht, Will Mcfarland, and Tim Perry. Algorithmic Reverberation Combining Moorer’s reverberator with simulated room IR reflection modeling. Technical report, 2009. Available online at <http://arqen.com/wp-content/docs/Hybrid-Convolution-Algorithmic-Reverb.pdf> (last accessed: Jan. 2017).
- [8] Tiziano Leidi, Thierry Heeb, Marco Colla, and Jean-philippe Thiran. Event-driven real-time audio processing with GPGPUs. In *Audio Engineering Society Convention 130*, pages 1–10, May 2011.

- [9] Andreas Partzsch and Ulrich Reiter. Multi core / multi thread processing in object based real time audio rendering: Approaches and solutions for an optimization problem. In *Audio Engineering Society Convention 122*, May 2007.
- [10] Steven W. Smith. *The Scientist and Engineer's Guide to Digital signal processing*. California Technical Publishing, 2 edition, 1999.
- [11] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos Reference Handbook. Technical report, 2017. Available online at http://patmos.compute.dtu.dk/patmos_handbook.pdf (last accessed: Jan. 2017).
- [12] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter Puschner. *The platin Tool Kit - The T-CREST Approach for Compiler and WCET Integration*. Available online at https://publik.tuwien.ac.at/files/PubDat_246928.pdf (last accessed: Jan. 2017).
- [13] Rasmus Bo Sørensen, Luca Pezzarossa, Martin Schoeberl, and Jens Sparsø. A Resource-Efficient Network Interface Supporting Low Latency Reconfiguration of Virtual Circuits in Time-Division Multiplexing Networks-on-Chip. *Submitted to Journal of Systems Architecture: Embedded Software Design*, 12 2016.
- [14] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christoph Müller, Kees Goossens, and Jens Sparsø. Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, 2016.
- [15] Martin Schoeberl, Sahar Abbaspourseyedi, Alexander Jordan, Evangelia Kasapaki, Wolfgang Puffitsch, Jens Sparsø, Benny Akesson, Neil Audsley, Jamie Garside, Raffaele Capasso, Alessandro Tocchi, Kees Goossens, Sven Goossens, Yonghui Li, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, Peter Puschner, André Rocha, and Cláudio Silva. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.
- [16] Luca Pezzarossa. *Hardware Accelerators in Network-on-Chip Based Multi-Core Platforms*. Technical University of Denmark, 2014.
- [17] Terasic Technologies. DE2-115 User Manual. 2010.
- [18] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for patmos. *International Symposium on Object-oriented Real-time Distributed Computing*, pages 100–108, 2014.

- [19] OCP International Partnership. Open Core Protocol Specification 3.0. Technical report, 2009.
- [20] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The t-crest approach of compiler and wcet-analysis integration. *Proceedings - Object-oriented Real-time Distributed Computing, International Symposium on*, page 6913220, 2013.
- [21] Chris Lattner. Introduction to the LLVM Compiler Infrastructure. *Computer-Aided Civil and Infrastructure Engineering*, 21(5):319–320, 2006.
- [22] Daniel Sanz Ausin and Fabian Goerge. Design of an Audio Interface for Patmos. 2016. Available online at <https://arxiv.org/abs/1701.06382> (last accessed: Jan. 2017).
- [23] Wolfson Microelectronics. WM8731 Audio Codec. Technical report, 2004. Available online at <http://www.cs.columbia.edu/~sedwards/classes/2012/4840/Wolfson-WM8731-audio-CODEC.pdf> (last accessed: Jan. 2017).
- [24] Roman Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms*. Springer Science + Business Media, 2005.
- [25] Whirlwind. Opening Pandora’s Box? Available online at <http://whirlwindusa.com/support/tech-articles/opening-pandoras-box/> (last accessed: Jan. 2017).
- [26] The CSound Community. Real-time audio using csound. Available online at <http://www.csounds.com/manual/html/UsingRealTime.html> (last accessed: Jan. 2017).
- [27] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications*. Prentice-Hall International, Inc., 3 edition, 1996.
- [28] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):840–848, 2000.
- [29] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 1998.
- [30] Nenad Cetic, Miroslav Popovic, Miodrag Djukic, and Momcilo Krunic. A Run-time Library for Parallel Processing on a Multi-core DSP. *Proceedings of 2013 IEEE 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2013*, pages 41–47, 2013.

- [31] Fatma A. Omara and Mona M. Arafa. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13–22, 2010.
- [32] Tiziano Leidi, Thierry Heeb, Marco Colla, and Jean-Philippe Thiran. Model-driven development of audio processing applications for multi-core processors. In *Audio Engineering Society Convention 128*, May 2010.
- [33] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the Ieee*, 75(9):1235–1245, 1987.
- [34] Claudius Ptolemaeus. *System Design, Modeling, and Simulation. Using Ptolemy II*. 1 edition, 2014. Available online at <http://ptolemy.org/systems> (last accessed: Jan. 2017).

APPENDIX A

Audio Interface: Hardware Design & API

This appendix includes *Chisel* and *C* code listings, which implement some of the hardware and software parts of the audio interface designed for Patmos and the WM8731 audio CODEC of the Altera DE2-115 board.

The first 2 Sections, A.1 and A.2, show the input and output buffers respectively. As explained in chapter 4, these are not the only hardware components of the interface, but they are the main ones that have been developed in this project. The rest can be found in the Patmos GitHub repository¹. Finally, Section A.3 shows the main *C* functions which form the software API to access the audio interface from Patmos.

A.1 ADC Buffer

```
1 // FIFO buffer for audio input from WM8731 Audio coded.
2
3
4 package io
5
```

¹<https://github.com/t-crest/patmos/tree/master/hardware/src/io>

```

6 import Chisel._
7
8 class AudioADCBuffer(AUDIOBITLENGTH: Int, MAXADCBUFFERPOWER: Int)
  extends Module {
9
10 // IOs
11 val io = new Bundle {
12 // to/from AudioADC
13   val audioLAdcI = UInt(INPUT, AUDIOBITLENGTH)
14   val audioRAdcI = UInt(INPUT, AUDIOBITLENGTH)
15   val enAdcO = UInt(OUTPUT, 1)
16   val readEnAdcI = UInt(INPUT, 1) //used to sync reads
17 // to/from PATMOS
18   val enAdcI = UInt(INPUT, 1)
19   val audioLPatmosO = UInt(OUTPUT, AUDIOBITLENGTH)
20   val audioRPatmosO = UInt(OUTPUT, AUDIOBITLENGTH)
21   val readPulseI = UInt(INPUT, 1)
22   val emptyO = UInt(OUTPUT, 1) // empty buffer indicator
23   val bufferSizeI = UInt(INPUT, MAXADCBUFFERPOWER+1) // maximum
     bufferSizeI: (2^MAXADCBUFFERPOWER) + 1
24 }
25
26 val BUFFERLENGTH : Int = (Math.pow(2,
     MAXADCBUFFERPOWER)).asInstanceOf[Int]
27
28 //Registers for output audio data (to PATMOS)
29 val audioLReg = Reg(init = UInt(0, AUDIOBITLENGTH))
30 val audioRReg = Reg(init = UInt(0, AUDIOBITLENGTH))
31 io.audioLPatmosO := audioLReg
32 io.audioRPatmosO := audioRReg
33
34 //FIFO buffer registers
35 val audioBufferL = Vec.fill(BUFFERLENGTH) { Reg(init = UInt(0,
     AUDIOBITLENGTH)) }
36 val audioBufferR = Vec.fill(BUFFERLENGTH) { Reg(init = UInt(0,
     AUDIOBITLENGTH)) }
37 val w_pnt = Reg(init = UInt(0, MAXADCBUFFERPOWER))
38 val r_pnt = Reg(init = UInt(0, MAXADCBUFFERPOWER))
39 val fullReg = Reg(init = UInt(0, 1))
40 val emptyReg = Reg(init = UInt(1, 1)) // starts empty
41 io.emptyO := emptyReg
42 val w_inc = Reg(init = UInt(0, 1)) // write pointer increment
43 val r_inc = Reg(init = UInt(0, 1)) // read pointer increment
44
45 // input handshake state machine (from AudioADC)
46 val sInIdle :: sInRead :: Nil = Enum(UInt(), 2)
47 val stateIn = Reg(init = sInIdle)
48 //counter for input handshake
49 val readCntReg = Reg(init = UInt(0, 3))
50 val READCNTLIMIT = UInt(3)
51
52 // output handshake state machine (to Patmos)
53 val sOutIdle :: sOutReading :: Nil = Enum(UInt(), 2)
54 val stateOut = Reg(init = sOutIdle)
55

