Diss. ETH No. 11936

Concept and Design of a Reconfigurable Parallel Processing System for Digital Audio

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by

Matthias Felix Rosenthal
Dipl. El.-Ing. ETH
born October 14, 1968
citizen of Diepflingen, BL

accepted on the recommendation of

Prof. Dr. A. Gunzinger, examiner
Prof. Dr. L. Thiele and Prof. Dr. E. Rathe, co-examiners

1997

Abstract

On the one hand, the signal processing of digital audio mixing consoles requires a complex parallel computer architecture. On the other hand, producing custom specific mixing consoles demands a high configuration flexibility. Combining these two topics, a new concept for an automatic configuration of digital audio mixing consoles on parallel computer architectures is presented. The goal is to configure a cost-effective and individually adapted system using a simple functional definition. System and production costs are substantially reduced compared to other concepts. The mixing console is defined in an audio processing graph. Distribution of function modules, communication among processors, synchronization of the graph and integration of realtime parameters build the main issues of mapping the graph onto the parallel architecture. A configuration software performs this mapping automatically. The implementation on the parallel computer MUSIC is described and allows the verification of the concept. Hardware and software are discussed in detail.

Zusammenfassung

Einerseits setzt die Signalverarbeitung innerhalb digitaler Audiomischpulte eine komplexe, parallele Rechnerarchitektur voraus. Andererseits verlangt die Produktion von kundenspezifischen Mischpulten eine hohe Konfigurationsflexibilität. Um diese beiden Aspekte zu vereinbaren wird ein neues Konzept zur automatischen Konfiguration von digitalen Audiomischpulten auf parallelen Rechnerarchitekturen vorgestellt. Ziel ist es, mittels einer einfachen, funktionalen Definition ein kostengünstiges und spezifisch angepasstes System zu konfigurieren. Im Vergleich zu anderen Konzepten können auf diese Weise System- und Produktionskosten wesentlich reduziert werden. Das Mischpult wird in einem Graphen definiert. Verteilung von Funktionsmodulen, Kommunikation unter den Prozessoren, Synchronisation des Graphen und Integration von Echtzeitparametern bilden die Hauptbestandteile für die Abbildung des Graphen auf die parallele Architektur. Eine Konfigurationssoftware führt eine automatische Abbildung durch. Die Implementation auf dem Parallelrechner MUSIC wird beschrieben und dient zur Verifikation des Konzeptes. Hardware und Software werden detailliert vorgestellt.

Preface

At the beginning of the MUSIC project, it was a challenge to build a compact parallel computer, having a performance comparable to costly and huge supercomputers. To achieve such a goal, the main processor needs to be fast. Back in 1990, a digital signal processor was used as main processor. This processor was chosen because of its remarkable performance rather than for signal processing. The first applications were training of neural networks, molecular dynamics, plasma physics, and parallel computer graphics. When I joined the group, the idea came up to use the MUSIC for a classical signal processing application: Digital audio processing. Processing a few digital audio channels is possible on a personal computer. Processing several tens of channels, however, requires a powerful parallel computer. First tests showed satisfying results. But soon an additional problem appeared: Changing the functionality of the processing inherited a new allocation of tasks to processors by hand and a new, complicated routing of signals through the system. This procedure was time consuming and difficult. Discussions with partners in the industry showed that the problem for configuring a mixing console is not new. More and more, high demands for a fast and efficient reconfiguration are required. Before, audio mixing consoles only worked in the analog domain and were configured by inserting the appropriate analog circuit boards. With the transfer of audio to the digital domain, a reconfiguration was equivalent with reprogramming a parallel processor system including its interprocessor network. Doing this task by hand is not tolerable, because each system is custom specific and has to be adapted.

It was likely to develop a concept for an automatic configuration of a digital mixing console on a parallel processor system. This thesis presents a possible solution.

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. A. Gunzinger for his confidence in me and my work, and his assistance during my years at the institute. Thanks to my associate advisors Prof. L. Thiele and Prof. E. Rathe for co-examing and the numerous valuable comments on this thesis.

Special thanks to all the members of the MUSIC group without whom the synergy, necessary to come up with this work, would not have exist. The corporation with Hansruedi Vonder Mühll and Peter Kohler in hardware designs, and with Bernhard Bäumle, René Hüsler, and Martin Frey in software development was always excellent and with great pleasure. The fruitful discussions during coffee-breaks and other occasions together with members of the lab, including Rolf Sommerhalder, Eduard Hiltebrand, and Roger Morel, were always helpful.

Thanks to the numerous students for their contributions during their graduate work.

Finally, I want to thank my wife and best friend, Anella. Without her help and encouragement this work would not have been possible.

Matthias Rosenthal

Contents

Chapter 1

Introduction

1.1. Motivation

In the recent years audio signals are being handled more and more in the digital domain. *Compact disc* (CD) and *digital audio tape* (DAT) formats are now widely accepted as being a very reliable and high quality medium for reproducing music. With these systems it is possible to store digital audio, the digitally sampled version of analog audio signals, in an efficient and inexpensive way. There are many advantages for using digital audio. As an example, the signal quality remains constant and the dynamic range can be enlarged using more bit per value. Other problems like quantization noise and the demand for high processor capacities are introduced. However, the improvements suppress the doubts of the still large community which prefers analog audio. There exists a great interest to minimize analog audio as much as possible. Intentions are to digitize an audio signal directly behind an acquisition device (microphone) and to convert it back to the analog domain only for reproduction. Recording and mixing of audio signals are complicated and costly processes, particularly when done in the digital domain. High signal processing demands exist for filtering and mixing. For instance, a digital mixing console, which concurrently handles more than 100 digital audio signals, uses several Gops[1] of processing power sustained. Such large scale audio processing systems require the realization of highly parallel architectures. The number of required processors is in the range of 100. Apart from the hardware development, the programming of parallel systems is a complicated topic. A large number of processors work on one big task, each is executing a single subtask. Such concurrent programs are inherently more complex than sequential

[1] 10^9 Operations per Second

programs, running on only one processor. Additionally, when a parallel system is used for digital processing and mixing, high demands in terms of flexibility are asked. The end-user, rather than the designer of a high-end digital mixing console, defines the functionality, the amount of channels, special features. As a result, the design of custom specific mixing consoles is a key feature that needs to be satisfied.

1.2. Digital Audio Processing

When designing a system for processing digital audio both, hardware and software aspects have to be taken into account. They are touched in the following areas:

- Flexibility: The hardware and software architecture need to be open for a fast reprogramming to allow custom specific designs.
- Cost: The amount of hardware resources defines the price of the system. For a parallel architecture it is primarily the number of processors used, which must be held low. The system cost depends on their optimal usage and on the efficiency of the software.
- Interactivity: Audio processing is a real-time application which demands interactive modifications of sound by the user. The aim is to find ways to integrate real-time controlling without loss of signal quality.

The subject of this thesis is to integrate these three issues into a parallel processor system. A wide range of parallel systems have already been built for general purpose applications such as, video processing, training of neural networks, simulations of chemical reactions, or weather prediction. Therefore, many problems which arise in parallel systems are known and are subject of technical research. It is interesting to adapt these problems for digital audio processing.

1.2.1. Parallel Computer Architectures

Parallel computers can execute applications faster than a single-processor system. The aim is to let many processors work on the same problem. Ideally, if an application is divided into p subtasks, each is

running on a separate processor, it finishes p times faster. However, communication, synchronization and granularity of the subtasks prevent reaching such an optimal speedup. The feasible performance depends on the application and the system architecture. General purpose parallel computers try to offer the best possible speedup for a large field of applications. Flynn divides the various computer architectures into four groups [Fly66, Fly72]:

- *Single Instruction stream - Single Data stream* (SISD) represents the single-processor system.
- *Single Instruction stream - Multiple Data stream* (SIMD) are parallel computers which run the same instructions on all processors simultaneously. Each processor works on a different data stream.
- *Multiple Instruction stream - Multiple Data stream* (MIMD) are parallel computers running different codes on each processor concurrently. MIMD computers are again subdivided into:
  1. *Single Program stream - Multiple Data stream* (SPMD) execute the same program (not the same instruction) on all processing elements.
  2. *Multiple Program stream - Multiple Data stream* (MPMD) run different codes and process different data on all processing elements.
- *Multiple Instruction stream - Single Data stream* (MISD) use a single data stream which flows through a linear array of processors executing different instruction streams. MISDs are special purpose machines.

Digital audio processing – in professional audio applications – is performed using a MIMD model. Each processor in the system has its own program. The multiple data stream, which is communicated among the processors, consists of a number of digitally sampled audio channels. This model is also MPMD, since each processor runs its own, distinct program.

Nevertheless, consumer systems are often built using dedicated hardware in a SIMD model. Several processors perform the same instructions on multiple audio streams. These systems, however, lack the possibility of open configurations.

1.2.2. Parallel Programming

Regardless of the architecture of the target parallel computer, a parallel application must coordinate two or more program segments. There are two fundamental programming models, which integrate such a coordination: *control-flow* and *data-parallel* [And91]. In the control-flow model, the execution of the program is dictated by a control path and does not depend on the availability of data. The synchronization of intermediate results, which are used by other program segments, has to be guaranteed solely by the program. For instance, if a code segment A wants to send a message to a code segment B, A must be sure that B is expecting its message. On the other hand, B has to rely on A that it will send the message, otherwise it waits forever. If such a synchronization does not work correctly, a *deadlock* occurs.

The data-parallel model requires the division of the data set into separate partitions. Each partition is processed in parallel. The program on each processor is identical in its functionality, but different data partitions are processed. The program passes three phases: First, data is partitioned and distributed, second, parallel processing is performed and third, results are collected into a new, single partition. In this model, deadlocks can not occur, if each processor contributes its part of data.

Digital audio arrives synchronously in given intervals. The exact time of data availability is known in advance. Using the periodicity of digital audio, synchronization of program segments can be done following the data-parallel model: Program segments start their execution with the availability of new data and have to complete before a new set of data arrives. A task may not take a longer processing time than the time between two audio samples. The end of this time is called *dead-line*. All distributed tasks are distinct. They hardly alter during operation. Filter coefficients or parameters may vary according to real-time manipulations.

1.2.3. Dedicated and General Purpose Signal Processing

The field of digital audio mixing consoles is divided into two areas. The low-end market looks for inexpensive, preconfigured systems. The price is an important factor in the decision process of the user. The high-end market, however, demands individuality. Almost every user asks for another special feature for his mixing console which is absolutely necessary and has to be realized without reduction in the functionality. Such demands of users have to be taken into account at the very beginning of a new development. As a result, configurability is required for the high-end market of digital audio mixing consoles while the low-end market needs preconfigured, cost optimized systems.

Since digital audio processing is a dedicated application, the hardware architecture of the system can be specialized to run just this task. Maximal performance can be reached this way. But it involves a reduction in terms of flexibility. As a result, processing digital audio with a parallel system can be done either with dedicated hardware or by using general purpose digital signal processors (DSP). There are many discussions what has to be used in which field of the audio world. The most obvious differences are: First, using dedicated hardware is faster, using DSPs offer more flexibility. Second, the design of integrated circuits (IC), as it is necessary for dedicated hardware, demands high initial costs. It is a reasonable solution if the corresponding market exists and hundreds of systems can be sold. Consequently, dedicated hardware is advisable for low-end systems, general purpose signal processors for high-end architectures.

1.2.4. Load-Balancing and Optimal Assignment

Parallelism is used out of two possible motivations:

1. The time until an application finishes can be reduced by exploiting parallelism. In general, the more processors are applied, the shorter it is.

2. Some applications have to be completed within a given time. If a single processor can not provide the required performance to compute the application within the given time, one needs as many processors as necessary to finish before the given dead-line.

General purpose computers are built out of the first reason. The application stops running when the last subtask in the system has been completed. The principle of the *load-balancing* is to balance the subtasks on the processors in order to let each processor do approximately the same work. It is an optimization procedure where the wait-time for the processor with the biggest task is minimized. This way, the entire application will finish in the least possible time. The more processors we use, the faster the application finishes.

The second reason is important for digital audio. The time between two audio sampling clocks defines the maximal computing time. It is a constant processor capacity in terms of processing cycles that can be performed within the given time. The principle of the *optimal assignment* is to use the processing capacity of the processors as well as possible. The goal is to finish the task before the given dead-line and with a small number of processors. As a result, a parallel system uses only a minimal number of processors and the total system cost is optimized.

1.2.5. Granularity

The *granularity* or *grain size* is used as a measure to determine the size of the basic subtask chosen for parallel processing. Often three types of granularity are distinguished: fine grain, medium grain and coarse grain [Hwa91]. *Fine grain* makes use of parallelism at instruction or loop level. The grain size is typically small and does not exceed 500 instructions. *Medium grain* executes subroutines or small subprograms in parallel. *Coarse grain* includes the parallel execution of big subprograms and independent programs.

Handling digital audio is a fine grain application. Within a period of two audio samples, between 500 and 1000 instructions can be executed. The small number of possible instructions requires an optimal implementation of the application. As a result, digital signal processors are programmed directly in machine language since compilers are not able to exploit the ability of DSPs to achieve three and more operations in one processing cycle. The architecture of a DSP is dedicated to implement signal processing algorithms straightforward. The lower bound for the grain size is at instruction or algorithm level which is,

for example, a digital filter or a fast Fourier transformation (FFT).

1.2.6. Communication

A large variety of processor interconnection techniques exist. From the hardware point of view architectures like global access, ring and multi-stage networks are common. Message-passing and shared memory build the software interface to the hardware [Hwa91].

Parallel audio processing is performed by synchronously transmitting audio samples through the parallel system. The bandwidth of the interprocessor network defines the possible amount of communicated data and the number of audio channels, which can be used within the system. For example to communicate 500 audio channels a transmission bandwidth of 25MWord/s is necessary. Considering a system with over 50 nodes it is a critical issue to construct a network which supplies this bandwidth. For the mixing of digital audio, broadcast capabilities of the network are required. The time division multiplexed (TDM) bus is the most frequently used communication architecture in digital audio. Within the period of two audio samples a defined number of channels is transmitted on a globally accessible bus. Ring networks and hierarchical bus structures are possible alternatives.

1.3. Outline of the Work

Chapter 2 gives an overview for digital audio mixing. The audio mixing console is introduced. The functionality, the market and differences between digital and analog audio mixing consoles are illustrated.

Chapter 3 discusses the configuration concept. Based on the idea of mapping a signal-flow graph on a parallel processing system, three problems are examined: Module assignment, processor communication and synchronization. Derived from heuristics of the bin-packing problem (BPP) a system prediction is presented.

Chapter 4 analyzes the digital signal processing within a mixing console. Modifications of the audio signal in the frequency-, dynamic- and time-domain are described. A special section observes the real-time controlling of processing modules.

Chapter 5 goes into the realization of the mixing-engine on the MU-

SIC parallel processing system. The hardware and the software architecture is presented. All aspects of the configuration concept, described in chapter 3, are included.

Chapter 6 shows the configuration results of the mixing-engine on MUSIC. The expected performance is given based on the implemented function library.

Chapter 7 concludes the achieved work and gives an outlook on further activities.

Chapter 2

Digital Audio Processing

In this chapter the architecture and usage of audio mixing consoles are discussed. A first section explains the principles of a mixing console. It is followed by a section describing the differences between analog and digital consoles. After a short summary of the worldwide market, the processing and communication requirements are examined. A section about digital audio data exchange concludes the chapter.