```

```

56 // full and empty state machine
57 val sFEIdle :: sFEAlmostFull :: sFEFull :: sFEAlmostEmpty ::
    sFEEmpty :: Nil = Enum(UInt(), 5)
58 val stateFE = Reg(init = sFEEmpty)
59
60 // register to keep track of buffer size
61 val bufferSizeReg = Reg(init = UInt(0, MAXADCBUFFERPOWER+1))
62 //update buffer size register
63 when(bufferSizeReg /= io.bufferSizeI) {
64     bufferSizeReg := io.bufferSizeI
65     r_pnt := r_pnt & (io.bufferSizeI - UInt(1))
66     w_pnt := w_pnt & (io.bufferSizeI - UInt(1))
67 }
68
69 // output enable: just wire from input enable
70 io.enAdcO := io.enAdcI
71
72 // audio input handshake: if enable
73 when (io.enAdcI == UInt(1)) {
74     //state machine
75     switch (stateIn) {
76         is (sInIdle) {
77             //wait until posEdge readEnAdcI
78             when(io.readEnAdcI == UInt(1)) {
79                 //wait READCNTLIMIT cycles until input data is written
80                 when(readCntReg == READCNTLIMIT) {
81                     //read input, increment write pointer
82                     audioBufferL(w_pnt) := io.audioLAdcI
83                     audioBufferR(w_pnt) := io.audioRAdcI
84                     w_pnt := (w_pnt + UInt(1)) & (io.bufferSizeI - UInt(1))
85                     w_inc := UInt(1)
86                     //if it is full, write, but increment read pointer too
87                     //to store new samples and dump older ones
88                     when(fullReg == UInt(1)) {
89                         r_pnt := (r_pnt + UInt(1)) & (io.bufferSizeI -
90 UInt(1))
91                         r_inc := UInt(1)
92                     }
93                     //update state
94                     stateIn := sInRead
95                 }
96                 .otherwise {
97                     readCntReg := readCntReg + UInt(1)
98                 }
99             }
100         is (sInRead) {
101             readCntReg := UInt(0)
102             //wait until negEdge readEnAdcI
103             when(io.readEnAdcI == UInt(0)) {
104                 //update state
105                 stateIn := sInIdle
106             }
107         }
108     }

```

```

109 }
110     .otherwise {
111         readCntReg := UInt(0)
112         stateIn := sInIdle
113         w_inc := UInt(0)
114     }
115
116
117
118 // audio output state machine: if enable and not empty
119 when ( (io.enAdcI === UInt(1)) && (emptyReg === UInt(0)) ) {
120     //state machine
121     switch (stateOut) {
122         is (sOutIdle) {
123             when(io.readPulseI === UInt(1)) {
124                 audioLReg := audioBufferL(r_pnt)
125                 audioRReg := audioBufferR(r_pnt)
126                 stateOut := sOutReading
127             }
128         }
129         is (sOutReading) {
130             when(io.readPulseI === UInt(0)) {
131                 r_pnt := (r_pnt + UInt(1)) & (io.bufferSizeI - UInt(1))
132                 r_inc := UInt(1)
133                 stateOut := sOutIdle
134             }
135         }
136     }
137 }
138 .otherwise {
139     stateOut := sOutIdle
140 }
141
142
143
144 //update full and empty states
145 when ( (w_inc === UInt(1)) || (r_inc === UInt(1)) ) {
146     //default: set back variables
147     w_inc := UInt(0)
148     r_inc := UInt(0)
149     //state machine
150     switch (stateFE) {
151         is (sFEIdle) {
152             fullReg := UInt(0)
153             emptyReg := UInt(0)
154             when( (w_inc === UInt(1)) && (w_pnt === ( (r_pnt -
155                 UInt(1)) & (io.bufferSizeI - UInt(1)) ) ) && (r_inc ===
156                 UInt(0)) ) {
157                 stateFE := sFEAlmostFull
158             }
159             .elsewhen( (r_inc === UInt(1)) && (r_pnt === ( (w_pnt -
160                 UInt(1)) & (io.bufferSizeI - UInt(1)) ) ) && (w_inc ===
161                 UInt(0)) ) {
162                 stateFE := sFEAlmostEmpty
163             }
164         }
165     }

```



```

160     }
161     is(sFEAlmostFull) {
162         fullReg := UInt(0)
163         emptyReg := UInt(0)
164         when( (r_inc === UInt(1)) && (w_inc === UInt(0)) ) {
165             stateFE := sFEIdle
166         }
167         .elsewhen( (w_inc === UInt(1)) && (r_inc === UInt(0)) ) {
168             stateFE := sFEFull
169             fullReg := UInt(1)
170         }
171     }
172     is(sFEFull) {
173         fullReg := UInt(1)
174         emptyReg := UInt(0)
175         when( (r_inc === UInt(1)) && (w_inc === UInt(0)) ) {
176             stateFE := sFEAlmostFull
177             fullReg := UInt(0)
178         }
179     }
180     is(sFEAlmostEmpty) {
181         fullReg := UInt(0)
182         emptyReg := UInt(0)
183         when( (w_inc === UInt(1)) && (r_inc === UInt(0)) ) {
184             stateFE := sFEIdle
185         }
186         .elsewhen( (r_inc === UInt(1)) && (w_inc === UInt(0)) ) {
187             stateFE := sFEEmpty
188             emptyReg := UInt(1)
189         }
190     }
191     is(sFEEmpty) {
192         fullReg := UInt(0)
193         emptyReg := UInt(1)
194         when( (w_inc === UInt(1)) && (r_inc === UInt(0)) ) {
195             stateFE := sFEAlmostEmpty
196             emptyReg := UInt(0)
197         }
198     }
199 }
200 }
201 }

```

A.2 DAC Buffer

```

1 // FIFO buffer for audio output to WM8731 Audio coded.
2
3
4 package io
5
6 import Chisel._

```

```

7
8 class AudioDACBuffer(AUDIOBITLENGTH: Int, MAXDACBUFFERPOWER: Int)
  extends Module {
9
10 // IOs
11 val io = new Bundle {
12 // to/from PATMOS
13 val audioLIPatmos = UInt(INPUT, AUDIOBITLENGTH)
14 val audioRIPatmos = UInt(INPUT, AUDIOBITLENGTH)
15 val enDacI = UInt(INPUT, 1) // enable signal
16 val writePulseI = UInt(INPUT, 1)
17 val full0 = UInt(OUTPUT, 1) // full buffer indicator
18 val bufferSizeI = UInt(INPUT, MAXDACBUFFERPOWER+1) // maximum
  bufferSizeI: (2^MAXDACBUFFERPOWER) + 1
19 // to/from AudioDAC
20 val audioLIDAC = UInt(OUTPUT, AUDIOBITLENGTH)
21 val audioRIDAC = UInt(OUTPUT, AUDIOBITLENGTH)
22 val enDacO = UInt(OUTPUT, 1) // enable signal
23 val writeEnDacI = UInt(INPUT, 1) // used to sync writes
24 val convEndI = UInt(INPUT, 1) // indicates end of conversion
25 }
26
27 val BUFFERLENGTH : Int = (Math.pow(2,
  MAXDACBUFFERPOWER)).asInstanceOf[Int]
28
29 //Registers for output audio data
30 val audioLIReg = Reg(init = UInt(0, AUDIOBITLENGTH))
31 val audioRIReg = Reg(init = UInt(0, AUDIOBITLENGTH))
32 io.audioLIDAC := audioLIReg
33 io.audioRIDAC := audioRIReg
34
35 //FIFO buffer registers
36 val audioBufferL = Vec.fill(BUFFERLENGTH) { Reg(init = UInt(0,
  AUDIOBITLENGTH)) }
37 val audioBufferR = Vec.fill(BUFFERLENGTH) { Reg(init = UInt(0,
  AUDIOBITLENGTH)) }
38 val w_pnt = Reg(init = UInt(0, MAXDACBUFFERPOWER))
39 val r_pnt = Reg(init = UInt(0, MAXDACBUFFERPOWER))
40 val fullReg = Reg(init = UInt(0, 1))
41 val emptyReg = Reg(init = UInt(1, 1)) // starts empty
42 io.full0 := fullReg
43 val w_inc = Reg(init = UInt(0, 1)) // write pointer increment
44 val r_inc = Reg(init = UInt(0, 1)) // read pointer increment
45
46
47 // output handshake state machine
48 val sOutIdle :: sOutWrote :: Nil = Enum(UInt(), 2)
49 val stateOut = Reg(init = sOutIdle)
50
51 // input handshake state machine
52 val sInIdle :: sInWriting :: Nil = Enum(UInt(), 2)
53 val stateIn = Reg(init = sInIdle)
54
55 // full and empty state machine

```

```

56 | val sFEIdle :: sFEAlmostFull :: sFEFull :: sFEAlmostEmpty ::
    |   sFEEmpty :: Nil = Enum(UInt(), 5)
57 | val stateFE = Reg(init = sFEEmpty)
58 |
59 | //output enable register: For AudioDAC and for output conversion
60 | val enOutReg = Reg(init = UInt(0, 1)) //starts low because its
    |   empty
61 | io.enDac0 := enOutReg
62 | val lastOutputReg = Reg(init = UInt(0, 1)) // indicator of last
    |   output conversion
63 |
64 | // state machine for last output conversion
65 | val sCEWaitFirst :: sCEFirstPulse :: sCEWaitSecond :: Nil =
    |   Enum(UInt(), 3)
66 | val stateCE = Reg(init = sCEWaitFirst)
67 |
68 | // register to keep track of buffer size
69 | val bufferSizeReg = Reg(init = UInt(0, MAXDACBUFFERPOWER+1))
70 | //update buffer size register
71 | when(bufferSizeReg /= io.bufferSizeI) {
72 |   bufferSizeReg := io.bufferSizeI
73 |   r_pnt := r_pnt & (io.bufferSizeI - UInt(1))
74 |   w_pnt := w_pnt & (io.bufferSizeI - UInt(1))
75 | }
76 |
77 | // audio output handshake: if output handshake enabled
78 | when (enOutReg === UInt(1)) {
79 |   //state machine
80 |   switch (stateOut) {
81 |     is (sOutIdle) {
82 |       //wait until posEdge writeEnDacI
83 |       when(io.writeEnDacI === UInt(1)) {
84 |         // write only when its not empty (for last conversion
    |   case)
85 |         when(emptyReg === UInt(0)) {
86 |           //write output, increment read pointer
87 |           audioLIReg := audioBufferL(r_pnt)
88 |           audioRIReg := audioBufferR(r_pnt)
89 |           r_pnt := (r_pnt + UInt(1)) & (io.bufferSizeI - UInt(1))
90 |           r_inc := UInt(1)
91 |         }
92 |         //update state
93 |         stateOut := sOutWrote
94 |       }
95 |     }
96 |     is (sOutWrote) {
97 |       //wait until negEdge writeEnDacI
98 |       when(io.writeEnDacI === UInt(0)) {
99 |         //update state
100 |         stateOut := sOutIdle
101 |       }
102 |     }
103 |   }
104 | }
105 | .otherwise {

```

```

106     stateOut := sOutIdle
107     r_inc := UInt(0)
108 }
109
110
111
112 // audio input handshake: if enable and not full
113 when ( (io.enDacI === UInt(1)) && (fullReg === UInt(0)) ) {
114     //state machine
115     switch (stateIn) {
116         is (sInIdle) {
117             when(io.writePulseI === UInt(1)) {
118                 audioBufferL(w_pnt) := io.audioLIPatmos
119                 audioBufferR(w_pnt) := io.audioRIPatmos
120                 stateIn := sInWriting
121             }
122         }
123         is (sInWriting) {
124             when(io.writePulseI === UInt(0)) {
125                 w_pnt := (w_pnt + UInt(1)) & (io.bufferSizeI - UInt(1))
126                 w_inc := UInt(1)
127                 stateIn := sInIdle
128             }
129         }
130     }
131 }
132 .otherwise {
133     stateIn := sInIdle
134 }
135
136
137
138 //update output handshake enable register
139 when (emptyReg === UInt(0)) { //if not empty, always enable
140     enOutReg := UInt(1)
141 }
142 .otherwise { // empty
143     when (lastOutputReg === UInt(1)) { // if last output
144         conversion, enable
145         enOutReg := UInt(1)
146     }
147     .otherwise {
148         enOutReg := UInt(0)
149     }
150 }
151
152 // when last conversion finishes:
153 when(lastOutputReg === UInt(1)) {
154     // if stateFE is not empty anymore, or if it is but conversion
155     ends
156     when(stateFE /= sFEEmpty) { // if state is not empty anymore
157         lastOutputReg := UInt(0)
158     }
159     .otherwise { // state machine to detect 2nd convEndI pulse
160         switch (stateCE) {

```

```

159     is (sCEWaitFirst) {
160         when(io.convEndI === UInt(1)) {
161             stateCE := sCEFirstPulse
162         }
163     }
164     is (sCEFirstPulse) {
165         when(io.convEndI === UInt(0)) {
166             stateCE := sCEWaitSecond
167         }
168     }
169     is (sCEWaitSecond) {
170         when(io.convEndI === UInt(1)) {
171             lastOutputReg := UInt(0)
172             stateCE := sCEWaitFirst
173         }
174     }
175 }
176 }
177 }
178 .otherwise {
179     stateCE := sCEWaitFirst
180 }
181
182
183
184
185 //update full and empty states
186 when ( (w_inc === UInt(1)) || (r_inc === UInt(1)) ) {
187     //default: set back variables
188     w_inc := UInt(0)
189     r_inc := UInt(0)
190     //state machine
191     switch (stateFE) {
192         is (sFEIdle) {
193             fullReg := UInt(0)
194             emptyReg := UInt(0)
195             when( (w_inc === UInt(1)) && (w_pnt === ( (r_pnt -
196                 UInt(1)) & (io.bufferSizeI - UInt(1)) ) ) && (r_inc ===
197                 UInt(0)) ) {
198                 stateFE := sFEAlmostFull
199             }
200             .elsewhen( (r_inc === UInt(1)) && (r_pnt === ( (w_pnt -
201                 UInt(1)) & (io.bufferSizeI - UInt(1)) ) ) && (w_inc ===
202                 UInt(0)) ) {
203                 stateFE := sFEAlmostEmpty
204             }
205         }
206         is(sFEAlmostFull) {
207             fullReg := UInt(0)
208             emptyReg := UInt(0)
209             when( (r_inc === UInt(1)) && (w_inc === UInt(0)) ) {
210                 stateFE := sFEIdle
211             }
212             .elsewhen( (w_inc === UInt(1)) && (r_inc === UInt(0)) ) {
213                 stateFE := sFEFull

```

```

210         fullReg := UInt(1)
211     }
212 }
213 is(sFEFull) {
214     fullReg := UInt(1)
215     emptyReg := UInt(0)
216     when( (r_inc === UInt(1)) && (w_inc === UInt(0)) ) {
217         stateFE := sFEAlmostFull
218         fullReg := UInt(0)
219     }
220 }
221 is(sFEAlmostEmpty) {
222     fullReg := UInt(0)
223     emptyReg := UInt(0)
224     when( (w_inc === UInt(1)) && (r_inc === UInt(0)) ) {
225         stateFE := sFEIdle
226     }
227     .elsewhen( (r_inc === UInt(1)) && (w_inc === UInt(0)) ) {
228         stateFE := sFEEmpty
229         emptyReg := UInt(1)
230         lastOutputReg := UInt(1) // indicator of last output
231     conversion
232     }
233 }
234 is(sFEEmpty) {
235     fullReg := UInt(0)
236     emptyReg := UInt(1)
237     when( (w_inc === UInt(1)) && (r_inc === UInt(0)) ) {
238         stateFE := sFEAlmostEmpty
239         emptyReg := UInt(0)
240     }
241 }
242 }
243 }

```

A.3 API Functions

```

1  /*
2  * @brief   Writes the supplied data to the address register,
3  * sets the request signal and waits for the acknowledge signal.
4  * @param[in] addr the address of which register to write to.
5  * Has to be 7 bit long.
6  * @param[in] data the data thats supposed to be written.
7  * Has to be 9 Bits long
8  * @reutrnrn returns 0 if successful and a negative number if
9  * there was an error.
10 */
11 int writeToI2C(char* addrC, char* dataC) {
12     int addr = 0;
13     int data = 0;