2.1. What is an Audio Mixing Console?

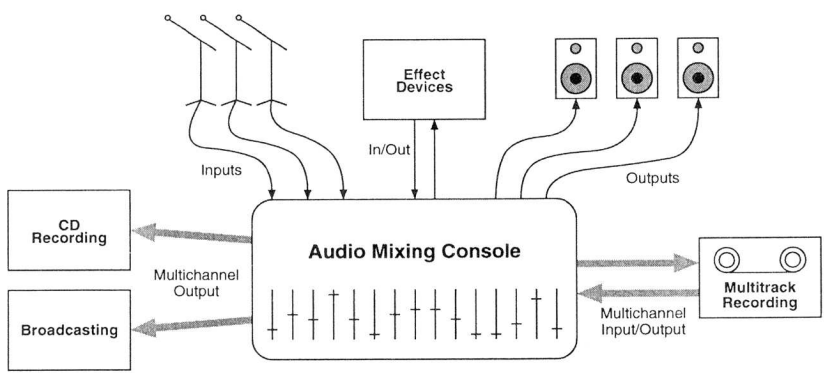


Figure 2.1: The audio mixing console typically builds the heart of an audio recording studio.

As shown in Fig. 2.1 the audio mixing console is typically the center in the recording studio. Audio acquisition and reproduction devices (microphones, loudspeakers, recording devices, etc.) are connected through one, two, or multichannel links. Within the mixing console, audio signals are modified in real-time using manipulation devices on the control-

desk (rulers, knobs, etc.). Modifications are performed in the frequency domain, the dynamic domain, and the time domain of an audio signal. The main purpose, the mixing itself, allows the routing of incoming audio channels onto any outgoing audio channel.

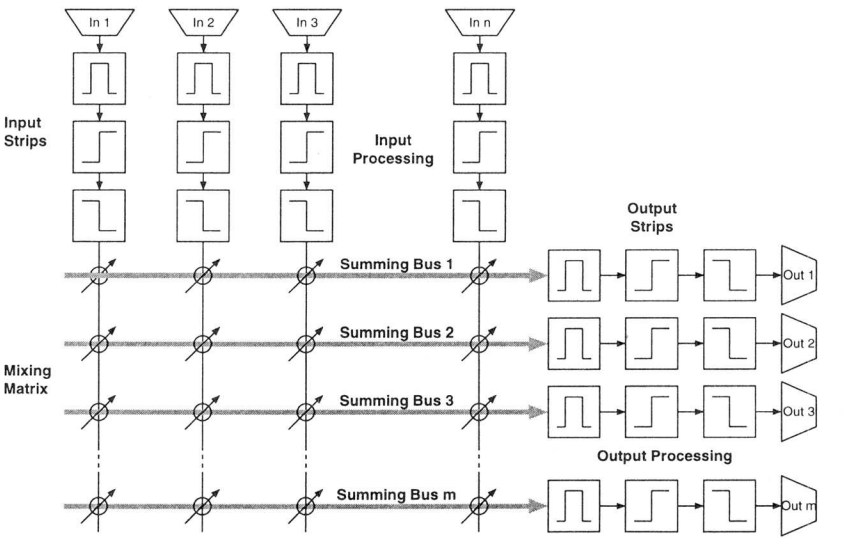


Figure 2.2: The basic functionality of an audio mixing console: The input and output processing parts are conjoined by a summation matrix, where each incoming signal is added by an adjustable factor.

This basic signal-flow in an audio mixing console is shown in Fig. 2.2. In principle, it is a $N \times M$ mixing matrix where N incoming audio signals are "mixed" to M outgoing audio signals. Incoming audio signals first enter an input processing section. They can be modified by a set of linear and nonlinear filters. One such processing path is called a *strip*. Each strip is represented on the control desk by rulers and knobs arranged in vertical columns. Thus, the amount of processing paths in a mixing console is directly evaluated by counting the strips on the control desk[1]. The mixing matrix, in the center of the signal-flow, allows

[1]This practice is used for analog consoles. New consoles have a layered control-

the individual accumulation of every incoming audio signal onto M different summing busses. Each summing bus produces a new audio signal out of a collection of input signals. Although different types of summing busses exist (main bus, auxiliary bus, etc.) their implementation is the same. There are up 48 summing busses implemented in a large mixing console. The factor, by which a specific signal is multiplied to such a summing bus, can be controlled by a ruler in the strip. M output signals go into an output processing section. Still, each output signal can be modified before it reaches the output connectors of the console. A common practice is to use some of the output signals again as input signals. The signal-flow of Fig. 2.2 is extended and intermediate stages are generated. Fig. 2.3 shows how the signal-flow is expanded. The size

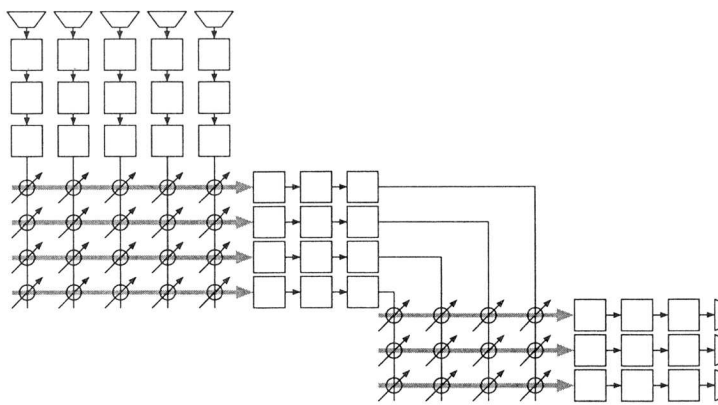


Figure 2.3: The extended model of a mixing console.

of the intermediate stages is not defined and may vary. When operating with an analog mixing console, bypass cables are connected at the rear of the console. Digital mixing consoles allow such routing procedures in software.

The functionality of a strip is not precisely defined and differs from implementation to implementation and within the console. In general, five main function blocks are distinguished. Fig. 2.4 depicts the possible components in a strip. The processing modules are, in some implemen-

desk where the number of strips on the desk is smaller than the number of physically

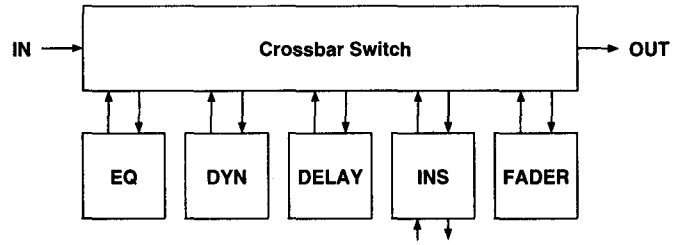


Figure 2.4: A strip consists of different function modules. Its sequence can be altered during runtime with a crossbar module.

tations, connected to a crossbar switch. This allows to change the sequential order of the modules in real-time. The *equalizer module* (EQ) alters the signal in the frequency domain. Parts of the spectral representation is increased or decreased, respectively, using high-pass, low-pass and band-pass filters. The dynamic range of the signal is modified in the *dynamic module* (DYN). Using a control system, the loudness of the audio signal is measured and, according to a given characteristic, the signal is manipulated with a certain gain factor. The *delay module* inserts a variable delay into the audio signal. External devices are inserted into an audio signal using the *insert device* (INS). The last module builds the *main fader*. All modules are regulated in real-time using control devices on the control desk.

As mentioned above, a strip is configured according to individual needs. Within one console a limited amount of different strips is implemented. For example, parts of the input processing section are equipped with a special equalizer or some output strips have two faders implemented.

2.2. Digital versus Analog

Audio mixing systems can be divided into two generic classes: analog and digital. The distinction between these two classes lies in the representation of the audio information. An analog system processes a continuous stream, whereas a digital system handles the discretely

implemented strips.

sampled and quantized representation. The design of an *analog mixing console* (AMC) is done directly according to the given basic signal-flow of Fig. 2.2. The sequential processing parts of the input and output sections are implemented in vertical strips. Analog filters and function modules are implemented on printed circuit boards directly under each strip. The summation matrix is realized by a number of horizontal summing tracks. They go from one end to the other end of the console and are accessible over the total width of the system. As a result, all audio signals are propagated through the entire console. It inherits a difficult analog circuit design on all boards. Noise and crosstalk can be reduced, but hardly eliminated. The amount of summing busses is fix and given by the number of summing tracks. There is no possibility of reconfiguring the mixing matrix without exchanging the hard-wired summing tracks. But, reconfiguring a single strip is equivalent with swapping the analog strip board and can be easily performed.

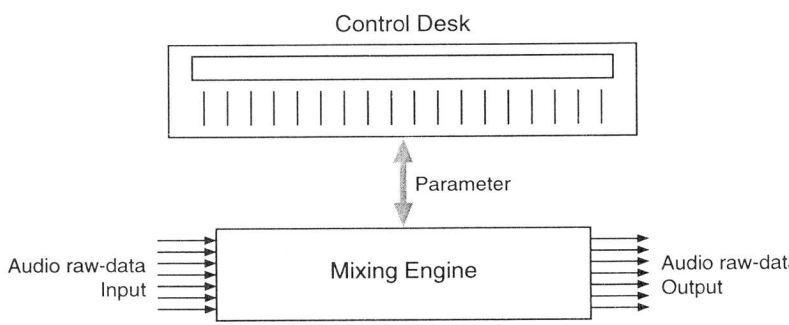


Figure 2.5: Architecture of a Digital Mixing Console

If we look at a *digital mixing console* (DMC) (Fig. 2.5), the most obvious difference to the analog counterpart is the clear division of control-desk and mixing-engine. To provide compatibility, the control-desk is designed similar to an AMC. The basic functionality is equal to those of an AMC. The clear and distinct separation opens the opportunity to divide the design and the implementation of a DMC into two parts. The mixing-engine consists of a parallel signal processing architecture with a sophisticated interprocessor communication network. All signal processing is performed in real-time in this central unit. Audio

data arrives in digital form via single or multichannel interfaces. The audio sampling rate defines the processing time for the entire system. The control-desk represents the interface to the user and provides the complete supervision over the mixing-engine. Each processing function, running on the mixing-engine, is directly controlled by a set of parameters. These are communicated between the two hardware parts through a separate communication interface. The update rate of parameters is not that critical and can be done much slower than the audio sampling rate. But, the new parameter values need to be interpolated within the mixing-engine to reach a smooth transition for each audio value.

This thesis concentrates on the concept and implementation of the mixing-engine. The design of the corresponding control-desk is subject to ergonomical and industrial aspects and will not be discussed further.

2.3. Sampling Frequency

The sampling frequency of the digitized audio signal is not defined precisely. Several factors have influence for its selection. Foremost is the requirement that the sample frequency f_s is at least two times the highest signal frequency of interest. In practical applications this ratio should be about 2.5 to avoid fold-over (aliasing) in the frequency domain. Low sampling frequencies are desirable to lower data transmission bandwidth in processing systems. High frequencies are necessary for high fidelity. Based on the fact that the human perception of sound is in the spectrum below 20kHz the sampling frequencies are all below 50kHz. Several standards are defined and used in commercially available systems. The Audio Engineering Society (AES) has recommended 48kHz for professional recording studio applications and 32kHz for broadcast and transmission related applications such as digital audio broadcasting (DAB) and digital satellite radio (DSR) [McN84b]. Consumer products, like compact disk (CD) and digital audio tape (DAT), sample at 44.1kHz. 44.056kHz is used for mixed video and audio data (VCR) in order to synchronize the audio samples with video framing formats.

A digital mixing console is a strongly synchronous system. Digital audio studios provide a master-clock, which defines the audio sampling rate for the complete studio. Therefore, the mixing console works at

this clock rate. If a device with a differing sampling clock is connected to the mixing console, the digital audio data must be converted to the master-clock. This is done using sampling-rate converters (SRC) at the digital interfaces. Even if a device with the same sampling rate, but with its own master-clock has to be connected, a SRC is required. This case is known as a plesiochronous sampling-rate conversion [Rot95].

The processing time within the system varies with the sampling frequency and according to the field where the system is used. The shortest processing time appears in professional audio applications where 48kHz are used.

2.4. Market

The market for audio mixing consoles can be divided into three major segments (low-end, medium, high-end). This division also corresponds to the console sizes. The worldwide turnover for audio mixing consoles is about 400Mio US$ per year. As shown in Tab. 2.1[2], the market vol-

System complexity	low-end	medium	high-end
Number of strips	6-24	24-96	96-192
Price [thousand US$]	1-10	10-100	100-1000
Worldwide turnover [MIO US$]	140	130	120
Part of digital consoles [%]	30	20	20
Number of Processors (when digital)	1-5	5-25	25-100
Demand for configurability	low	low	high

Table 2.1: The market of audio mixing consoles is divided into three areas: low-end, medium, and high-end. The presented data is based on estimations of the market for 1995.

ume in each segment is approximately equal. Most of the sold mixing consoles still work analog and digital consoles make less than 30%. The trend is clearly toward the digital domain. Small and medium sized mixing consoles are designed for a large low-price market. A configuration is done once for a specific type of console, which then is sold thousands

[2]Data collected from personal conversations with associates in the industry.

of times. Changing a configuration is equivalent to constructing another type of console.

Other requirements exist in the high-end market: In this section the user defines the architecture of the mixing console. Not only type and amount of strips are to be defined, but also the inner functionality, like filter types, insert points, additional modules, etc. The producer of a large console must follow these demands. Based on this fact, a great interest for a rapid mixing console configuration exists [LTP91].

2.5. System Requirements

Fig. 2.2 on page 10 reveals two topics for the implementation of a mixing-engine: Processing and communication. The functionality of the strips mainly require processing power. The summing or mixing matrix, which combines the processing parts, demands high communication support. The sampling frequency is an additional parameter which has influence on the required hardware resources. The following calculations are based on a sampling frequency of 48kHz, which corresponds to a maximal processing time of $20.83\mu s$. This is the highest sampling rate for audio and sets the minimal processing time.

2.5.1. Processing

The size of an audio mixing console depends upon the number and the complexity of the implemented strips. To evaluate the needed processing performance in detail, we examine the implementation of one strip. Tab. 2.2 demonstrates an example of a mono audio strip. The amount of processing cycles is given for each subfunction. Parameter interpolation is included, which is necessary to eliminate perceptible distortion.

Beside the functionality of the strip, high processing demands lie in the implementation of the summing matrix. Consider a large digital mixing console with 192 strips and 48 summing busses. The worst case arises when every strip needs to be connected to each summing bus. If we calculate two processing cycles for each connection point in the matrix, which is a rather optimistic estimation, already 900Mips[3] are

[3]Million Instructions per Second

Function	Amount	Cycles	Total Cycles
IIR filter 2nd ord.	4	18	72
Dynamic range	1	43	43
Fader	4	13	52
Panorama	2	17	34
Delay	1	14	14
Insert	1	8	8
Total			223

Table 2.2: Typical processing requirements for a mono audio path given in processor cycles on a Motorola DSP96002.

required at 48kHz. Fortunately, this case never appears, because of a grouping of summing busses in the mixing console: Each strip can only be connected to defined groups of busses. A processing estimation for the summing matrix can hardly be made, since it depends on the final design and the group size of summing busses. Large and complex consoles use up to 25% of processing cycles for summation. A total of around 300 cycles per strip results, when looking at Tab. 2.2.

Currently available digital signal processors (DSP) are running at 40MHz. The DSP, which was used in this thesis, is a Motorola DSP-96002. It needs two processor clocks per instruction cycle and performs around 400 cycles between two audio sampling clocks. More recent DSPs are able to execute a processing cycle in one processor clock and therefore perform up to 800 cycles at 40MHz processor speed. About two complete strips can be implemented on the Motorola processor while up to 3 or 4 strips can be handled by improved processors. Since the size of a large mixing console varies from 100 to 200 strips, the number of used processors is between 25 and 100. Note that this calculation does not include any processing overhead for communication among processors, synchronization and data input/output.