```

```

13
14 //Convert binary String of address to int
15 for(int i = 0; i < 7; i++) {
16     addr *= 2;
17     if (*addrC++ == '1') addr += 1;
18 }
19
20 //Convert binary String of data to int
21 for(int i = 0; i < 9; i++) {
22     data *= 2;
23     if (*dataC++ == '1') data += 1;
24 }
25
26 //Debug info:
27 printf("Sending Data: %i to address %i\n",data,addr);
28
29 *i2cDataReg = data;
30 *i2cAdrReg  = addr;
31 *i2cReqReg  = 1;
32
33
34 while(*i2cAckReg == 0) {
35     printf("Waiting ...\n");
36     //Maybe input something like a timeout ...
37 }
38 for (int i = 0; i<200; i++) { *i2cReqReg=0; }
39
40 printf("success\n");
41
42 return 0;
43 }
44
45 /*
46 * @brief Sets the default values
47 * @param[in] guitar    used to select the input: line in or mic in
48 */
49
50 void setup(int guitar) {
51
52     /*
53     //-----Line in-----
54     char *addrLeftIn  = "0000000";
55     char *dataLineIn  = "100010111"; //disable Mute, Enable
56         Simultaneous Load, LinVol: 10111 - Set volume to 23 (of 31)
57     writeToI2C(addrLeftIn,dataLineIn);
58
59     char *addrRigthIn = "00000001";
60     writeToI2C(addrRigthIn,dataLineIn);
61     */
62
63     //-----Headphones-----
64     char *addrLeftHead = "0000010";
65     char *dataHeadphone = "001111001"; // disable simultaneous
66         loads, zero cross disable, LinVol: 1111001 (0db)
67     writeToI2C(addrLeftHead,dataHeadphone);

```

```

66
67 char *addrRightHead = "0000011";
68 writeToI2C(addrRightHead,dataHeadphone);
69
70 //-----Analogue Audio Path Control-----
71 char *addrAnalogue = "0000100";
72 if(guitar == 0) {
73     char *dataAnalogue = "000010010"; //DAC selected, rest
74     disabled, MIC muted, Line input select to ADC
75     writeToI2C(addrAnalogue,dataAnalogue);
76 }
77 else {
78     char *dataAnalogueGuit = "000010101"; //MIC selected to ADC,
79     MIC enabled, MIC boost enabled
80     writeToI2C(addrAnalogue,dataAnalogueGuit);
81 }
82
83 //-----Digital Audio Path Control-----
84 char *addrDigital = "0000101";
85 char *dataDigital = "000000001"; //disable soft mute and
86     disable de-emphasis, disable highpass filter
87 writeToI2C(addrDigital,dataDigital);
88
89 //-----Digital Audio Interface Format-----
90 char *addrInterface = "0000111";
91 char *dataInterface = "000010011"; //BCLK not inverted, slave,
92     right-channel-right-data, LRP = A mode for DSP, 16-bit audio,
93     DSP mode
94 writeToI2C(addrInterface,dataInterface);
95
96 //-----Sampling Control-----
97 char *addrSample = "0001000";
98 char *dataSample = "000000000"; //USB mode, BOSR=1, Sample Rate
99     = 44.1 kHz both for ADC and DAC
100 writeToI2C(addrSample,dataSample);
101
102 printf("FINISHED SETUP!\n");
103 }
104
105 int isPowerOfTwo (unsigned int x) {
106     while (((x % 2) == 0) && x > 1) /* While x is even and > 1 */
107         x /= 2;
108     return (x == 1);
109 }
110
111 /*
112 * @brief sets the size of the input (ADC) buffer. Must be a power
113 * of 2
114 * @param[in] bufferSize length of the buffer
115 * @return returns 0 if successful and a 1 if there was an error.
116 */

```



```

114 int setInputBufferSize(int bufferSize) {
115     if(isPowerOfTwo(bufferSize)) {
116         printf("Input buffer size set to %d\n", bufferSize);
117         *audioAdcBufferSizeReg = bufferSize;
118         return 0;
119     }
120     else {
121         printf("ERROR: Buffer Size must be power of 2\n");
122         return 1;
123     }
124 }
125
126 /*
127  * @brief sets the size of the output (DAC) buffer. Must be a
128     power of 2
129  * @param[in] bufferSize length of the buffer
130  * @return returns 0 if successful and a 1 if there was an error.
131 */
132 int setOutputBufferSize(int bufferSize) {
133     if(isPowerOfTwo(bufferSize)) {
134         printf("Output buffer size set to %d\n", bufferSize);
135         *audioDacBufferSizeReg = bufferSize;
136         return 0;
137     }
138     else {
139         printf("ERROR: Buffer Size must be power of 2\n");
140         return 1;
141     }
142 }
143 /*
144  * @brief reads data from the input (ADC) buffer into Patmos
145  * @param[in] *l pointer to left audio data
146  * @param[in] *r pointer to right audio data
147  * @return returns 0 if successful
148 */
149 int getInputBufferSPM(volatile _SPM short *l, volatile _SPM short
    *r) {
150     while(*audioAdcBufferEmptyReg == 1); // wait until not empty
151     *audioAdcBufferReadPulseReg = 1; // begin pulse
152     *audioAdcBufferReadPulseReg = 0; // end pulse
153     *l = *audioAdcLReg;
154     *r = *audioAdcRReg;
155     return 0;
156 }
157
158 /*
159  * @brief writes data from patmos into the output (DAC) buffer
160  * @param[in] l left audio data
161  * @param[in] r right audio data
162  * @return returns 0 if successful
163 */
164 int setOutputBufferSPM(volatile _SPM short *l, volatile _SPM short
    *r) {
165     //write data first: it will stay in AudioInterface, won't go to

```

```
166 //AudioDacBuffer until the write pulse
167 *audioDacLReg = *l;
168 *audioDacRReg = *r;
169 while(*audioDacBufferFullReg == 1); // wait until not full
170 *audioDacBufferWritePulseReg = 1; // begin pulse
171 *audioDacBufferWritePulseReg = 0; // end pulse
172
173 return 0;
174 }
```

APPENDIX B

Examples of some Audio Effects

This appendix shows the C implementation for Patmos of some of the audio effects that have been used in this project. The effects shown here are the Filter (B.1), the Delay (B.2), the Wah-wah (B.3) and the Overdrive (B.4). These are not the only effects that have been implemented, but it is enough to show these ones here to understand the implementation. All the effects implemented can be found in 2 C files of the Patmos GitHub repository: *audio.c*¹ and *dsp_algorithms.c*².

Some words to explain each of these effects have been given in chapter 5. For each effect, the parts of code shown here are the effect `struct` (i.e. the class), the allocation function and the processing function. Additionally, the DSP algorithm implementation function might be shown when needed.

B.1 Filter

```
1 struct Filter {  
2     int    accum[2]; //accumulator accum[2]
```

¹<https://github.com/t-crest/patmos/tree/master/c/libaudio/audio.c>

²https://github.com/t-crest/patmos/tree/master/c/libaudio/dsp_algorithms.c

```

3  short x_buf[3][2]; // input buffer
4  short y_buf[3][2]; // output buffer
5  short A[3]; // [a2, a1, 1]
6  short B[3]; // [b2, b1, b0]
7  int pnt; //audio input pointer
8  int sftLft; //x or y buffer pointer
9  int type; // to choose between HP, LP, BP or BR
10 };

```

```

1  unsigned int alloc_filter_vars(_SPM struct Filter *filtP, unsigned
   int LAST_ADDR, int Fc, float QorFb, int thisType) {
2
3  //calculate filter coefficients (2nd order)
4  filtP->type = thisType;
5  if (filtP->type < 2) { //HP or LP
6      filter_coeff_hp_lp(3, filtP->B, filtP->A, Fc, QorFb,
7      &filtP->sftLft, 0, thisType); //type: HPF or LPF
8  }
9  else { // 2 or 3: BP or BR
10     filter_coeff_bp_br(3, filtP->B, filtP->A, Fc, (int)QorFb,
11     &filtP->sftLft, 0);
12 }
13
14 filtP->pnt = 2;
15
16 LAST_ADDR += (sizeof(struct Filter));
17
18 return LAST_ADDR;
19 }

```

```

1  int audio_filter(_SPM struct Filter *filtP, volatile _SPM short
   *xP, volatile _SPM short *yP) {
2  //increment pointer
3  filtP->pnt = ( filtP->pnt + 1 ) % 3;
4  //first, read sample
5  filtP->x_buf[filtP->pnt][0] = xP[0];
6  filtP->x_buf[filtP->pnt][1] = xP[1];
7  //then, calculate filter
8  filterIIR_2nd(&filtP->pnt, filtP->x_buf, filtP->y_buf,
9  filtP->accum, filtP->B, filtP->A, &filtP->sftLft);
10 //check if it is BP/BR
11 if(filtP->type == 2) { //BP
12     filtP->accum[0] = ( (int)xP[0] -
13     (int)filtP->y_buf[filtP->pnt][0] ) >> 1;
14     filtP->accum[1] = ( (int)xP[1] -
15     (int)filtP->y_buf[filtP->pnt][1] ) >> 1;
16 }
17 else {
18     if(filtP->type == 3) { //BR
19         filtP->accum[0] = ( (int)xP[0] +
20         (int)filtP->y_buf[filtP->pnt][0] ) >> 1;
21         filtP->accum[1] = ( (int)xP[1] +
22         (int)filtP->y_buf[filtP->pnt][1] ) >> 1;
23     }
24 }

```

```

19     else { //HP or LP
20         filtP->accum[0] = filtP->y_buf[filtP->pnt][0];
21         filtP->accum[1] = filtP->y_buf[filtP->pnt][1];
22     }
23 }
24 //set output
25 yP[0] = (short)filtP->accum[0];
26 yP[1] = (short)filtP->accum[1];
27
28 return 0;
29 }

```

```

1 int filterIIR_2nd(_SPM int *pnt_i, _SPM short (*x)[2], _SPM short
  (*y)[2], _SPM int *accum, _SPM short *B, _SPM short *A, _SPM
  int *shiftLeft) {
2     int pnt; //pointer for x_filter
3     accum[0] = 0;
4     accum[1] = 0;
5     for(int i=0; i<3; i++) { //FILTER_ORDER_1PLUS = 3
6         pnt = (*pnt_i + i + 1) % 3; //FILTER_ORDER_1PLUS = 3
7         // SIGNED SHIFT (arithmetical): losing a 2-bit resolution
8         accum[0] += (B[i]*x[pnt][0]) >> 2;
9         accum[0] -= (A[i]*y[pnt][0]) >> 2;
10        accum[1] += (B[i]*x[pnt][1]) >> 2;
11        accum[1] -= (A[i]*y[pnt][1]) >> 2;
12    }
13    //accumulator limits: [ (2^(30-2-1))-1 , -(2^(30-2-1)) ]
14    //accumulator limits: [ 0x7FFFFFFF, 0x80000000 ]
15    // digital saturation
16    for(int i=0; i<2; i++) {
17        if (accum[i] > 0x7FFFFFFF) {
18            accum[i] = 0x7FFFFFFF;
19        }
20        else {
21            if (accum[i] < -0x80000000) {
22                accum[i] = -0x80000000;
23            }
24        }
25    }
26    y[*pnt_i][0] = accum[0] >> (13-*shiftLeft);
27    y[*pnt_i][1] = accum[1] >> (13-*shiftLeft);
28
29    return 0;
30 }

```

B.2 Delay

```

1 struct IIRdelay {
2     int accum[2]; //accumulator accum[2]
3     short g[2]; //gains [g1, g0]

```

```

4   int    del[2]; // delays [d1, d0]
5   int    pnt; //audio input pointer
6   //SRAM Memory variables
7   short (*audio_buf)[DELAY_L]; // audio_buf[2][DELAY_L]
8 };

```

```

1  unsigned int alloc_delay_vars(_SPM struct IIRdelay *delP, unsigned
   int LAST_ADDR) {
2
3     //initialise delay variables
4     //set gains: for comb delay:
5     delP->g[1] = ONE_16b; // g0 = 1
6     delP->g[0] = ONE_16b * 0.5; // g1 = 0.5
7     //set delays:
8     delP->del[1] = 0; // always d0 = 0
9     delP->del[0] = DELAY_L - 1; // d1 = as long as delay buffer
10    //pointer starts on top
11    delP->pnt = DELAY_L - 1;
12
13    //alloc audio array
14    delP->audio_buf = malloc(DELAY_L * 2 * sizeof(short)); //
   short audio_buf[2][DELAY_L]
15
16    //empty buffer
17    for(int i=0; i<DELAY_L; i++) {
18        delP->audio_buf[0][i] = 0;
19        delP->audio_buf[1][i] = 0;
20    }
21
22    LAST_ADDR += (sizeof(struct IIRdelay));
23
24    return LAST_ADDR;
25 }

```

```

1  int audio_delay(_SPM struct IIRdelay *delP, volatile _SPM short
   *xP, volatile _SPM short *yP) {
2     //first, read sample
3     delP->audio_buf[0][delP->pnt] = xP[0];
4     delP->audio_buf[1][delP->pnt] = xP[1];
5     //calculate IIR comb filter
6     combFilter_1st(DELAY_L, &delP->pnt, delP->audio_buf, yP,
   delP->accum, delP->g, delP->del);
7     //replace content on buffer
8     delP->audio_buf[0][delP->pnt] = yP[0];
9     delP->audio_buf[1][delP->pnt] = yP[1];
10    //update pointer
11    if(delP->pnt == 0) {
12        delP->pnt = DELAY_L - 1;
13    }
14    else {
15        delP->pnt = delP->pnt - 1;
16    }
17
18    return 0;

```

```
19 }
```

```

1  __attribute__((always_inline))
2  int combFilter_1st(int AUDIO_BUF_LEN, _SPM int *pnt, short
   (*audio_buffer)[AUDIO_BUF_LEN], volatile _SPM short *y, _SPM
   int *accum, _SPM short *g, _SPM int *del) {
3     accum[0] = 0;
4     accum[1] = 0;
5     int audio_pnt = (*pnt+del[0])%AUDIO_BUF_LEN;
6     accum[0] += (g[0]*audio_buffer[0][audio_pnt]) >> 2;
7     accum[1] += (g[0]*audio_buffer[1][audio_pnt]) >> 2;
8     audio_pnt = (*pnt+del[1])%AUDIO_BUF_LEN;
9     accum[0] += (g[1]*audio_buffer[0][audio_pnt]) >> 2;
10    accum[1] += (g[1]*audio_buffer[1][audio_pnt]) >> 2;
11    //accumulator limits: [ (2^(30-2-1))-1 , -(2^(30-2-1)) ]
12    //accumulator limits: [ 0x7FFFFFFF, 0x8000000 ]
13    // digital saturation
14    for(int i=0; i<2; i++) {
15        if (accum[i] > 0x7FFFFFFF) {
16            accum[i] = 0x7FFFFFFF;
17        }
18        else {
19            if (accum[i] < -0x8000000) {
20                accum[i] = -0x8000000;
21            }
22        }
23    }
24    y[0] = accum[0] >> 13;
25    y[1] = accum[1] >> 13;
26
27    return 0;
28 }
```

B.3 WahWah

```

1  struct WahWah {
2     int accum[2]; //accumulator accum[2]
3     short x_buf[3][2]; // input buffer
4     short y_buf[3][2]; // output buffer
5     short A[3]; // [a2, a1, 1]
6     short B[3]; // [b2, b1, b0]
7     int pnt; //audio input pointer
8     int wah_pnt; //modulation pointer
9     int sftLft; //x or y buffer pointer
10    //SRAM Memory Variables
11    int *fc_array; // fc_array[WAHWAH_P]
12    int *fb_array; // fb_array[WAHWAH_P]
13    short (*a_array)[WAHWAH_P]; //for A coefficients:
   a_array[3][WAHWAH_P]
```

```

14     short (*b_array)[WAHWAH_P]; //for B coefficients:
15     b_array[3][WAHWAH_P]
};

```

```

1 unsigned int alloc_wahwah_vars(_SPM struct WahWah *wahP, unsigned
  int LAST_ADDR) {
2
3     //shift left is fixed!
4     wahP->sftLft = 1;
5
6     //modulation arrays
7     wahP->fc_array = malloc(WAHWAH_P * sizeof(int)); // int
  fc_array[WAHWAH_P]
8     wahP->fb_array = malloc(WAHWAH_P * sizeof(int)); // int
  fb_array[WAHWAH_P]
9     wahP->a_array = malloc(WAHWAH_P * 3 * sizeof(short)); //
  short a_array[3][WAHWAH_P]
10    wahP->b_array = malloc(WAHWAH_P * 3 * sizeof(short)); //
  short b_array[3][WAHWAH_P]
11
12    //store sin Arrays of Fc and Fb
13    storeSin(wahP->fc_array, WAHWAH_P, WAHWAH_FC_CEN,
  WAHWAH_FC_AMP);
14    storeSin(wahP->fb_array, WAHWAH_P, WAHWAH_FB_CEN,
  WAHWAH_FB_AMP);
15
16    //calculate band-pass filter coefficients
17    for(int i=0; i<WAHWAH_P; i++) {
18        filter_coeff_bp_br(3, wahP->B, wahP->A, wahP->fc_array[i],
  wahP->fb_array[i], &wahP->sftLft, 1);
19        wahP->b_array[2][i] = wahP->B[2];
20        wahP->b_array[1][i] = wahP->B[1];
21        wahP->b_array[0][i] = wahP->B[0];
22        wahP->a_array[2][i] = 0;
23        wahP->a_array[1][i] = wahP->A[1];
24        wahP->a_array[0][i] = wahP->A[0];
25    }
26
27    wahP->wah_pnt = 2;
28
29    LAST_ADDR += (sizeof(struct WahWah));
30
31    return LAST_ADDR;
32 }