2.5.2. Interprocessor Communication

The second topic of a mixing-engine is the data exchange between processing elements. Most communication support is demanded for the mixing matrix. In general, every processor needs a certain amount of

audio channels as input data and produces a number of new channels, that need to be communicated to other processors. The amount of transmitted data depends on the complexity of the configuration. The worst case arises when all data produced on all processing elements has to be communicated to all other processing elements. This case is called *all-to-all broadcasting*. Several requirements for the network have to be taken into account:

1. High bandwidth: Assume that at least one signal per strip goes into the interprocessor network. For a system with 200 strips, a minimal bandwidth of 10MWord/s is required at a sampling frequency of 48kHz.

2. Concurrent communication and processing: Data communication should work as independently as possible to relieve the processors.

3. Scalability: For a given configuration only the necessary number of processors should be installed to optimize the total system cost. It makes a scalable network necessary.

4. Short latency: The time when a signal enters the mixing console until it reaches the output connectors should be as short as possible.

The communication network of digital mixing consoles is often designed directly in hardware, because a software controlled communication does not satisfy the strict requirements of speed or uses too much processing power. As a result, the communication network becomes very inflexible and changes in the digital data-flow of audio signals are difficult.

2.6. Digital Audio Connections

Digital audio exchange with external devices is provided by two mainly used connection standards. Both are defined by the Audio Engineering Society (AES) and the European Broadcasting Union (EBU).

First, the AES/EBU digital interface is a two channel digital interface [AES92]. During one audio sample interval, one frame of digital

information is transported. It is divided into two subframes which contain the left and right audio channel of a stereo signal. Each subframe consists of 32 bit of information. 24 bit are used for the raw audio signal. 4 bit serve as a preamble. Apart from the information about left or right channel, the receiver can reconstruct the sampling frequency from the preamble. This way an extra link for the sampling clock can be omitted. The last 4 bit of a subframe contain specific data, called *VUCP*: A validity bit *V*, a user bit *U*, a channel-status-bit *C* and a parity bit *P*. The user bit can be applied for custom specific information. 192 frames together build one block. Within one block, 24 Bytes of status information is serially transmitted using the channel-status-bit. The sampling rate, channel mode (stereo, monaural, etc.), copy permission and other specific data is encoded. The consumer format, named Sony/Philips digital interface (SPDIF), is a slightly modified version of the AES/EBU digital interface. Only the definition of the channel-status-bit and the mechanical specifications of the connectors differ. While AES/EBU uses the large symmetric XLR-Connector SPDIF uses Cinch[4] or optical links. The data rate is 3.072 Mbit/s at 48kHz sampling rate.

The second digital interface is the multichannel audio digital interface (MADI) [AES91]. MADI is used especially in professional audio environments and allows the communication of up to 56 digital audio channels through a coaxial or fiber-optic link. A minimal data rate of 86.016 Mbit/s is required. The standard demands to calculate with a tolerance of 12.5%. A maximal data rate of 96.768 Mbit/s results. Hardware components, used for the fiber distributed digital interface (FDDI)[5] [FDD92] standard, are applied to support this bandwidth. They are called TAXI [AMD92] and include a 4/5-coding for error correction. The MADI format is derived from the AES/EBU format. One MADI frame consists of 56 AES/EBU subframes. Each of these mono channels includes 32 bit: 24 bit of audio, 4 bit preamble and 4 bit VUCP.

[4]Connector type invented by CINCH Connectors Inc., Elk Grove Village, Illinois
[5]FDDI allows a bandwidth of 100 Mbit/s.

2.7. Synchronization

The mixing console has to provide a unique system latency. The time when a signal enters the mixing console until it reaches the output connectors can vary according to the different processing times for each signal. The different latencies have to be synchronized to a single system latency. Signals having a shorter latency have to be stored and held in memory until they are synchronous with the signal having the longest latency. Thus, the data values of all outgoing signals correspond to one sampling time. A system latency of several milliseconds for recording is accepted. A certain part of this time is used for analog to digital and digital to analog conversion.

Chapter 3

Concept

What are the requirements for an adaptable and efficient configuration of a mixing console on a parallel processor system? The basic signal-flow of the console, as described in chapter 2, needs to be generalized. A new mixing model is therefore introduced. The functionality of a mixing-engine is characterized in a graph. The issue lies in mapping the graph onto a parallel processor system. It leads to three major problems: Module assignment, communication and synchronization. They are examined in detail below.

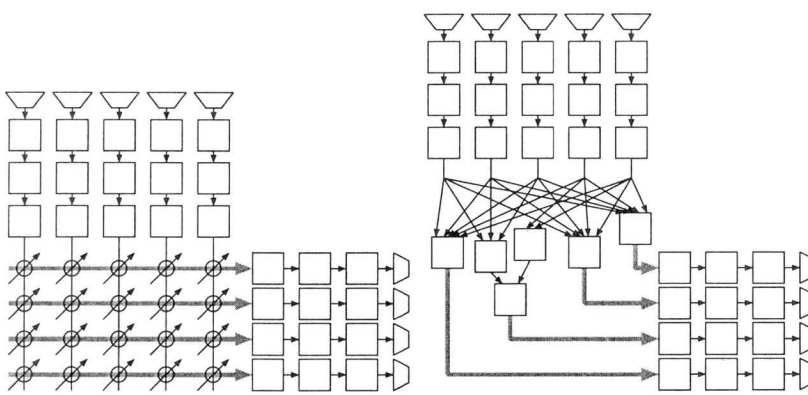
3.1. Configurability

A digital mixing-engine can be designed and constructed directly by programming each processor "by hand". A major problem arises, when each sold mixing console differs from another. It must be possible to configure the mixing console in a very short time. The desk can be configured by inserting the corresponding strip modules. The mixing-engine, however, demands an automatic configuration procedure. The program of each processor in the core has to be produced by a configurator software with an integrated code generator. Several papers propose concepts for the automatic configuration of an audio mixing-engine [LTP91, LGTP91a, LGTP90]. But, their efficiency is not satisfying. Precedence and communication constraints are mixed with load-balancing of processors and do not exploit the periodicity of digital audio. Additionally, system scalability is poor [LGTP91b] or not taken into account [MK91].

A configuration concept needs a certain functional description of a mixing-engine as input data. A mixing model is therefore introduced.

3.1.1. Mixing Model

A digital mixing console can, in general, be described as a network of signal paths, where audio processing modules (e.g., functions, algorithms) are combined. The summing busses build a crucial exception. They expect inputs from each path of the processing section and cause waste communication overhead among processors. Other implementations of digital mixing consoles prefer a realization directly in hardware. A fixed amount of summing busses is available in the system. Changes in the signal-flow through these busses are hardly possible. A configuration is confined to exchange the functionality of processing sections. The implementation style with separate summing busses and processing sections still resembles the architecture of analog mixing consoles. But, the realization of summing busses can be efficiently done as a processor task. Fig. 3.1 shows the transformation into a general description. Summing busses are integrated in processing modules. The advantage here lies in the flexibility of the new description. It is most similar to a directed acyclic graph (DAG).



(a) Standard Model (b) New Model

Figure 3.1: The new model uses only audio connections and processing modules. Summing busses are realized using processing modules.

Definition: DAG *A DAG is a directed acyclic graph $G(V, E)$ with a set of nodes V and a set of edges E. Each node $v_i \in V$ represents an object (e.g., task, instruction, program) and each edge $e = (v_i, v_j) \in E$ represents a data dependency.*

The representation of a DAG is similar to the intermediary form used by parallel compilers like the SR Programming Language [AO93] or SISAL [SG85, ODFB91]. SISAL (Streams and Iterations in a Single-Assignment Language), for example, is a high level functional data-flow language. It generates IF1 (Intermediate Form 1) which is a language based on DAGs. The idea is to split the program into small pieces with a finer granularity. Based on a high level programming language, parallel compilers describe the program flow at instruction level. Digital signal processing is extremely time critical. High level programming languages, like C or Modula, lose a factor three to four in performance compared to an implementation in machine code on a DSP. The optimized programming style in machine code is therefore still reasonable for digital signal processing. How can the efficiency of optimized code and the flexibility of a DAG be combined? One solution is increasing the granularity of processing modules. Their size should be as big as the implementation of a certain functionality requires. On the other hand, when different combinations of modules are allocated to a processor, the resulting processor load has to be high. If the modules are too large, a good processor usage is not guaranteed.

Additionally, there is an important difference between a DAG and the model shown in Fig. 3.1(b). To build the summing matrix the output of a specific node may feed several summing nodes. The new description is called a directed acyclic audio graph (DAAG), shown in Fig. 3.2.

Definition: Audio Channel *Given a set of nodes V. An audio channel $c = (v_s, v_a, v_b, \ldots, v_n)$ is a digital audio connection between a source node $v_s \in V$ and a set of sink nodes $A \in V$. Data is transmitted from v_s to $(v_a, v_b, \ldots, v_n) \in A$ through this channel periodically to the master-clock.*

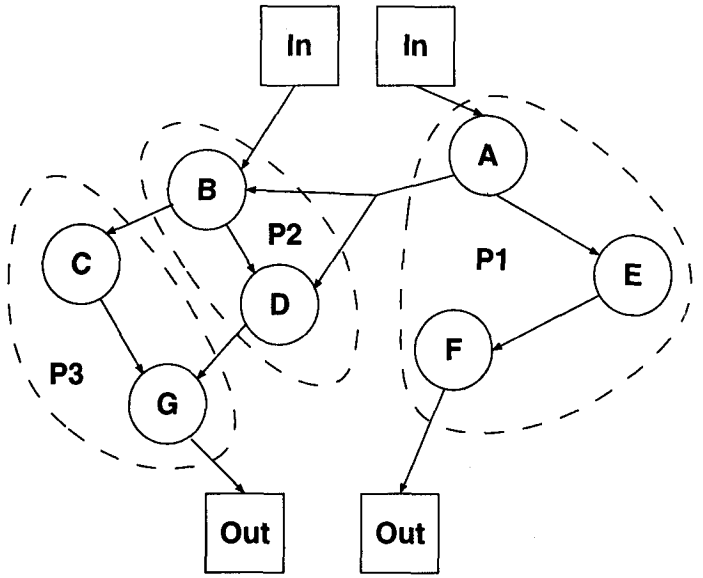


Figure 3.2: The description model for the configuration of a digital mixing-engine is a DAAG. The main topic lies in finding an efficient solution of grouping processing modules $A, B, .., G$ onto processors $P_1, P_2, .., P_n$ considering processor capacity, communication and synchronization.

Definition: Processing Module *A processing module is a sequence of processor instructions. After starting a module on a processor, it can not be interrupted until the last instruction has finished. The time for executing a module on a processor is constant .*

Definition: DAAG *A DAAG is a directed acyclic audio graph $G(V, C)$ with a set of nodes V and a set of audio channels C. Each node $v_i \in V$ represents a processing module. Each channel $c = (v_s, v_a, v_b, \ldots, v_n) \in C$ represents an audio channel.*

A complex signal processing task is broken down into subtasks. Each node stands for a processing module, which is implemented in optimized machine language. Every node receives data from a preceding

node and sends at least one audio channel to one or several following nodes. Therefore, every module has at least one incoming and one outgoing connection. The data-flow principle is that every module can perform its computation whenever input data is available on its incoming connections. As the program execution is controlled by the availability of data, the entire system is said to be *data-driven*. The squares indicate hardware input and output devices, they are marked with 'In' and 'Out', respectively. These are the only nodes that are dedicated sources and sinks of the DAAG. In this model no computation is performed by these devices. A DAAG has the following properties:

1. The graph has a defined number of inputs and outputs.
2. Any connection between modules is allowed.
3. All modules have at least one input and one output.
4. Loops are not allowed.
5. New data arrives at the input nodes periodically with the master-clock

Problems arise in mapping this graph to the parallel processor architecture. If the capacity of the processors can be optimally used, the cost of the resulting system is adapted for each graph individually.

3.1.2. Static and Dynamic Configuration

A primary condition for an optimal assignment is its predictability. It is important to have all information about computation times, resource requirements, precedence relationships and communication requirements of each module in advance. A module has a certain deadline by which it must be completed. In general, all modules used in a digital mixing-engine are periodic tasks and the deadline complies with the period of the given module. Consequently, only one execution time for each module exists on a uniform processor system.

It is possible to perform a feasibility analysis to predict whether a configuration is possible or not. The feasibility depends upon the processor capacity and the communication bandwidth of the interprocessor

network. This analysis produces a schedule according to which tasks are dispatched onto processors. We distinguish two major approaches for a configuration of a digital mixing-engine:

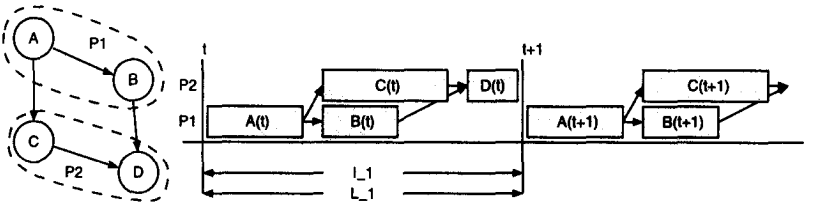
- *Static table-driven configuration:* A static schedule (e.g., table, sequence) is generated for each processor according to a feasibility analysis. It is then used at runtime.
- *Dynamic planning-based configuration:* Unlike the previous approach, the feasibility is checked at runtime. Tasks arrive dynamically and are accepted only if there are enough processing and communication resources for their execution.

A *static table-driven configuration* is applicable to systems performing periodic processing. This approach is extremely predictable since start and completion times of each module are known *a priori* to the runtime schedule. Another advantage is a minimal runtime software overhead on each processor because modules are executed according to a fix schedule. However, any change of a module and its characteristic requires a complete remake of a configuration and a new schedule has to be generated for the entire system.

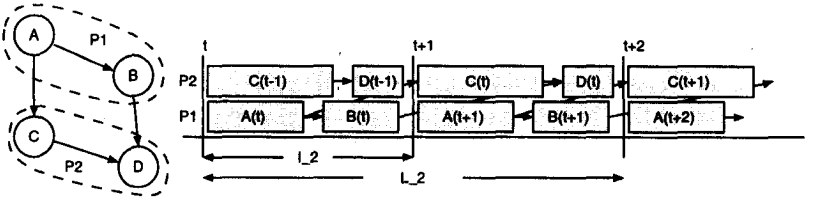
The *dynamic planning-based configuration* provides the flexibility of dynamically allocating modules during runtime. A runtime schedule on each processor exists and is periodically executed. If a new module needs to be assigned, an attempt is made to create a new schedule that contains the previous tasks as well as the new ones. If the attempt was successful, the new schedule can be started. If it fails, the system takes alternative steps. The feasibility checking and the generation of a new schedule has to be made within the timing constraints of the system. Thus, in a dynamic configured system, the overhead of dynamically allocating modules is enormous and the utilization of the processors is rather low. The advantages in terms of flexibility during runtime result in a performance that is far from optimal.

3.1.3. Pipelining

Since a DAAG is periodically executed according to a static schedule, it can be pipelined through a parallel processor system. This way, shorter intervals are applicable. Additionally, precedence relationships between processing modules running on different processors do not have an influence to the load of the processor. Two important criterions are used to define the efficiency of pipelining. It is the latency L and the interval time I.



(a) No Pipelining



(b) Pipelining

Figure 3.3: The mapping of a simple DAAG onto two processors with and without pipelining.