```

```

1 int audio_wahwah(_SPM struct WahWah *wahP, volatile _SPM short
  *xP, volatile _SPM short *yP) {
2     //update filter coefficients
3     wahP->B[2] = wahP->b_array[2][wahP->wah_pnt]; //b0
4     wahP->B[1] = wahP->b_array[1][wahP->wah_pnt]; //b1
5     // b2 doesnt need to be updated: always 1
6     wahP->A[1] = wahP->a_array[1][wahP->wah_pnt]; //a1
7     wahP->A[0] = wahP->a_array[0][wahP->wah_pnt]; //a2
8     //update pointers

```



```

9   wahP->wah_pnt = (wahP->wah_pnt+1) % WAHWAH_P;
10  wahP->pnt = (wahP->pnt+1) % 3;
11  //first, read sample
12  wahP->x_buf[wahP->pnt][0] = xP[0];
13  wahP->x_buf[wahP->pnt][1] = xP[1];
14  //then, calculate filter
15  filterIIR_2nd(&wahP->pnt, wahP->x_buf, wahP->y_buf,
16  wahP->accum, wahP->B, wahP->A, &wahP->sftLft);
17  //Band-Pass stuff
18  wahP->accum[0] = ( (int)xP[0] - (int)wahP->y_buf[wahP->pnt][0]
19  );
20  wahP->accum[1] = ( (int)xP[1] - (int)wahP->y_buf[wahP->pnt][1]
21  );
22  //mix with original: gains are fixed
23  wahP->accum[0] = ( (int)(WAHWAH_WET_GAIN*wahP->accum[0]) >> 15
24  ) + ( (int)(WAHWAH_DRY_GAIN*xP[0]) >> 15 );
25  wahP->accum[1] = ( (int)(WAHWAH_WET_GAIN*wahP->accum[1]) >> 15
26  ) + ( (int)(WAHWAH_DRY_GAIN*xP[1]) >> 15 );
27  //set output
28  yP[0] = (short)wahP->accum[0];
29  yP[1] = (short)wahP->accum[1];
30
31  return 0;
32 }

```

B.4 Overdrive

```

1  struct Overdrive {
2     int   accum[2]; //accumulator accum[2]
3  };

```

```

1  unsigned int alloc_overdrive_vars(_SPM struct Overdrive *odP,
2     unsigned int LAST_ADDR) {
3
4     LAST_ADDR += (sizeof(struct Overdrive));
5
6     return LAST_ADDR;
7  }

```

```

1  int audio_overdrive(_SPM struct Overdrive *odP, volatile _SPM
2     short *xP, volatile _SPM short *yP) {
3     //THRESHOLD IS 1/3 = 0x2AAB
4     //input abs:
5     unsigned int x_abs[2];
6     for(int j=0; j<2; j++) {
7         x_abs[j] = abs(xP[j]);
8         if(x_abs[j] > (2 * 0x2AAB)) { // saturation : y = 1
9             if (xP[j] > 0) {
10                yP[j] = 0x7FFF;
11            }
12        }
13    }
14 }

```

```

10         }
11         else {
12             yP[j] = 0x8000;
13         }
14     }
15     else {
16         if(x_abs[j] > 0x2AAB) { // smooth overdrive: y = ( 3 -
(2-3*x)^2 ) / 3;
17             odP->accum[j] = (0x17FFF * x_abs[j]) >> 15 ; //
result is 1 sign + 17 bits
18             odP->accum[j] = 0xFFFF - odP->accum[j];
19             odP->accum[j] = (odP->accum[j] * odP->accum[j]) >>
15;
20             odP->accum[j] = 0x17FFF - odP->accum[j];
21             odP->accum[j] = (odP->accum[j] * 0x2AAB) >> 15;
22             if(xP[j] > 0) { //positive
23                 if(odP->accum[j] > 32767) {
24                     yP[j] = 32767;
25                 }
26                 else {
27                     yP[j] = odP->accum[j];
28                 }
29             }
30             else { // negative
31                 yP[j] = -odP->accum[j];
32             }
33         }
34         else { // linear zone: y = 2*x
35             yP[j] = xP[j] << 1;
36         }
37     }
38 }
39
40 return 0;
41 }

```

APPENDIX C

Audio Processing Function in the Multicore Platform

This appendix shows 2 C files and one XML file. The C files are some of the most important ones related to the multicore audio processing platform. The first one, shown in Section C.1, is the C implementation of the main audio processing function for the T-CREST multicore platform. This function is named `audio_process`, and can be found in the `audio.c`¹ file of the `libaudio` library for Patmos. Some parts of it are hidden (to reduce the extension of the code), which are related to the processing of each one of the implemented audio effects, some of which are shown in appendix B. The second one, shown in Section C.2, is an example of the `audioinit.h` header file containing the description of the effect setup, and generated by the task allocator. It corresponds to the audio application shown in Listing 7.2. Finally, the XML file is the custom NoC configuration XML file generated by the scheduler, also corresponding to Listing 7.2.

C.1 `audio_process` function

```
1 int audio_process(struct AudioFX *audioP) {  
2     int retval = 0;
```

¹<https://github.com/t-crest/patmos/tree/master/c/libaudio/audio.c>

```

3  /* -----X and Y locations----- */
4  volatile _SPM short * xP;
5  volatile _SPM short * yP;
6  if(*audioP->in_con != NOC) { //same core : data==*x_pnt
7      xP = (volatile _SPM short *)*(audioP->x_pnt+0);
8  }
9  if(*audioP->out_con != NOC) { //same core: data==*y_pnt
10     yP = (volatile _SPM short *)*(audioP->y_pnt+0);
11 }
12
13 else { //NoC: data==*y_pnt
14     yP = (volatile _SPM short *)*(_SPM unsigned int *)
15         *(audioP->y_pnt+0);
16 }
17
18 /* -----SEND/PROCESS/RECEIVE-----*/
19
20 unsigned int ind; //index used for each operation
21 unsigned int offs; // can be x_offs or y_offs, depending on XgY
    or XlY
22
23 switch(*audioP->pt) {
24 case XeY:
25     //check if it is 0, is last and needs to wait due to latency
26     if( (*audioP->cpuid != 0) || (*audioP->out_con == SAME) ||
27         (*audioP->out_con == NOC) || (*audioP->last_init == 0) ) {
28         //RECEIVE ONCE
29         if(*audioP->in_con == NOC) {
30             //receive from all recv channels
31             for(int i=0; i<*audioP->recv_am; i++) {
32                 if(mp_recv((cpd_t *)*(audioP->recvChanP+i),
33                     TIMEOUT) == 0) {
34                     if(*audioP->cpuid == 0) {
35                         printf("RCV TIMED OUT!\n");
36                     }
37                     retval = 1;
38                 }
39             }
40             //update X pointer after each recv
41             xP = (volatile _SPM short *)*(_SPM unsigned int *)
42                 *(audioP->x_pnt+0);
43             //after receiving, add all signals into recvChanel[0]
44             int shift_am = *audioP->recv_am - 1;
45             for(int i=1; i<(*audioP->recv_am); i++) {
46                 volatile _SPM short * xnxtP =
47                     (volatile _SPM short *)*(_SPM unsigned int *)
48                     *(audioP->x_pnt+i);
49                 for(int j=0; j<(*audioP->xb_size); j++) {
50                     xP[2*j] = (xP[2*j]>>shift_am) +
51                         (xnxtP[2*j]>>shift_am);
52                     xP[2*j+1] = (xP[2*j+1]>>shift_am) +
53                         (xnxtP[2*j+1]>>shift_am);
54                 }
55             }
56         }

```

```

57     else { //same core
58         if( (*audioP->cpuid == 0) &&
59             (*audioP->in_con == FIRST) ) {
60             audioIn(audioP, xP);
61         }
62     }
63     //PROCESS Nf TIMES
64     switch(*audioP->fx) {
65     case DRY:
66         for(unsigned int i=0; i < *audioP->Nf; i++) {
67             ind = 2 * i * (*audioP->s);
68             audio_dry(&xP[ind], &yP[ind]);
69         }
70         break;
71     case DELAY: ;
72         _SPM_struct IIRdelay *delP = (_SPM_struct IIRdelay *)
73             *audioP->fx_pnt;
74         for(unsigned int i=0; i < *audioP->Nf; i++) {
75             ind = 2 * i * (*audioP->s);
76             audio_delay(delP, &xP[ind], &yP[ind]);
77         }
78         break;
79         /*
80         MORE EFFECTS HERE: NOT SHOWN
81         */
82
83     default:
84         if(get_cpuid() == 0) {
85             printf("effect not implemented yet\n");
86         }
87         break;
88     }
89     //ACKNOWLEDGE ONCE AFTER PROCESSING
90     if(*audioP->in_con == NOC) {
91         //acknowledge to all recv channels
92         for(int i=0; i<*audioP->recv_am; i++) {
93             if(mp_ack((qpd_t *)*(audioP->recvChanP+i),
94                 TIMEOUT) == 0) {
95                 if(*audioP->cpuid == 0) {
96                     printf("ACK TIMED OUT!\n");
97                 }
98                 retval = 1;
99             }
100         }
101     }
102     //SEND ONCE
103     if(*audioP->out_con == NOC) { //send to NoC
104         //before sending, copy send data to all send channel
105         buffers
106         for(int i=1; i<(*audioP->send_am); i++) {
107             volatile _SPM_short * ynxtP =
108                 (volatile _SPM_short *)*( _SPM_unsigned_int *)
109                 *(audioP->y_pnt+i);
110             for(int j=0; j<(*audioP->yb_size); j++) {
111                 ynxtP[2*j] = yP[2*j];

```

```

111         ynxtP[2*j+1] = yP[2*j+1];
112     }
113 }
114 //send to all send channels
115 for(int i=0; i<(*audioP->send_am); i++) {
116     if(mp_send((qpd_t *)*(audioP->sendChanP+i),
117             TIMEOUT) == 0) {
118         if(*audioP->cpuid == 0) {
119             printf("SEND TIMED OUT!\n");
120         }
121         retval = 1;
122     }
123 }
124 }
125 else { //same core
126     if( (*audioP->cpuid == 0) &&
127         (*audioP->out_con == LAST) ) {
128         audioOut(audioP, yP);
129     }
130 }
131 }
132 //if it is last and needs to wait
133 else {
134     *audioP->last_count = *audioP->last_count + 1;
135     if(*audioP->last_count == *audioP->latency) {
136         *audioP->last_init = 0;
137         //printf("latency limit reached!\n");
138     }
139 }
140 break;
141 case XgY:
142     //RECEIVE ONCE
143     if(*audioP->in_con == NOC) { //receive from NoC
144         //receive from all recv channels
145         for(int i=0; i<*audioP->recv_am; i++) {
146             if(mp_recv((qpd_t *)*(audioP->recvChanP+i),
147                     TIMEOUT) == 0) {
148                 retval = 1;
149             }
150         }
151         //update X pointer after each recv
152         xP = (volatile _SPM short *)*(_SPM unsigned int *)
153             *(audioP->x_pnt+0);
154         //after receiving, add all signals into recvChanel[0]
155         int shift_am = *audioP->recv_am - 1;
156         for(int i=1; i<(*audioP->recv_am); i++) {
157             volatile _SPM short * xnxtP =
158                 (volatile _SPM short *)*(_SPM unsigned int *)
159                 *(audioP->x_pnt+i);
160             for(int j=0; j<(*audioP->xb_size); j++) {
161                 xP[2*j] = (xP[2*j]>>shift_am) +
162                     (xnxtP[2*j]>>shift_am);
163                 xP[2*j+1] = (xP[2*j+1]>>shift_am) +
164                     (xnxtP[2*j+1]>>shift_am);
165             }

```

```

166     }
167 }
168 //REPEAT Ns TIMES:
169 for(unsigned int j=0;j<*audioP->Ns; j++) {
170     //PROCESS Nf TIMES
171     offs = 2 * j * (*audioP->yb_size);
172     switch(*audioP->fx) {
173     case DRY:
174         for(unsigned int i=0; i < *audioP->Nf; i++) {
175             ind = 2 * i * (*audioP->s);
176             audio_dry(&xP[offs+ind], &yP[ind]);
177         }
178         break;
179     case DELAY: ;
180         _SPM struct IIRdelay *delP = (_SPM struct IIRdelay *)
181             *audioP->fx_pnt;
182         for(unsigned int i=0; i < *audioP->Nf; i++) {
183             ind = 2 * i * (*audioP->s);
184             audio_delay(delP, &xP[offs+ind], &yP[ind]);
185         }
186         break;
187         /*
188          MORE EFFECTS HERE: NOT SHOWN
189         */
190     default:
191         if(get_cpuid() == 0) {
192             printf("effect not implemented yet\n");
193         }
194         break;
195     }
196     //ACK: ONLY ONCE AT THE END
197     if(j == (*audioP->Ns - 1)) {
198         if(*audioP->in_con == NOC) {
199             //acknowledge to all recv channels
200             for(int i=0; i<*audioP->recv_am; i++) {
201                 if(mp_ack((qpd_t *)*audioP->recvChanP+i),
202                     TIMEOUT) == 0) {
203                     retval = 1;
204                 }
205             }
206         }
207     }
208     //SEND ONCE
209     if(*audioP->out_con == NOC) {
210         //before sending, copy send data to all send channel
211         buffers
212         for(int i=1; i<(*audioP->send_am); i++) {
213             volatile _SPM short * ynxtP =
214                 (volatile _SPM short *)*(_SPM unsigned int *)
215                 *(audioP->y_pnt+i);
216             for(int j=0; j<(*audioP->yb_size); j++) {
217                 ynxtP[2*j] = yP[2*j];
218                 ynxtP[2*j+1] = yP[2*j+1];
219             }
220         }

```

```

220         //send to all send channels
221         for(int i=0; i<*audioP->send_am; i++) {
222             if(mp_send((qpd_t *)*(audioP->sendChanP+i),
223                 TIMEOUT) == 0) {
224                 retval = 1;
225             }
226         }
227         //update Y pointer after each send
228         yP = (volatile _SPM short *)*(_SPM unsigned int *)
229             *(audioP->y_pnt+0);
230     }
231 }
232 break;
233 case XLY:
234     //REPEAT Nr TIMES:
235     for(unsigned int j=0; j<*audioP->Nr; j++) {
236         //RECEIVE ONCE
237         if(*audioP->in_con == NOC) {
238             //receive from all recv channels
239             for(int i=0; i<*audioP->recv_am; i++) {
240                 if(mp_recv((qpd_t *)*(audioP->recvChanP+i),
241                     TIMEOUT) == 0) {
242                     retval = 1;
243                 }
244             }
245             //update X pointer after each recv
246             xP = (volatile _SPM short *)*(_SPM unsigned int *)
247                 *(audioP->x_pnt+0);
248             //after receiving, add all signals into recvChanel[0]
249             int shift_am = *audioP->recv_am - 1;
250             for(int i=1; i<(*audioP->recv_am); i++) {
251                 volatile _SPM short * xnxtP =
252                     (volatile _SPM short *)*(_SPM unsigned int *)
253                         *(audioP->x_pnt+i);
254                 for(int j=0; j<(*audioP->xb_size); j++) {
255                     xP[2*j] = (xP[2*j]>>shift_am) +
256                         (xnxtP[2*j]>>shift_am);
257                     xP[2*j+1] = (xP[2*j+1]>>shift_am) +
258                         (xnxtP[2*j+1]>>shift_am);
259                 }
260             }
261         }
262         //PROCESS Nf TIMES
263         offs = 2 * j * (*audioP->xb_size);
264         switch(*audioP->fx) {
265             case DRY:
266                 for(unsigned int i=0; i < *audioP->Nf; i++) {
267                     ind = 2 * i * (*audioP->s);
268                     audio_dry(&xP[ind], &yP[offs+ind]);
269                 }
270                 break;
271             case DELAY: ;
272                 _SPM struct IIRdelay *delP = (_SPM struct IIRdelay *)
273                     *audioP->fx_pnt;
274                 for(unsigned int i=0; i < *audioP->Nf; i++) {