Definition: Latency *The latency L of a DAAG represents the required time for processing the entire graph once on a parallel system.*

Definition: Interval Time *The interval time I of a periodically processed graph is the time between two periods of the graph.*

Definition: Pipelining *A pipelining of a DAAG on a parallel system is called pipelining, if the relation between the resulting latency and the interval time fulfills $I < L$.*

Fig. 3.3 shows the mapping of a simple DAAG onto two processors with and without pipelining. The assignment of modules to processors is $(A, B) \in P_1$ and $(C, D) \in P_2$ is the same in both cases. The interval time of one master-clock I_1 and the latency L_1 is the same when no pipelining is performed (Fig. 3.3(a)). Processor P_2 has to wait for input data from P_1. With pipelining a shorter interval time I_2 is reached (Fig. 3.3(b)). The processor P_2 executes its processing modules without considering precedence constraints during one master-clock.

The interval time for digital audio described in a DAAG is equivalent with the master-clock. When pipelining is applied, no wait time for preceding processing modules exists. The full capacity of each processor can be utilized during one master-clock.

3.1.4. Orthogonality

Suppose a static schedule of a DAAG is generated for a parallel processor architecture. The architecture consists of a number of processors and an interprocessor communication network. If a module needs to communicate with another module running on a different processor, the interprocessor network has to establish a communication channel. If the two modules are situated on neighboring processors, a direct communication (nearest neighbor communication) between these processors is possible. But, the distance of a processor-to-processor communication depends on the complexity of the graph. The worst case arises when every module has a connection with a module situated on the farest processor in the network.

Another topic for the communication network is the implementation of a mixing matrix, where summing busses are combined. It needs an extremely high communication rate over the entire system. Processing modules have to collect and add a large amount of audio channels. One

module is responsible for a complete summing bus, as shown in Fig. 3.1. Assuming that the processing modules, which are connected with one or several summing busses, broadcast their data globally in the system, the location, on which processor a summing module is running, can be neglected.

For a fully routable system, that allows all interconnections between two or more modules, each processor must be able to have access to all data from all other processors. The architecture, that fulfills this requirement is a *broadcast network*.

For a certain configuration it is necessary to bind only the needed amount of processors in a system. This way system costs are individually adapted and optimized for each configuration. *Scalability* of the interconnection network is a condition to provide such optimizations.

In the case of a mixing-engine the interconnection network must support both features, broadcast capabilities and scalability. Such a network is called an *orthogonal* network and has the following properties:

- The interprocessor network is a broadcast network
- The amount of processors is scalable
- The number of globally communicated audio channels does not depend on the number of processors

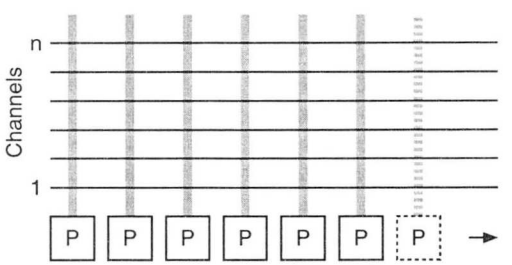


Figure 3.4: On an orthogonal system, the amount of processors and the number of communicated channels are independent. Scalability is provided by a selectable number of processor.

The last topic implies that the communication bandwidth does not depend on the scalability. Fig. 3.4 depicts the orthogonality of communicated audio channels and processing elements. The amount of communicated channels is given by the network. It remains constant under the assumption that the network performs the best possible communication throughput. Therefore, the highest possible communication bandwidth is provided even if only 2 processing elements are combined. The number of PEs, however, is scalable to a maximum bounded by electrical or mechanical characteristics.

3.2. Graph Mapping

Mapping a DAAG onto a parallel architecture can be performed in respect to one or several constraints:

1. Minimal number of processors
2. Minimal communication cost
3. Minimal system latency

In general, these subjects are closely dependent on each other. For example, the number of processors, the communication bandwidth and the system latency vary depending on the location of processing modules.

In the following considerations it is assumed that the parallel processor architecture is given and is orthogonal. The cost of such a system is minimized with the minimal number of processors. Modules can be executed on any processor. Therefore the sum of computation, that is required by all modules, gives the lower bound of the needed processors. This bound can hardly be reached because of the granularity of the different processing modules. Nevertheless, the minimal number of processors for a given DAAG is the primary requirement on an orthogonal system. The final price and the acceptance of the mixing console is dominated by this constraint.

Considering the system latency builds a secondary problem in the assignment process and can be neglected under the following condition:

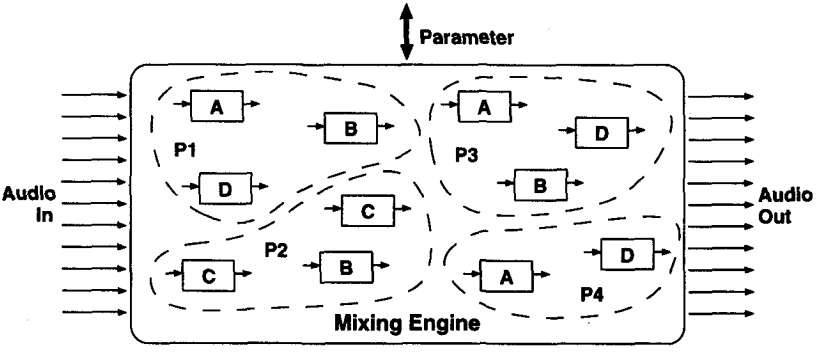


Figure 3.5: A configuration is performed by distributing groups of modules onto processors. The optimal assignment is equivalent the minimum amount of hardware resources needed for a certain configuration.

Neglecting System Latency: A digital mixing-engine can have a latency in the range of 1ms[1]. This time corresponds to almost 50 audio samples. It means, an audio signal can be pipelined through 50 processors until it reaches the output connectors of the system. Looking at the signal flow of a mixing-engine (Fig. 2.2 on page 10) the number of processing modules involved in one audio signal will not exceed 20. In most implementations, minimizing the system latency can therefore be neglected.

As a result, it is possible to concentrate on the assignment of processing modules without considering system latency. But, it has to be guaranteed that the time limitation, as mentioned above, is not exceeded. If an assignment results in a higher latency than the given limitations, an optimization has to be performed.

Two different configuration approaches result, where the number of processors and communication costs are involved:

1. **Graph Partitioning:** Consider partitioning and communication costs together

[1]Analog to digital and digital to analog conversion requires around 2ms. An additional latency of 1ms is tolerable for processing.

2. **Assign and Optimize:** Assign processing modules ignoring communication costs and optimize communication costs only when necessary

The first approach tries to partition the graph into equally sized subsets taking all communication costs into consideration. The second approach is a combination, which uses one algorithm to assign modules on processors and another algorithm to optimize communication costs if necessary. Both approaches are discussed in the following sections. In this work, the second approach was applied.

3.2.1. Graph Partitioning

The goal of the graph partitioning problem is to partition the vertices of a graph in k roughly equal parts, such that the number of edges connecting vertices in different parts is minimized. Formally, the k-way graph partitioning problem is defined as follows: Given a graph $G = (V, C)$, create a partition V into k equally sized subsets $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = 0$ for $i \neq j$, and $\cup_i V_i = V$. The number of edges of C whose incident vertices belong to different subsets has to be minimized. A k-way partition of V is commonly represented by a vector P of length n, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and k, indicating the partition to which vertex v belongs. Given a partition P, the number of edges whose incident vertices belong to different subsets is called the *edge-cut* of the partition.

The implementation of many parallel algorithms need an optimization of both, communication and processor costs. A possibility is the solution of the graph partitioning problem, where vertices represent computational tasks, and edges represent data exchanges. A k-way partition of the computation graph can be used to assign tasks to k processors, and because it minimizes the edge-cut, the communication costs are also minimized. Therefore, finding a partition with minimal communication costs is equivalent with computing a minimum edge-cut bisection P of a given DAAG $G(V, C)$.

The k-way partition problem is most frequently solved by recursive bisection. It means, we first obtain a 2-way partition of V, and then we further subdivide each part using 2-way partitions. After $log\ k$ phases, graph G is partitioned into k parts. Thus, the problem of performing

a k-way partition is reduced to that of performing a sequence of 2-way partitions or bisections. This scheme does not necessarily lead to the optimal partition, but it is used extensively due to its simplicity.

A partitioning P can be obtained using various algorithms such as (a) spectral bisection [PSL90, PSWB92, BS93], (b) geometric bisection [MTTV93], and (c) combinatorial methods [KL70, Bar84, GL81]. Four popular algorithms are described in the next sections. The first uses spectral bisection, the other three algorithms make use of combinatorial methods. Geometric bisection techniques are not described since these methods are applicable only if coordinates are available for the vertices of the graph.

Spectral Bisection (SB) In this algorithm, the spectral information is used to partition the graph [PSL90, BS93, HL93]. This method is very expensive since it requires the computation of the eigenvector corresponding to the second smallest eigenvalue (Fiedler vector). The execution of the SB algorithm can be speeded up if the computation of the Fiedler vector is done using a multilevel algorithm [BS93]. However, even such multilevel spectral bisection (MSB) algorithms usually take a large amount of computation time.

Kernighan-Lin Algorithm (KL) The Kernighan-Lin algorithm [KL70] is of iterative nature. It starts with an initial partition of the graph. Each iteration tries to find a subset of vertices, from each part of the graph, such that swapping them leads to a partition with smaller edge-cut and hence with lower communication cost. If such subsets exist, then the swap is performed and this becomes the partition for the next iteration. The algorithm continues by repeating this process. If it cannot find two such subsets, then the algorithm terminates, since the partition is at a local minima and no further improvement can be made by the KL algorithm. Each iteration of the KL algorithm, as described in [KL70], takes $O(|E| \log |E|)$ time. Several improvements to the original KL algorithm have been developed. One such algorithm is found by Fiduccia and Mattheyses [FM82] that reduces the complexity to $O(|E|)$, by using appropriate data structures.

The Kernighan-Lin algorithm has been found to be effective in finding locally optimal partitions when it starts with a fairly good initial partition and when the average degree of the graph is large [BJ93]. If no good initial partition is known, the KL algorithm is repeated with different randomly selected initial partitions, and the one that yields the smaller edge-cut is selected. Requiring multiple runs can be expensive, especially if the graph is large.

Formally, the KL algorithm is applied as follows. Suppose P is the initial partition of the vertices of the graph $G(V, C)$. The *gain* g_v of one vertex v is defined as the reduction in the edge-cut if the vertex v moves from one partition to the other. This gain is generally defined by

$$g_v = \sum_{(v,u)\in C \wedge P[v]\neq P[u]} w(v,u) - \sum_{(v,u)\in C \wedge P[v]=P[u]} w(v,u) \qquad (3.1)$$

where $w(v,u)$ is the weight of edge (v,u). If g_v is positive, then by moving v to the other partition the edge-cut decreases by g_v. If g_v is negative, the edge-cut increases by the same amount. If a vertex v is moved from one partition to the other, then the gains of the vertices adjacent to v may change. Thus, after moving a vertex, we need to update the gains of its adjacent vertices.

Given this definition of gain, the KL algorithm proceeds by repeatedly selecting from the larger part a vertex v with the largest gain and moves it to the other part. After moving v, v is marked so it will not be considered again in the same iteration, and the gains adjacent to v are updated to reflect the change in the partition. The KL algorithm, described in [FM82], continues moving vertices between the partitions, until all the vertices are moved.

Graph Growing Partitioning (GGP) Another simple way of bisecting a graph is to start from a vertex and grow a region around it, until half of the vertices have been included (or half of the total vertex weight) [GL81]. The advantage of this scheme over KL is that if the graph is connected, then the GGP algorithm produces connected partitions. This may not be the case with KL, since initial partitions are selected randomly, and KL might terminate at a local minima that yields disconnected partitions.

The quality of the GGP algorithm is sensitive to the choice of the vertex from which to start growing the graph, and different starting vertices yield different edge-cuts.

Greedy Graph Growing Partitioning (GGGP) The graph growing algorithm described in the previous section grows a partition in a strict breath-first fashion. However, as in the KL algorithm, for each vertex v we can define the gain in the edge-cut obtained by inserting v into the growing region. Thus the vertex with the largest decrease (or the smallest increase) in the edge-cut is inserted first.

As in the case of the GGP algorithm, this algorithm is also sensitive to the choice of the initial vertex, however, this sensitivity is much smaller.

Multilevel Graph Bisection (MGB) Multilevel graph bisection algorithms reduce the size of the graph (i.e. coarsen the graph) by collapsing vertices and edges. Then the smaller graph is partitioned, for example with an algorithm as described above. Finally, the graph is uncoarsed to construct a partition for the original graph. Such multilevel schemes were primarily investigated to decrease the partition time. Recently, a number of multilevel algorithms have been proposed [HL93, BJ93, KK95] that further refine the partition during the uncoarsening phase. Good results are obtainable this way.

3.2.2. Assign and Optimize

This approach is a combination of two algorithms, one for assigning processing modules and, if the system cannot provide the required communication bandwidth, another algorithm for optimizing communication costs. The division into two algorithms works especially well for mixing engines on orthogonal systems because

- the interprocessor communication bandwidth is constant and is sufficient for most mixing engine configurations
- if the network is not able to support all communication requirements of a specific graph, an optimization can easily be performed

Such an optimization is done by clustering:

Clustering When a mixing-engine is described with a DAAG, then the structure of the graph will be equivalent to the basic signal-flow of a mixing-engine shown in Fig. 2.2. Processing-paths, which represent strips on the desk, have minimal communication with the rest of the system and can be grouped together. A larger module can be formed out of this group and inserted in the graph. Then, the new graph has less connections, since the communication can be performed locally. Additionally, local communication within one module inherits a smaller latency. As a result, clustering reduces global communication bandwidth and latency. But, the size of the modules increases and leads to a less optimal usage of the processors.

A complete configuration concept for a mixing-engine can be implemented in a configuration program. The following steps are performed by the program:

1. **Module assignment** The modules of the DAAG are assigned as well as possible to a minimal number of processors. Communication costs and system latency are neglected.

2. **Communication** A communication through the interprocessor network is established for every connection in the graph, where modules run on different processors.

3. If the bandwidth of the interprocessor network can not satisfy all communication requirements, small modules with low communication are grouped to larger modules and local connections are applied within these modules. A new graph is built with the new modules and a restart at step 1 is executed. The procedure continues with step 4.

4. **Synchronization** To provide a unique system latency, a synchronization scheme is generated for modules, which have more than 1 input connection.

With this approach three distinct problems, module assignment, communication, and synchronization, are isolated in steps 1, 2 and 4, respectively. An analysis of these topics follows.

3.3. Module Assignment

Since all communication costs among modules are neglected in this section, the assignment problem for a parallel system is reduced to a multiprocessor assignment problem. These problems are known to be NP-Complete.

3.3.1. NP-Complete Problems

In order to discuss configuration problems it is important to look at a special part of problems in computer science: The *NP-Complete Problems.* For our purpose, a problem can be described by a number of parameters and by an explanation of the properties of a desired solution. If we have a problem, that can be described in a mathematically precise form, it is – in most cases – possible to find at least one algorithm that is capable of finding a solution. Such a solution is not necessarily unique, only its properties need to be satisfied. We are interested to find an *efficient* algorithm for solving a problem. Efficiency can be expressed by two main factors: First, how much time does this algorithm need on a computer and second, what computing resources are required.

The size of a problem varies with the amount of input data. Thus the time for finding a solution for a specific problem grows by the size of the problem. To measure efficiency of an algorithm one can look at the *time complexity function* [GJ79]. It is the order $O(n)$ by which the calculation time grows for the given amount of input data n.

Definition: Polynomial Computing Time *If a problem can be solved in a computing time, which is proportional to the size of the problem and its time complexity function $O(n)$ is bounded by a polynomial, it is said to have a polynomial computing time.*

Let an algorithm be efficient, when the problem can be solved in polynomial computing time. All currently available computers work deterministic and there are many problems which can be efficiently solved on deterministic computers. We can classify these problems in a set P:

Definition: Deterministic Computer *A deterministic computer performs after each processing step exactly one possible action.*

Definition: P *The set of problems P represents the class of problems, for which each problem $x_i \in P$ can be solved in polynomial computing time using a deterministic computer.*

In contrast to P, there are problems, for which no known algorithm exists for its solution in polynomial computing time using a deterministic computer. If the ability of a computer is extended with nondeterminism, a polynomial computing time can be feasible for some problems. We can therefore extend P to NP:

Definition: Nondeterministic Computer *For a set of possible solutions S, a nondeterministic computer is able to choose the best solution $s_i \in S$.*

Definition: NP *The set of problems NP represents the class of problems, for which each problem $y_i \in NP$ can be solved in polynomial computing time using a nondeterministic computer.*

Obviously, $P \in NP$ and it seems that more problems belong to NP than to P. Practically, there exists no known proof that a problem belongs to NP and *not* to P. In other words, the possibility exists that $P = NP$ and all problems in NP can be solved in polynomial computing time on a deterministic computer.

A large group of problems in NP exists, for which no known efficient algorithm exists for a deterministic computer nor a proof was found, that they belong to P. This group is called NP-complete and all such problems have one important property in common: If one of these problems can be solved in polynomial compute time on a deterministic computer, this must be possible for all other NP-complete problems. It can be proven by transforming a problem $y_i \in NP\text{-}complete$ into the problem $y_j \in NP\text{-}complete$, for which a polynomial algorithm on a deterministic computer exists.

Let us take the classical *traveling salesman problem* as an example. A finite set of cities is sufficiently connected by roads. The problem is defined: Find the shortest route which stops at each city once. The size of the problem is defined by the number of cities involved. The simplest algorithm, which finds the optimal solution, is to test all possible routes. If we want to solve the traveling salesman problem with 10 cities, this would be a reasonable approach. But to test several tens of cities, it already takes years on a supercomputer.

Multiprocessor assignment problems are NP-complete problems. The input data of a multiprocessor assignment problem is in general a list $L_n = (X_1, X_2, \ldots, X_n)$ of nonnegative numbers that have to be grouped into blocks satisfying some given properties. Three major assignment problems arise: The objective of the *makespan scheduling problem* is to find a partition of L_n into m blocks so that the maximal block sum is minimized over all such partitions. A related problem is the *partition problem*, where an assignment of L_n into m is sought after such that the difference between the maximal and the minimal block sums is minimized over all blocks. Both problems have L_n and a fixed number of blocks as input data. The third example is the *bin-packing problem*. Here a partitioning of L_n into a variable number of uniform sized blocks is sought [SSNB95]. Since this problem is of major interest for the configuration of a digital mixing-engine, it is discussed in the next section.

3.3.2. Bin-Packing Problem

The bin-packing problem (BPP) is equivalent to the problem of "packing" a number of "weights" into a minimum number of "bins". It is assumed that all bins have a uniform capacity and no single weight exceeds this capacity. We note a complementary statement in the problem of assigning a number of tasks on a set of processors. The minimum number of processors needed for finishing all tasks within a given period of time is sought.

Garey and Johnson [GJ79] proved that the BPP is NP-complete. We note the non existence of an algorithm which finds the optimal solution of the problem in a polynomial computing time. However, simple heuristics already show satisfying results in a very short computing

time. The most common algorithms, which are currently used for the BPP, are:

- *Next-Fit (NF):* This is the easiest approach to the BPP. The first item of an unordered set of items is assigned to the first bin. The next item is then assigned to the same bin, when it fits. Otherwise it is allocated to a new bin. This new bin is then used until one of the next items again has to be assigned to a new bin.
- *First-Fit (FF):* Always beginning with the first bin each item is allocated to the next bin, which has enough capacity for this item. A new bin is only introduced if the current item did not fit into one of the already tested bins.
- *Best-Fit (BF):* This algorithm is obtained from FF. The current item is assigned to the bin for which the resulting capacity of that bin is minimal.
- *Next-Fit Decreasing (NFD):* In this approach the items are first organized in a decreasing order.Then the NF algorithm is applied.
- *First-Fit Decreasing (FFD):* Once again the items are sorted in a decreasing order and then the FF algorithm is applied.
- *Best-Fit Decreasing (BFD):* The BF algorithm is applied after having sorted the set of items in a decreasing order.

The first three approaches do not produce unique solutions, as the set of items is randomly organized at the startup of the algorithm. They are also called *online packing algorithms* since an a priori knowledge of the complete set of items is not given and the items possibly arrive in an unordered stream. The latter three algorithms are called *offline packing algorithms*. The criterion for an algorithm as "good" or "not good" differs for online and offline packing. For example the big (and only) advantage of the NF algorithm is its dramatically small processing effort.

Various algorithms, which are related with these fundamental BPP heuristics, exist. For example Shor [Sho91] presents a derivation of the online algorithm BF. Burchard et al. [BLOS95] introduce an offline algorithm based on NFD and FFD.

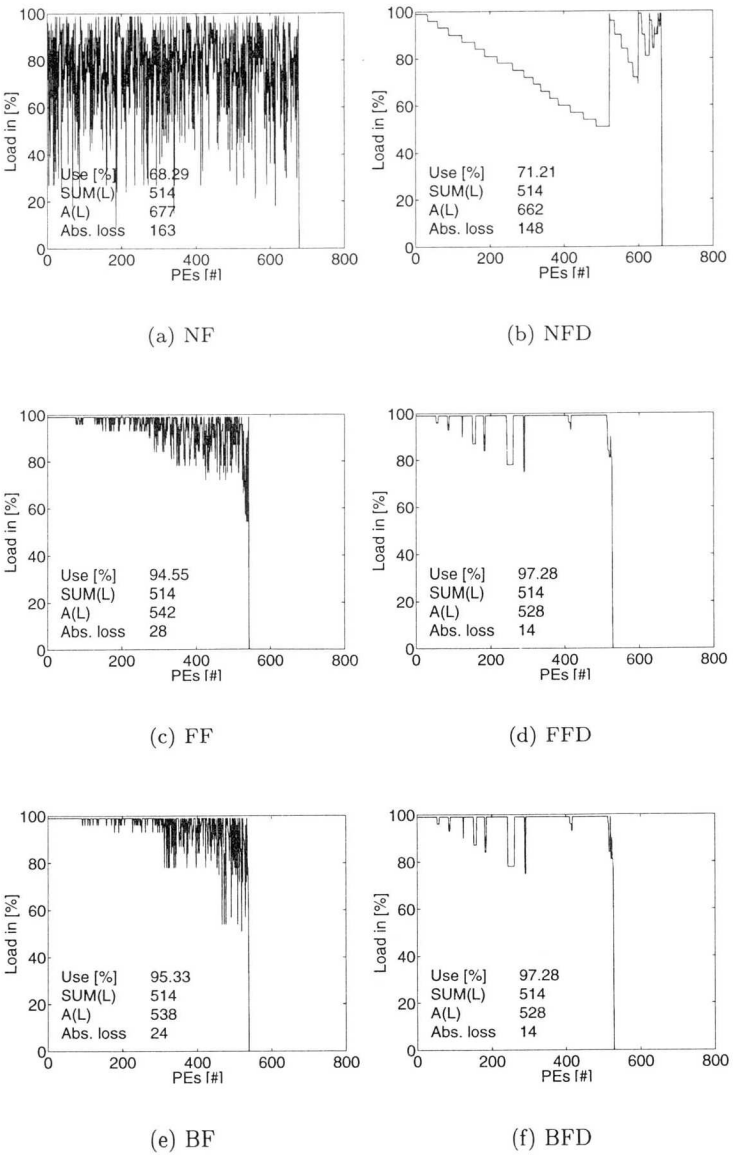


Figure 3.6: The assignment of a uniform random set L_n with $n = 1000$ using different BPP heuristics.

To compare the performance of the different heuristics we first define $OPT(L_n)$ as the optimum solution value of the BPP for a set $L_n = (m_1, m_2, \ldots, m_n)$ of processing modules. For our purpose $OPT(L_n)$ denotes the minimum number of processors. Trivially, $OPT(L_n)$ is bounded by

$$\sum_{i=1}^{n} m_i \leq OPT(L_n) \tag{3.2}$$

and is the best solution after trying every possible assignment. Without losing generality we assume the elements of L_n to be real numbers in $(0, 1]$ and the processors have a uniform capacity $c = 1$. The worst case performance ratio $r(A)$ is specified as the smallest real number for a heuristic algorithm A, with

$$\frac{A(L_n)}{OPT(L_n)} \leq r(A) \tag{3.3}$$

and $A(L_n)$ as the solution value of needed processors for the algorithm A. It can be shown, that $NF(L_n)$ satisfies the upper bound

$$NF(L_n) \leq 2\ OPT(L_n) \tag{3.4}$$

The complexity to calculate $NF(L_n)$ is simply $O(n)$, because for every element only one decision has to be made. Johnson, Demers and Ullmann [JDU+74] proved that

$$FF(L_n) = BF(L_n) \leq \frac{17}{10}\ OPT(L_n) + 2 \tag{3.5}$$

holds for all n when FF or BF is applied. In the same paper the resulting worst case bounds for FFD and BFD are proven:

$$FFD(L_n) = BFD(L_n) \leq \frac{11}{9}\ OPT(L_n) + 4 \tag{3.6}$$

The time complexity for the last four algorithms is $O(n \log n)$[2]. For arbitrarily large $OPT(L_n)$ the constant term in 3.5 and 3.6 disappears. Thus the *asymptotic worst case performance ratio* is used. It is defined

[2]It can be achieved applying a 2-3 tree data structure [AHU83]

as the minimum real number $r_\infty(A)$ such that, for some positive integer k,

$$\frac{A(L_n)}{OPT(L_n)} \leq r_\infty(A) \tag{3.7}$$

for all cases L_n and satisfying $OPT(L_n) \geq k$. Table 3.1 demonstrates the asymptotic worst case performance ratios of the discussed heuristics (from Coffman, Garey and Johnson [CGJ84]). The last three columns show $r_{u,\infty}$ for $u = \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$. They refer to instances where the items in L_n have values in $(0, u]$.

Algorithm	Complexity	r_∞	$r_{1/2,\infty}$	$r_{1/3,\infty}$	$r_{1/4,\infty}$
NF	$O(n)$	2.00	2.00	1.50	1.33
FF	$O(n \log n)$	1.70	1.50	1.33	1.25
BF	$O(n \log n)$	1.70	1.50	1.33	1.25
NFD	$O(n \log n)$	1.69	1.42	1.30	1.23
FFD	$O(n \log n)$	1.22	1.18	1.18	1.15
BFD	$O(n \log n)$	1.22	1.18	1.18	1.15

Table 3.1: The asymptotic worst case performance ratios of the classical BPP heuristics are shown. The three columns on the right refer to lists where the size of the elements are below $\frac{1}{2}, \frac{1}{3}$ and $\frac{1}{4}$.

Obviously, if the grain size of the distributed items is smaller, the utilization of bins is better. For FFD and BFD only 15% of the capacity is lost in the worst case, when the distributed items are in $(0, \frac{1}{4}]$. Let $L_{u,n}$ denote lists, where the item sizes are in $(0, u]$ and $0 < u \leq 1$. Bentley et al. [BJLM83] have reported an interesting behaviour of FFD applied on lists $L_{u,n}$. Experimentally the absolute wasted space

$$E_A(u, n) = A(L_{u,n}) - \sum_{i=1}^{n} m_i \tag{3.8}$$

was measured for heuristic algorithms A. With FFD on lists $L_{u,n}$ and $0 < u \leq \frac{1}{2}$, the amount of wasted space tends to be extremely small. Even when packing over hundred thousand items the excess of bins is mostly less than one bin! The experiments were done with up to 128'000 items in a list. Tests with different n's indicated an average

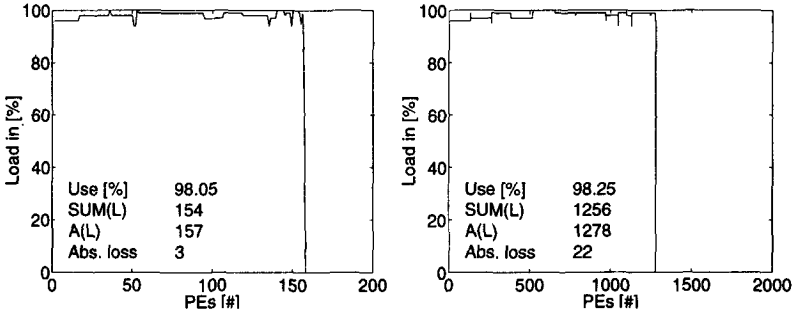
loss of 0.7 bins not depending on n. Moreover, no excess of more than 1.3 bins was ever encountered, and the excess of less than 1 bin appeared in roughly 75 percent of the time. The simulations show that the FFD algorithm is an *optimal* algorithm for 3/4 of the tested cases. In the next paper by Bentley et al. [BJL+84] a proof is given that the expected amount of wasted space is bound by a constant number independent of n. However, the constant upper bound presented there is of the order 10^{10} and is shown to hold only when $n > e^{100}$. So the difference between experimental and theoretical results is enormous. Nevertheless, the proof that there is a constant upper bound gives confidence in the experimental results. A recent paper by Floyd and Karp [FK91] investigates FFD under the assumption that the number of items has a Poisson distribution and the sizes are in $(0, \frac{1}{2}]$. With these restrictions an upper bound of 11.3 bins on the absolute excess of bins, independent of n, was proven.

3.3.3. Assignment using FFD Bin-Packing

In a mixing-engine, the granularity, the size, and the amount of the items are within a certain range. Usually only a small amount of different items exists. For example, it is possible to build a mixing engine only with an equalizer, a scale, and a summing function. They can be chosen in any combination and therefore build a library for another large list. Let $S_x[L_{u,n}]$ denote the list S_x with x items taken from the sublist $L_{u,n}$ where $n \leq x$. It is interesting to verify the results of Bentley et al. [BJLM83] for lists satisfying the following constraints:

- The used item sizes are within $0.05 \leq u \leq 0.5$.
- The number of different items is small, $5 \leq n \leq 50$.
- The number x of items is in the range $500 \leq x \leq 5000$.

These constraints result, first, from the typical module size implemented on currently available DSPs, second, from the amount of different modules currently used in mixing consoles and third, from the size of large mixing consoles that are normally produced. Tab. 2.2 on page 17 confirms the indicated ranges.



(a) $x = 500$ (b) $x = 5000$

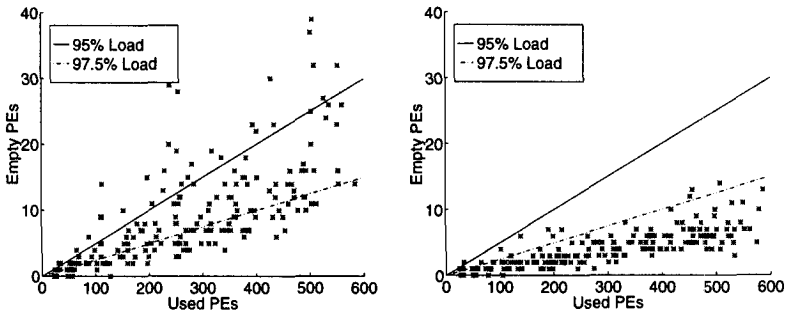
Figure 3.7: The distribution of $S_x[L_{u,n}]$ over the PEs with $n = 20$ and $0.05 \leq u \leq 0.5$. The absolute loss of PEs compared to the computational sum over all modules is measured.