```



```

275         ind = 2 * i * (*audioP->s);
276         audio_delay(delP, &xP[ind], &yP[offs+ind]);
277     }
278     break;
279     /*
280     MORE EFFECTS HERE: NOT SHOWN
281     */
282     default:
283         if(get_cpuid() == 0) {
284             printf("effect not implemented yet\n");
285         }
286         break;
287     }
288     //ACK ONCE
289     if(*audioP->in_con == NOC) {
290         //acknowledge to all recv channels
291         for(int i=0; i<*audioP->recv_am; i++) {
292             if(mp_ack((qpd_t *)*(audioP->recvChanP+i),
293                 TIMEOUT) == 0) {
294                 retval = 1;
295             }
296         }
297     }
298 }
299 //SEND ONCE
300 if(*audioP->out_con == NOC) {
301     //before sending, copy send data to all send channel
302     buffers
303     for(int i=1; i<(*audioP->send_am); i++) {
304         volatile _SPM short * ynxtP =
305             (volatile _SPM short *)*( _SPM unsigned int *)
306             *(audioP->y_pnt+i);
307         for(int j=0; j<(*audioP->yb_size); j++) {
308             ynxtP[2*j] = yP[2*j];
309             ynxtP[2*j+1] = yP[2*j+1];
310         }
311     }
312     //send to all send channels
313     for(int i=0; i<(*audioP->send_am); i++) {
314         if(mp_send((qpd_t *)*(audioP->sendChanP+i),
315             TIMEOUT) == 0) {
316             retval = 1;
317         }
318     }
319     break;
320 }
321
322 return retval;
323 }

```

C.2 audioint.h

```

1  #ifndef _AUDIOINIT_H_
2  #define _AUDIOINIT_H_
3
4
5  //input/output buffer sizes
6  const unsigned int BUFFER_SIZE = 128;
7  //amount of configuration modes
8  const int MODES = 2;
9  //how many cores take part in the audio system (from all modes)
10 const int AUDIO_CORES = 4;
11 //how many effects are on each mode in total
12 const int FX_AMOUNT[MODES] = {5, 5, };
13 //maximum amount of effects per core
14 const int MAX_FX_PER_CORE[AUDIO_CORES] = {2, 1, 2, 1, };
15 //maximum FX_AMOUNT
16 const int MAX_FX = 5;
17 // FX_ID | CORE | FX_TYPE | XB_SIZE | YB_SIZE | S | IN_TYPE |
   OUT_TYPE //
18 const int FX_SCHED_0[5][8] = {
19     { 0, 0, 0, 16, 16, 1, 0, 2 },
20     { 1, 1, 4, 16, 16, 1, 2, 2 },
21     { 2, 2, 2, 16, 16, 1, 2, 2 },
22     { 3, 3, 6, 16, 16, 1, 2, 2 },
23     { 4, 0, 0, 16, 16, 1, 2, 1 },
24 };
25 const int FX_SCHED_1[5][8] = {
26     { 0, 0, 0, 64, 64, 1, 0, 2 },
27     { 1, 1, 3, 64, 64, 1, 2, 2 },
28     { 2, 2, 11, 64, 64, 1, 2, 3 },
29     { 3, 2, 7, 64, 64, 1, 3, 2 },
30     { 4, 0, 0, 64, 64, 1, 2, 1 },
31 };
32 //pointer to schedules
33 const int *FX_SCHED_P[MODES] = {
34     (const int *)FX_SCHED_0,
35     (const int *)FX_SCHED_1,
36 };
37 //amount of NoC channels (NoC or same core) on all modes
38 const int CHAN_AMOUNT = 9;
39 //amount of buffers on each NoC channel ID
40 const int CHAN_BUF_AMOUNT[CHAN_AMOUNT] = { 3, 3, 3, 3, 3, 3, 3, 1,
   3, };
41 // column: FX_ID source , row: CHAN_ID dest
42 const int SEND_ARRAY_0[5][CHAN_AMOUNT] = {
43     {1, 0, 0, 0, 0, 0, 0, 0, 0, },
44     {0, 1, 1, 0, 0, 0, 0, 0, 0, },
45     {0, 0, 0, 1, 0, 0, 0, 0, 0, },
46     {0, 0, 0, 0, 1, 0, 0, 0, 0, },
47     {0, 0, 0, 0, 0, 0, 0, 0, 0, },
48 };
49 const int SEND_ARRAY_1[5][CHAN_AMOUNT] = {
50     {0, 0, 0, 0, 0, 1, 0, 0, 0, },

```

```

51     {0, 0, 0, 0, 0, 0, 1, 0, 0, },
52     {0, 0, 0, 0, 0, 0, 0, 1, 0, },
53     {0, 0, 0, 0, 0, 0, 0, 0, 1, },
54     {0, 0, 0, 0, 0, 0, 0, 0, 0, },
55 };
56 //pointer to send arrays
57 const int *SEND_ARRAY_P[MODES] = {
58     (const int *)SEND_ARRAY_0,
59     (const int *)SEND_ARRAY_1,
60 };
61 // column: FX_ID dest    ,   row: CHAN_ID source
62 const int RECV_ARRAY_0[5][CHAN_AMOUNT] = {
63     {0, 0, 0, 0, 0, 0, 0, 0, 0, },
64     {1, 0, 0, 0, 0, 0, 0, 0, 0, },
65     {0, 1, 0, 0, 0, 0, 0, 0, 0, },
66     {0, 0, 1, 0, 0, 0, 0, 0, 0, },
67     {0, 0, 0, 1, 1, 0, 0, 0, 0, },
68 };
69 const int RECV_ARRAY_1[5][CHAN_AMOUNT] = {
70     {0, 0, 0, 0, 0, 0, 0, 0, 0, },
71     {0, 0, 0, 0, 0, 1, 0, 0, 0, },
72     {0, 0, 0, 0, 0, 0, 1, 0, 0, },
73     {0, 0, 0, 0, 0, 0, 0, 1, 0, },
74     {0, 0, 0, 0, 0, 0, 0, 0, 1, },
75 };
76 //pointer to receive arrays
77 const int *RECV_ARRAY_P[MODES] = {
78     (const int *)RECV_ARRAY_0,
79     (const int *)RECV_ARRAY_1,
80 };
81
82 #endif /* _AUDIOINIT_H_ */

```

C.3 NoC Schedule XML file

```

1  <?xml version='1.0' encoding='UTF-8'?>
2  <nocsched xmlns:xi="http://www.w3.org/2001/XInclude" version="0.1">
3  <description>NoC TDM scheduling</description>
4  <platform height="2" width="2">
5  <topology linkDepth="0" routerDepth="3" routerType="sync"
6  topoType="bitorus"/>
7  </platform>
8  <application>
9  <configurations>
10 <communication comType="custom" phits="3">
11 <channel bandwidth="4" from="(0,0)" to="(1,0)"/>
12 <channel bandwidth="1" from="(1,0)" to="(0,0)"/>
13 <channel bandwidth="4" from="(1,0)" to="(0,1)"/>
14 <channel bandwidth="1" from="(0,1)" to="(1,0)"/>
15 <channel bandwidth="4" from="(1,0)" to="(1,1)"/>
16 <channel bandwidth="1" from="(1,1)" to="(1,0)"/>

```

```
16     <channel bandwidth="4" from="(0,1)" to="(0,0)"/>
17     <channel bandwidth="1" from="(0,0)" to="(0,1)"/>
18     <channel bandwidth="4" from="(1,1)" to="(0,0)"/>
19     <channel bandwidth="1" from="(0,0)" to="(1,1)"/>
20 </communication>
21 <communication comType="custom" phits="3">
22     <channel bandwidth="4" from="(0,0)" to="(1,0)"/>
23     <channel bandwidth="1" from="(1,0)" to="(0,0)"/>
24     <channel bandwidth="4" from="(1,0)" to="(0,1)"/>
25     <channel bandwidth="1" from="(0,1)" to="(1,0)"/>
26     <channel bandwidth="4" from="(0,1)" to="(0,0)"/>
27     <channel bandwidth="1" from="(0,0)" to="(0,1)"/>
28 </communication>
29 </configurations>
30 </application>
31 </nocsched>
```