Since FFD uses small items to fill gaps left by larger items it is natural to expect more wasted space when the granularity of the item sizes is rather coarse. Because the item size is at least 0.05 the lost space will increase with the number of distributed items. As an example, a library of $n = 20$ items with random size in $0.05 \leq u \leq 0.5$ is generated. Fig. 3.7 shows a FFD assignment of a list S_x over the PEs with $x = 500$ and $x = 5000$. The utilization shows good results with a average load around 98%. It seems that the excess of space grows linearly with x and the average load remains constant.

Fig. 3.8 demonstrates the growth of lost space with different n's. The lost space is qualified by the number of empty PEs when all gaps in the system are collected. Two statements can be made from the figures:

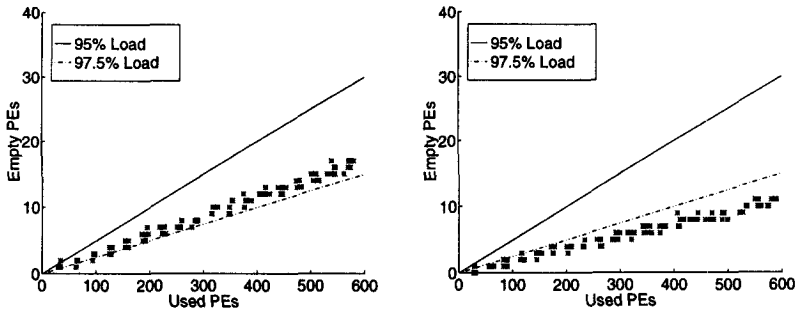
- The expected loss of PEs grows with x.
- The growth is flatter with a finer granularity.

There is no known proof for this statement. But, the simulation can give an order by which the excess of PEs grows. For this simulation, a new library $L_{u,n}$ was generated for each assignment. A rather rough



(a) $n = 10$ (b) $n = 40$

Figure 3.8: The expected loss of PEs grows with the amount of modules. But, the loss is more predictable if the granularity of the items is finer. Ten measurements are shown for each list $S_x[L_{u,n}]$ with $0.05 \leq u \leq 0.5$. The sublist $L_{u,n}$ contains 10 and 40 items with random weights, generated for each S_x.



(a) $n = 10$ (b) $n = 40$

Figure 3.9: For distributions, where $L_{u,n}$ remains constant for all S_x, the expected loss of PEs grows linear with x. Ten measurements are shown for each list $S_x[L_{u,n}]$ with $0.05 \leq u \leq 0.5$. The sublist $L_{u,n}$ contains 10 and 40 items with random weights, generated once.

scattering results and prevents to make a more precise statement. An interesting behaviour is shown in Fig. 3.9: When the sublist $L_{u,n}$ remains constant for all x, the expected loss of PEs seems to grow linearly with x and consequently linearly with the amount of PEs. It can be assumed, that with a given sublist $L_{u,n}$, the expected loss of PEs can be ascertained for any x applying simulations with random generated S_x.

The following result can be obtained for the configuration of a mixing engine: If a mixing-engine has a homogeneous, parallel processor architecture and a random configuration $S_x[L_{u,n}]$, where $L_{u,n}$ denotes a library of n modules and $0.05 \leq u \leq 0.5$, then it is possible to evaluate an expected loss of PEs for all x under the condition, that the items in S_x are picked from $L_{u,n}$ randomly. A usage factor arises which indicates the average processor load for a given library.

3.4. Communication

3.4.1. Synchronous Communication

Within each clock cycle of the master-clock a new data value arrives from every connected device. Simultaneously, a new data value must be present at all outgoing channels. Consequently the communication time for the interprocessor network is one master-clock period. This time is $20.83\mu s$ for professional audio. Arising out of this short time the interprocessor communication needs to work as independently as possible to relieve the processors.

Fig. 3.10 illustrates the overlapping of communication and processing. The communication starts with the beginning of the master-clock while the processing is synchronized with the end of the communication. To prevent data inconsistency, the processing has to take place on a different data set than the communicated set (double-buffering). Each communication going through the interprocessor network inherits a pipelining of 2 data values in an audio channel.

3.4.2. Network Topologies

There are several possible network topologies, which can be applied for a digital mixing-engine. To provide an orthogonal system the network has to supply broadcasting capabilities. Therefore, networks with

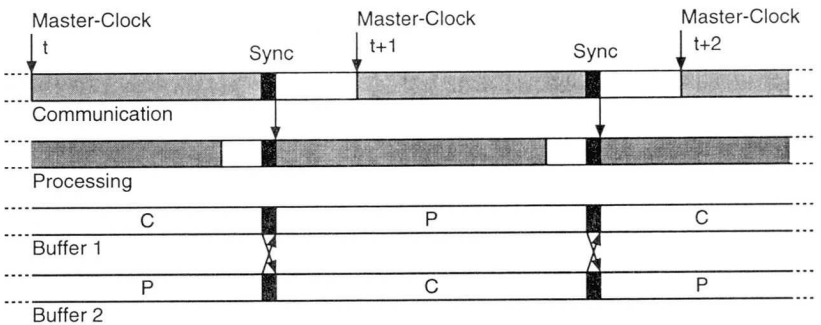


Figure 3.10: The communication starts with the master-clock. The processing begins after the comunication cycle has finished. Communication and processing use different buffers "C" and "P", which are switched during the synchronization phase.

poor or complicated broadcast support like crossbar switches, multistage networks or mesh networks are inconvenient. The most reasonable topologies are shown in Fig. 3.11. Two main distinctions are made: First, ring bus or common bus and second, a hierarchical network or a non-hierarchical one.

Common Bus. A common bus offers the best broadcasting possibilites of all network architectures. A node can produce its data directly to all other nodes. To avoid network conflicts, an arbitration must be made globally. In a synchronous system, where the nodes periodically produce the same data patterns, it is possible to manage with a static arbitration. Each node can produce data to the net at a predefined time. Any change in the order of the data production needs a new global synchronization. The network is called a time division multiplexed (TDM) bus, when a defined amount of communicated channels fit in the time of one master-clock period. TDM busses are very common in digital audio processing. A major drawback of a common bus is the limited amount of nodes and the demand of short connection lengths of the bus. For example a common bus implemented in a VME[3] sized cabinet

[3]Versa Module Europe, industrial standard

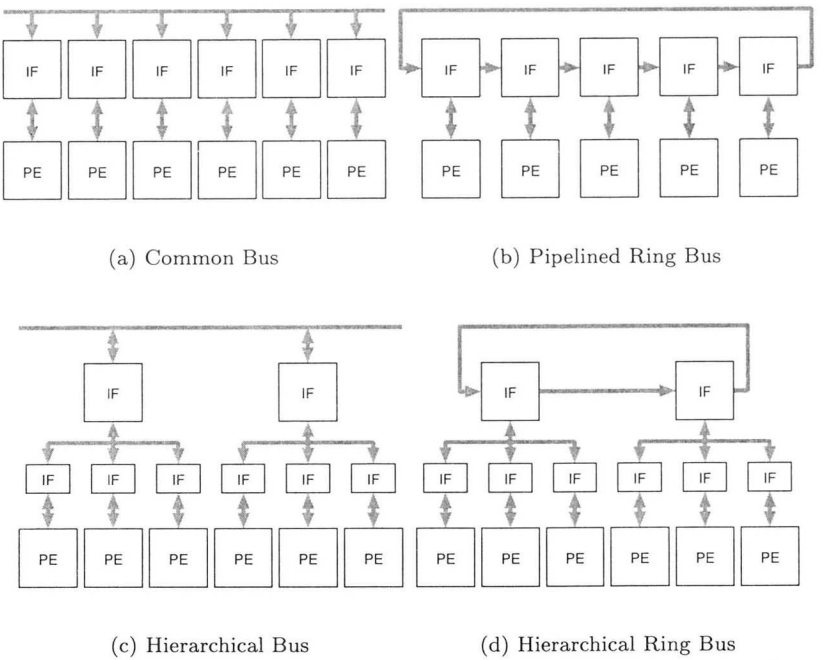


Figure 3.11: Bus Structures allowing broadcast capabilities.

allows us to combine a maximum of 21 boards. An extension of a high speed common bus is due to its electrical characteristics impossible. Considering the network security of a common bus, the network still operates when one node stops working. Using special hardware components it is even possible to exchange one node while the network is running. Such features are important in highly sensitive environments like broadcasting studios. A loss of the audio signal can be fatal when one node stops working.

Pipelined Ring Bus. A ring bus can be seen as a circular linear shift register, where every node stores a data word. To apply a broadcast on a ring bus, every data value has to proceed through the complete network once. This implies a maximal latency of n clock cycles for a network

with n nodes. For a synchronous communication, however, the latency can be reduced, since the communicated data is pipelined through the network. It requires that every node produces *only one joined data block* and appends its block directly after its predecessor. If the next sending node is not the direct neighbor, but is at a distance of x nodes, $(x-1)$ empty slots appear in the data stream of the network.

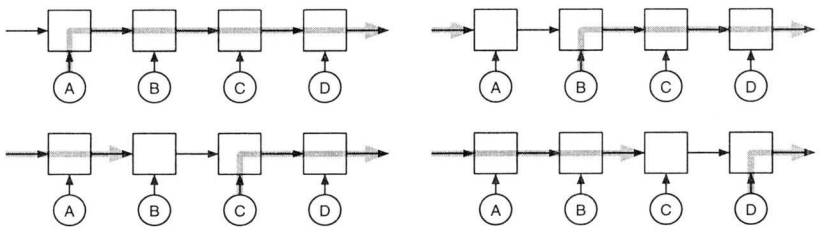


Figure 3.12: During one communication cycle, every node sequentially produces its data block to the ring bus.

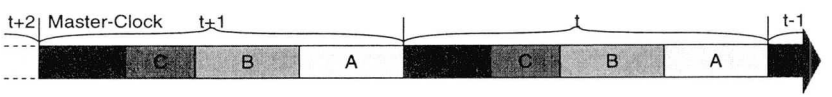


Figure 3.13: The data-flow of Fig. 3.12 is shown, as it proceeds through the network.

For example looking at an all-to-all broadcasting during one master-clock cycle (Fig. 3.12), the first node begins to produce the first contiguous data block to the network. After the last value of this block was passed to the second node, the next data block is produced by the second node. As shown in Fig. 3.13, all communicated data travels trough the network as a combined block.

One problem arises at the beginning and the end of a communication cycle. The entire network pipeline has to be filled and emptied. As an example, when a ring bus is used for digital audio with 60 nodes and the communication performance is 500 audio channels, transmitted at 25MHz, filling and emptying decreases the bandwidth by almost 25%. 60 unused audio channels appear for each, filling and emptying! To avoid this overhead, the communication cycles have to be pipelined

as well. Fig. 3.14 demonstrates the communication time t_c and the synchronization time t_s at the nodes. At the beginning of the communication cycle the pipeline has to be filled. Except of the first node, all other nodes have to wait. The same effect arrives at the end of the communication cycle. The first node has to wait until all other nodes have finished communicating. The synchronization time t_s is long. In

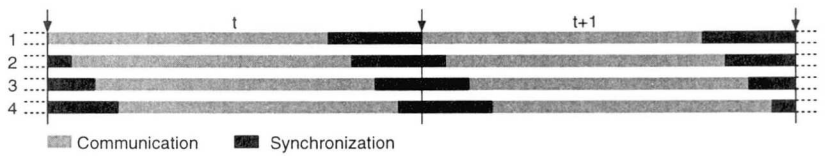


Figure 3.14: The communication and synchronization time at each node for a globally synchronized ring bus.

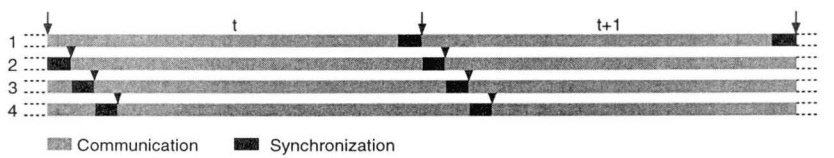


Figure 3.15: The communication and synchronization time at each node for a pipelined synchronized ring bus.

Fig. 3.15 t_s is shorter and arrives consecutively at each node. The communication cycle starts individually at the nodes.

The important advantage of a pipelined ring bus is its scalability. As only close neighbors are connected, very large ring bus architectures at high clock rates are feasible. A major drawback lies in operation security. In contrast to a common bus, the correct working of the network depends on each single node. If one node gives up operation, the complete network stops working.

Hierarchical Common Bus A hierarchical bus gives the possibility to reduce the hardware overhead for a parallel processor system, since only one interface to the global interprocessor network is needed for a

cluster of processors. Local interfaces within the cluster are less complex than the global interfaces and can be realized more economically . The broadcast capabilities are comparable with a common bus, but more processors can be reached. However, two aspects have to be considered when implementing a hierarchical architecture:

- The bandwidth of the local bus has to be shared by all processors in the same cluster.
- Data consistency has to be guaranteed within the cluster.

Normally, the local bus of a processor cluster is implemented as shared memory. Any access of a processor to the interface results in a arbitration with other processors. Only a limited amount of accesses to the interface are possible within one master-clock. The number of shared memory accesses per master-clock is, in general, equal to the number of processing cycles on one processor. Therefore, the size of a cluster depends on the type of processing modules that run on the processors. If the modules use much data from the shared memory, most of processing time is lost in arbitration of the local bus. Let $a(f)$ denote the ratio of shared memory access cycles $m(f)$ versus processing cycles $p(f)$ for a processing module f:

$$\frac{m(f)}{p(f)} = a(f) \tag{3.9}$$

If a reaches 1, processing and shared memory accesses are equal and processing cycles are not the limiting factor within a cluster. For example in a summing module $a(f) = 1/2$ because for every channel that has to be added two processing cycles are necessary. For modules, where $a(f)$ is close to 1, the cluster size has to reach 1, under the assumption, that $p(f)$ and $m(f)$ use the same time on a processor.

Local data inconsistency can occur when a processor uses a channel, which is produced by a processor of the same cluster. Take a module m_1 on processor A, that needs a channel from a module m_2 on processor B. If a double buffer is used (Fig. 3.10) and A and B are in the same cluster, it is not known a priori when the channel for module m_1 is produced by m_2. Thus m_1 possibly gets an old channel, communicated over the interprocessor network. There are two solutions for this problem: First,

the modules in a processor cluster are scheduled according to their precedence constraints. Second, a triple buffer is implemented.

In the first case an a priori timing list of all functions in the cluster is needed. However, because of the shared memory arbitration the predictability is not guaranteed. The second method, where read, write and communication buffer are separated, is simpler (Fig. 3.16). The three buffers switch after a communication cycle. The communication-buffer becomes the new read buffer, the write buffer is going to be the communication buffer and the old read buffer switches to a new write buffer. This way, all data produced in one cluster is first communicated through the network before it arrives in the read buffer.

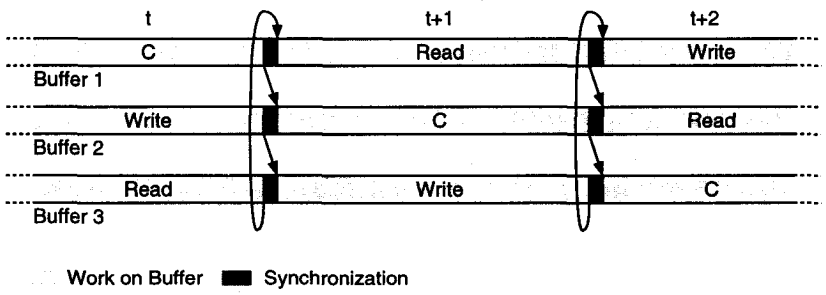


Figure 3.16: Triple buffer for data consistency in a hierarchical bus system. Read, write, and communication is performed on different buffers during one master-clock.

Hierarchical Ring Bus A hierarchical ring bus joins the properties of a ring bus and a hierarchical common bus. Large systems with a moderate hardware overhead are feasible. Like in a pipelined ring bus, the operation security depends on each node.

3.5. Synchronization

Different path lengths in a graph need to be synchronized. Such a synchronization takes place, where several paths come together. Let modules, that are located at a meeting point of paths, be called *merging modules*.

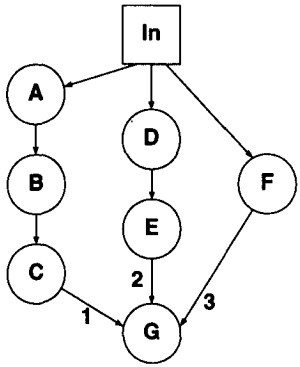


Figure 3.17: Different path lengths need to be synchronized at merging modules. Here G is a merging module.

Thus data values of shorter paths need to be stored at merging modules in a FIFO[4] structured buffer. The longest path going into the merging module defines the length of the buffer, where the data values from shorter paths are stored. Since every communication cycle of the interprocessor network corresponds to a master-clock, the FIFO buffer turns out to be a history buffer of the last communications. The implementation of FIFO buffers is a complicated and costly procedure, when for every channel going into a module such a buffer has to be realized. Therefore the implementation of a global history, where *all* incoming data is stored, can reduce the processing overhead massively. The next section discusses a solution with the example of a summing module.

3.5.1. Two Dimensional Channel Mapping

The most important merging module in a mixing console is the summing module, which is responsible for the summation of a complete summing bus (Fig. 2.2). A summing module usually has the form of large polynoms (Eq. 3.10), where the channels, that have to be summed, depend on time and are weighted with a certain factor p_i.

$$y(t) = p_1 * x_1(t - d_1) + p_2 * x_2(t - d_2) + ... + p_n * x_n(t - d_n) \quad (3.10)$$

[4]First In First Out

A straight forward implementation requires a FIFO buffering of every summing channel. Another possibility is the storing of the channels in a two dimensional block, where the channels appear as lines and the columns as a history. This grouping and saving of data into the memory should work autonomously without using processing power.

For example consider the implementation of Eq. 3.11. As shown in Fig. 3.18, it is possible to address the input channels $x_i(t - d_i)$ directly out of the memory.

$$y(t) = p_0 * x_2(t-1) + p_1 * x_4(t-4) + p_2 * x_{n-1}(t-3) \qquad (3.11)$$

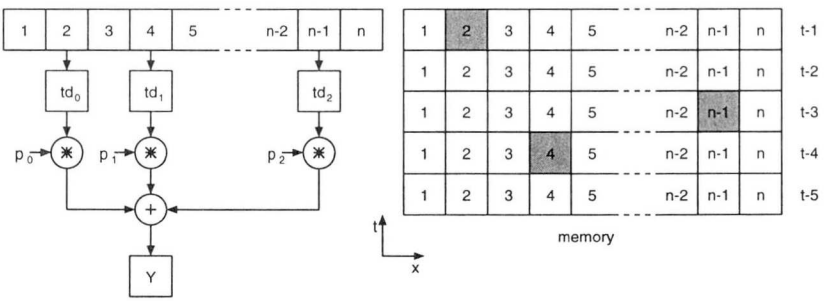


Figure 3.18: Each row of the memory corresponds to one communication, each column to a global channel. The input values $x_i(t - d_i)$ of Eq. 3.11 are marked.

The realization on a DSP is similar to a standard FIR[5] filter implementation, where a multiply-accumulate instruction (MAC) is applied. One extra processing cycle has to be inserted to get the offset d_i to the correct data value in the right time slot (row) in the history. An implementation on a DSP-96002 is shown as follows:

```
...
do   #N,_end                          ; Hardware Loop N-times
fmpy d4,d6,d1 fadd d1,d0              ; MAC instruction
           x:(r1+n1),d4 y:(r2)+,d6    ;     and fetch new x and p
move x:(r0)+,n1                       ; Get new offset d
_end                                  ; End Loop
...
```

[5]Finite Impulse Response

Therefore, $2n$ instruction cycles are used for the summing of n time dependent and weighted values on a DSP. The same concept can be applied for any merging module. The processing overhead for the synchronization of audio channels remains constant and in general is one extra processing cycle compared to an FIR implementation.

3.6. Parameter Processing and Communication

Each module running on the PEs allows the controlling of its behavior in real-time. Therefore every module needs to receive a certain amount of parameters directly from the desk. The communication concept described in section 3.4 defines a homogeneous communication of global audio channels. No separate bus or connection is provided for the communication of parameters from the desk to the modules. To keep the communication network as homogeneous as possible it is likely to transfer parameters through global audio channels. The update rate of parameters is much slower than the global master-clock. It may take several milliseconds until a module operates with a new set of parameters. Although there is no explicit rule what latency is allowed for parameter communication, a maximum time of 10ms is convenient in professional audio engineering. This time is required for the mixing of audio and video data, since the period of a video signal is about 20ms. A simple method for transferring parameters through global audio channels is the time–multiplexing of audio channels. Instead of transferring audio, each data value contains one parameter. Only a few audio channels are necessary to communicate a large amount of parameters. For instance the communication of 5000 parameters with an update rate of 10ms demands 10 global audio channels, where every channel communicates 500 parameters.

A module needs to access its parameters directly. Therefore, parameters are stored in the memory of the processor at a predefined location. The same strategy as described in section 3.5.1 can be applied. The channels, that transfer parameters, build a complete block of parameters in the memory. The parameters appear in columns of the audio history at a absolute positions and are refreshed with each period of the ring buffer (Fig. 3.19). The address of each parameter can

be calculated in advance and integrated in every module. Clearly, the depth of the channel history and the update rate of the parameters are related. If 500 parameters are multiplexed on one audio channel, the history within the memory has a depth of 500 values per channel.

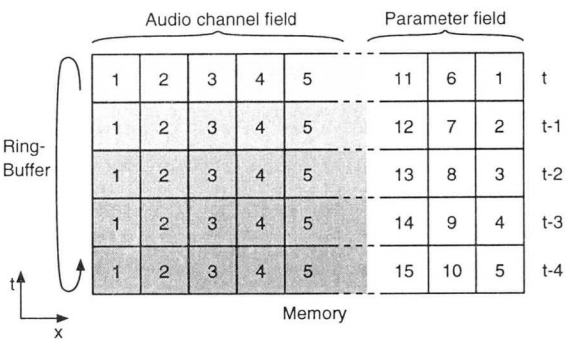


Figure 3.19: Parameters are multiplexed on some audio channels and appear in the memory at absolute positions.

3.6.1. Synchronous and asynchronous tasks

Since the update rate of the parameters is slower than the audio sampling rate it is not useful to handle new parameters on every audio clock cycle. A slower preprocessing task can prepare parameters for the fast processing task running with the audio sampling rate. If 10 processing cycles are used to manage a parameter and if it is done in a separate, periodic preprocessing task once every 10ms, up to 50 parameters can be handled and an average of only 1 processing cycle per audio clock is used. It is therefore useful to have tasks with different periods in a mixing-engine. Also an event driven, asynchronous task is possible. Such a task can act on events or changes of single parameters coming from the desk. As a result we distinguish three different types of tasks:

- **Definition: Synchronous Task** *A synchronous task S represents a set processing modules $m_i \in S$. Its execution time t_e on a processor p fulfills $t_e \leq \frac{1}{f_s}$, where f_s is the audio sampling frequency. The task is processed during one master-clock.*

- **Definition: Buffer Synchronous Task** *A buffer synchronous task B represents a set processing modules $m_i \in B$. Its execution time t_e on a processor p fulfills $t_e \leq l * \frac{1}{f_s}$, where l denotes the length of a data buffer. During this time a complete buffer of length l is produced.*

- **Definition: Asynchronous Task** *An asynchronous task A represents a set processing modules $m_i \in A$. It is executed after an external event appeared. Its execution time t_e on a processor p is not limited.*

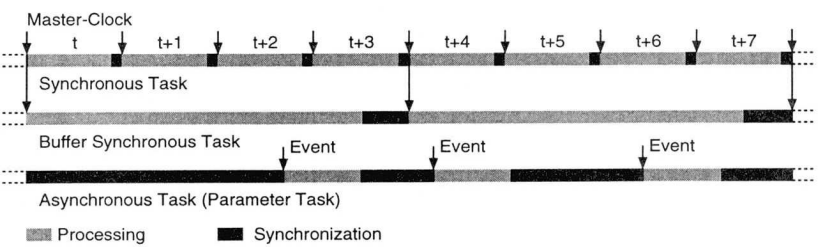


Figure 3.20: Three different types of processing tasks can be implemented. They synchronize either with each master-clock or after several master-clocks or according to an external event.

A *synchronous task* is the classical processing task which is repeated every master-clock cycle. It is built out of a set of audio modules according to the configuration procedure described in section 3.2. The cost of a processing cycle in such tasks is extremely high. Therefore only the minimum amount of code is integrated in the task. Input and output data are loaded and stored at predefined memory addresses. Parameters, that control the task, are preprocessed by another task and located at a defined position.

Buffer synchronous tasks are also periodic tasks, but they typically work on a large data buffer or use more than one master-clock cycle to finish. An example for such a task is a parameter task or a Fast Fourier Transformation (FFT). The processing is done on a double-buffer during several master-clocks. At the end of the processing the produced

buffer is communicated as a standard audio channel through the communication network. In the meantime a new buffer is prepared. The difference to the method of double-buffering described in section 3.4.1 lies in the buffer size. The processing time of a buffer synchronous task defines the size of the double buffer. This way it is possible to break the limit of processing time given by the master-clock. A module which includes a 1024 FFT can run more than 20ms until a new buffer of 1024 values has to be ready. Of course, every communicated audio channel needs to have valid data words on every master-clock. Consequently a buffer synchronous task always has to produce a complete buffer.

Asynchronous tasks work independently in the synchronous system. This can be a parameter task that is responsible to produce a complete buffer of parameters. The task reacts on changes of parameters coming from the desk and places its results in a buffer, a status field, where changes can be made at any time. Although the buffer is communicated through the communication network periodically, there is no need to produce the entire buffer every time. Since the buffer is communicated autonomously by the hardware, the communication overhead does not influence the activities of the processor.

It is intended that one processor runs only one type of task. A mixture of different task-types on one processor is possible. However, this would result in a higher system software overhead. The fast reaction time of the asynchronous task is bought with long synchronization times and with a lower processor usage. But, in general, only one processor in a full system performs an asynchronous task and the lower efficiency is tolerable.

3.7. Summary

A concept for the static configuration of a mixing-engine is given. The functionality of the system is described in a directed acyclic audio graph. A DAAG with audio channels and processing modules results. It is mapped onto the parallel processor system by sequentially performing module assignment, communication and synchronization. This separation into three steps is possible under the condition that the in-

terprocessor network is a broadcast network and the communication bandwidth remains constant. If this bandwidth is not able to satisfy all communication requirements, modules are grouped to larger modules with local communication. This way, the granularity is increased and global communication is reduced. However, the higher grain size results in less used processing resources and in a higher number of processors.

For the task allocation, all communication costs are neglected. Modules are assigned using the FFD bin-packing heuristic. It is shown, that this algorithm shows extremely good results when used in digital audio. The expected loss of processing cycles is even predictable when modules are randomly distributed. It increases linearly with the amount of PEs.

Different topologies of interprocessor networks with broadcast capability are discussed and tested for the usage in digital audio. Amount of processing elements, processor to network interfaces and communication bandwidth are considered.

For the synchronization of different path length in the DAAG a memory concept is presented. It allows the accumulation of time dependent audio channels with very low processing overhead. Additionally, parameters for the real-time controlling of modules are communicated using the same memory concept. They appear at predefined locations in the processors memory.

Chapter 4

Audio Signal Processing

4.1. Real-time Processing

The number of signal processing modules in a digital mixing console is not significantly high. There are several main modules with typical functionalities used in the audio strips of the mixing console. A strip may contain any number and combination of such modules. As the designer of a digital mixing console is free to configure a strip according to his needs, only a basic outline of a strip is given (Fig. 4.1).

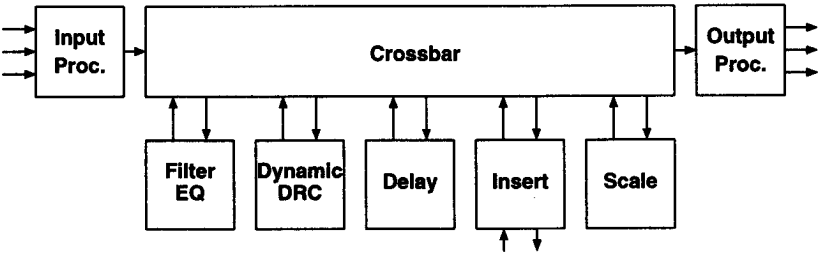


Figure 4.1: A general strip consists of different function modules. The sequential order can be altered during runtime using a crossbar module.

The main processing modules are connected to a crossbar switch. The sequential order of the modules can be altered even during runtime. The input- and output-processing boxes in Fig. 4.1 represent a fix configured combination of scale modules that allow a regulated incoming and outgoing of signals. Since these sections are derivations of a standard scale–module, they are not further examined.

Fig. 4.2 depicts three approaches of how an audio signal can be manipulated. First, the spectrum of a signal can be modified by increasing or decreasing frequency parts. This task is performed by a

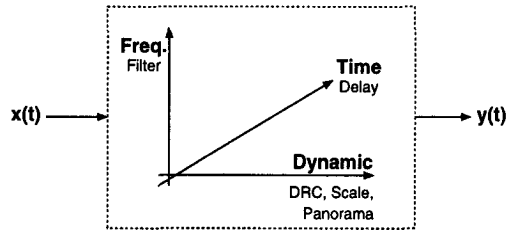


Figure 4.2: The manipulation of an audio signal is done in three directions: frequency, dynamic and time.

filter or equalizer module. Second, the dynamical behaviour of a signal can be controlled. This is done by regulating the volume automatically according to the amplitude of an audio signal. Compression, expansion, and limitation of the dynamic range are possible this way. A third manipulation is done in time. By an adjustable delay, the arrival time at the listener of a specific signal can be altered. Therefore, the perception of this signal is modified in comparison to other signals. All these influences are principly independent of each other. For example, changes in the delay module do not alter the dynamic range of the signal. Modifications in the scale module do not change the characteristic of the filter/equalizer module. One restriction arises: As the dynamic range control (DRC) module builds non–linear function, the frequency domain is altered, too. Conclusively, the exact characteristic of the signal depends on the position of the different modules in the strip. This is one reason for the changeability of modules in the strip using a crossbar module. The other purpose of the crossbar is to define the position of the insert module. Hence, external devices are connected at a choosable insert-point.

The various manipulation possibilities and the corresponding modules are discussed.

4.2. Filter and Equalizer

A primary module in a strip is the digital audio filter. This module makes changes in the spectral characteristic of the audio signal possible. Apart from the standard low-pass and high-pass filters, there are several

other filters used for the modification in the frequency domain.

4.2.1. Filter Types

A variety of filter types are used for audio recording and broadcasting. Fig. 4.3 shows the characteristic of each type in the frequency domain.

- *High-pass (HP)* and *low-pass (LP)* filters are the most popular filters. They control the margins in the spectrum of an audio signal. Second and fourth order filters are commonly used. The cut-off frequency f_c is adjustable in real-time.
- *High frequency shelving (HFS)* and *low frequency shelving (LFS)* filters allow to increase or decrease the amplitude of a signal starting from the cut-off frequency f_c by a factor V_0.
- *Peak* filters allow to manipulate the amplitude at a defined frequency f_c. Also the gain factor V_0 and Q-factor can be modified in real-time. Normally, several peak filters are combined and together build an *equalizer*.
- The *notch* filter is a special kind of a peak filter. It is used to remove certain frequencies out of the spectrum. Therefore, it is designed in a higher order.

4.2.2. Implementation

Filters can be implemented in a recursive or a non-recursive structure. The two forms differ in the characteristic of the impulse response. Non-recursive filters have a finite impulse response (FIR) and supply linear phase transmission. The guaranteed stability and the linear phase transmission are important advantages of FIR filters. The use of FIR filters in digital audio systems is restricted due to the high processing power needed. Recursive filters have an infinite impulse response (IIR) and can become instable with some coefficients. Due to the low processing power they are still the most frequently implemented filter structures in digital audio processing. The non-linear phase transmission of IIR filters resemble the filter implementation in analog mixing

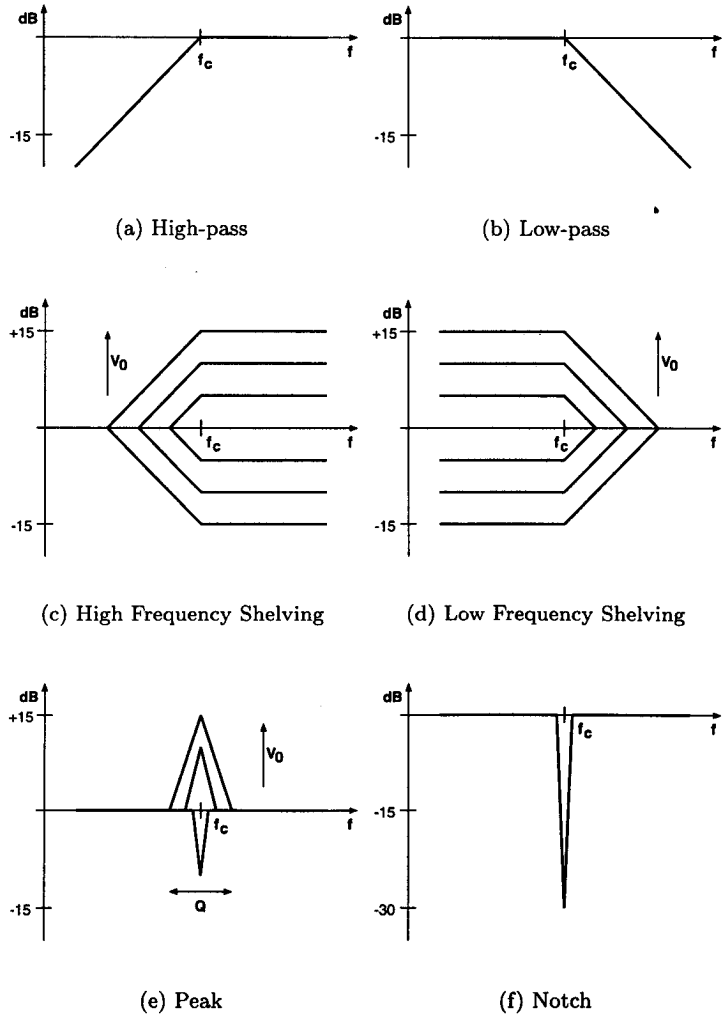


Figure 4.3: The most used filter types for audio processing. Cut-off frequency f_c, gain factor V_0 and Q-factor are adjustable in real-time.

consoles, which also provide non-linear phase transmission. As a result, audio specialists prefer IIR filters and accept the non-linear phase behaviour as a special feature.

A N^{th} order digital filter can be represented by the equation

$$\frac{Y(z)}{X(z)} = H(z) = \frac{N(z)}{D(z)} = \frac{N(z)}{1 + D'(z)} \tag{4.1}$$

where

$$N(z) = \sum_{i=0}^{N} a_i z^{-i} \tag{4.2}$$

and

$$D'(z) = \sum_{i=1}^{N} b_i z^{-i} \tag{4.3}$$

From Eq. 4.1 we can write

$$Y(z) = N(z)X(z) - D'(z)Y(z) = U_1(z) + U_2(z) \tag{4.4}$$

where $U_1(z)$ and $U_2(z)$ are two individual FIR filters [Ant93]. As shown

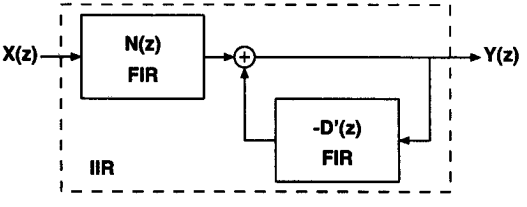


Figure 4.4: An infinite impulse response (IIR) filter can be divided into two individual FIR filter parts.

in Fig. 4.4, an IIR filter can be constructed by connecting two FIR filters together. As an example, a 2^{nd} order IIR filter has the form

$$\frac{Y(z)}{X(z)} = H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \tag{4.5}$$

Typically no higher classes than second-order IIR filters are designed for digital audio processing. The reason for this lies in the sensitivity of the filter-coefficients. Since the coefficients are quantized, small errors are introduced in the amplitude and phase responses of the filter [Jac86, Ant93]. As a result, these errors increase with the order of the filter and become intolerable for large Ns. A possibility to overcome the coefficient quantization effects is to connect multiple first- and

second-order filters in cascade or in parallel [Che96]. The cascade form of several 2^{nd} order filters is often found in digital audio processing. It is represented by the equation

$$H(z) = \prod_{i=1}^{M} \frac{a_{i0} + a_{i1}z^{-1} + a_{i2}z^{-2}}{1 + b_{i1}z^{-1} + b_{i2}z^{-2}} \tag{4.6}$$

The realization of a 2^{nd} order filter on a digital signal processor can be done in different ways. Only two are illustrated here. The most obvious recursive form results out of Eq. 4.4 and has the structure

$$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} - b_1y_{n-1} - b_2y_{n-2} \tag{4.7}$$

which is called *Direct Form I*. An implementation on a processor requires four delay units, five multiplications and four additions. A second form can be achieved by defining

$$W(z) = \frac{1}{1 + b_1z^{-1} + b_2z^{-2}}X(z) \tag{4.8}$$

and therefore

$$Y(z) = (a_0 + a_1z^{-1} + a_2z^{-2})W(z) \tag{4.9}$$

With Eq.4.8 and Eq.4.9, the recursive form for w_n and y_n, respectively, can be obtained

$$w_n = x_n - b_1w_{n-1} - b_2w_{n-2} \tag{4.10}$$

$$y_n = a_0w_n + a_1w_{n-1} + a_2w_{n-2} \tag{4.11}$$

This form is called *Direct Form II* and its realization needs only two delay units compared to the Direct Form I. Although two equations have to be calculated, only one multiplier and one accumulator are needed. Since, after w_n is calculated, the same resources are used for y_n. More important is the fact that only two intermediate states have to be stored for the Direct Form II. A DSP, in general, is able to perform one multiplication, one accumulation and two data moves per cycle. The data manipulation of the intermediate states w_{n-1} and w_{n-2} can be done during the multiply-accumulate (MAC) instructions. This makes it possible to implement the Direct Form II in only 7 processing cycles on a DSP [Mot89].

The cascading of several second order filters can be realized even more efficiently. After an initialization phase, only 5 instruction cycles per filter are needed [Mot89].

Apart from the discussed forms there are other time domain expressions such as the Transposed Form II, Gold&Radar, Kingsbury and Zölzer. These structures provide better signal to noise ratios in fixed point DSP implementations. However, the processing cost is in general higher. Detailed considerations about different time domain expressions can be found in [Zoe96].

A fundamental problem is to find an optimal approximation of the coefficients in the form $H(z)$ for a given transfer-function specification. The problem is even more complex for IIR filters. Some filter approximations have been found to be consistently useful and their characteristics is found in the literature. Common approximations for the low-pass filters are [Lam79]:

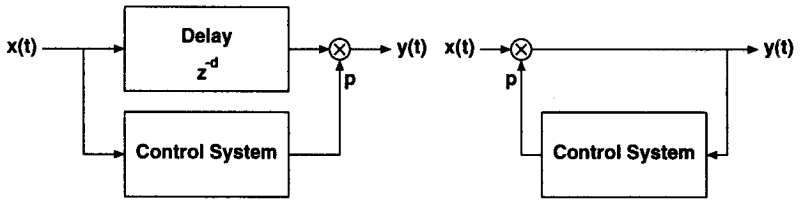
- The *Butterworth* filter, characterized by a monotonically decreasing amplitude function of ω for $\omega \geq 0$.
- The *Chebyshev* filter, characterized by an equiripple amplitude response in the passband and a monotonically decreasing amplitude response in the stopband.
- The *inverse Chebyshev* filter, characterized by a monotonically decreasing amplitude response in the passband and an amplitude response in the stopband.
- The *elliptic* or *Cauer* filter, characterized by a equiripple response in both, the passband and the stopband.
- The *Bessel* filter, characterized as having an optimally linear phase response.

The approximations give coefficients for a transfer function $H(s)$ in the S-domain. Finding the corresponding coefficients in the Z-domain can be done using a suitable transformation. The *Bilinear* transformation has shown good results:

$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{4.12}$$

4.3. Dynamic Range Control

The dynamic range control (DRC) devices offer the multiplicative manipulation of the audio signal. The dynamic range can be specified as the difference between the loudest and the quietest passage of an audio signal in dB[1]. The aim of such a module is to increase or decrease the dynamic range in a prescribed way. Furthermore, the protection of signal overload without introducing perceptible distortion is an important task.



(a) Feedforward control (b) Feedback control

Figure 4.5: Dynamic range control device implemented as feedforward and feedback controlled system.

Fig. 4.5 depicts two principle arrangements of a DRC system. The *feedforward controlled* DRC, as mainly used in the digital domain, has a preliminary delay. The signal is measured in a side-chain, where a corresponding gain factor p is calculated. The control system needs a precisely specified control characteristic. The delay of d samples in the main signal gives the possibility to react to fast changes in the audio signal in advance and so, overshoots can be eliminated. The *feedback controlled* DRC is used in conventional analog systems. The major drawback of this structures is the possible introduction of overshoots and distortion in the audio signal. The control system reacts with the gain factor p after the signal is already at the output. Thus, short excessive levels cannot be prevented because of the delay in the control system. However, this feedback chain helps to keep the control system

[1] dezi-Bel

simple. An implementation in the analog domain is easier.

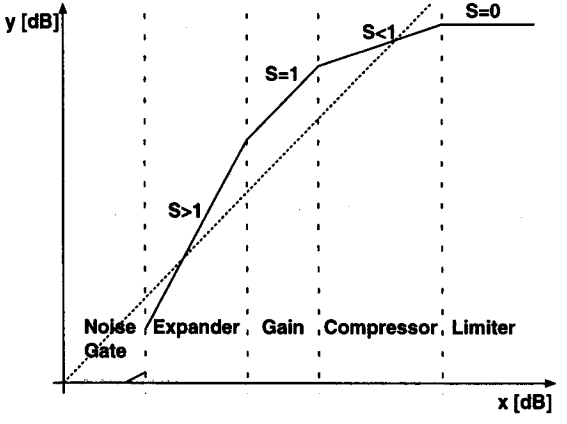


Figure 4.6: The dynamic range control characteristic covers five different sections.

As shown Fig. 4.6 DRC covers a signal manipulation in five parts of the loudness of an input signal: Noise-gate, expander, gain, compressor and limiter. The noise gate allows to mute or even suppress the signal when it is below a certain level. Expander and compressor are used to enlarge and shorten the dynamic range. A limitation of the signal is given by the limiter section of the DRC. The gain section completes the DRC curve and represents the multiplication by a constant factor. The slope S has a constant value in each of these sections in the xy-level diagram. Not all sections have to be implemented. One or more sections can be omitted depending on the field, where the mixing console is used.

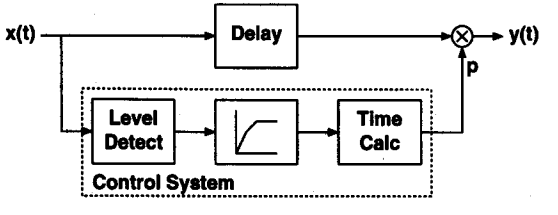


Figure 4.7: The DRC control system.

The control system, shown in Fig. 4.7, consists of a level detector, a

DRC characteristic (according to Fig. 4.6) and a time calculation section. For the limiter, a simple peak detector is sufficient to measure the signal level. In all other cases, *ms*-detection[2] or *rms*-detection[3] may be used, since it gives a better indication of loudness than peak values [McN84a]. The level detector is adjustable with system specific time constants. Depending on the audio signal, different transient constants are used for attack time T_a and release time T_r of the control system. The latter is also referred as recovery time or decay time. Fig. 4.8 de-

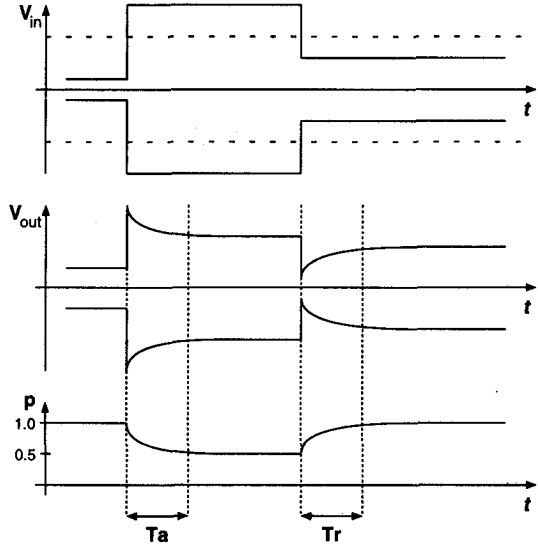


Figure 4.8: The behaviour of attack time T_a and release time T_r is shown after a brusque level change in the signal. This system has no preliminary delay. The resulting gain factor p is shown at the bottom.

picts the behaviour of a limiter control system and the resulting gain factor p for an abrupt high–level to low–level change. This example provides no preliminary delay. In some implementations also a hold time T_h is supported. It describes how long p remains constant after the input level began to decrease. This way an oscillation of the con-

[2] mean square
[3] root mean square

trol system can be prevented. There are several definitions for these parameters found in the literature. An IEC recommendation specifies the attack time for limiters as the time for an initial 6-dB overshoot to be reduced to within $2dB$ of its final value [IEC73]. The conventional definition in professional audio processing determines the attack time T_a as the time taken to achieve 63.2% of the total change of the gain [McN84a]. The definitions for T_r and T_h are exactly the same.

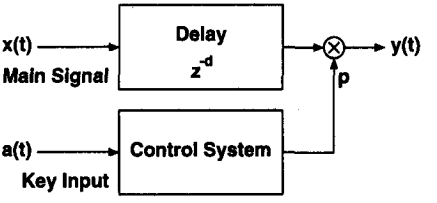


Figure 4.9: Linked dynamic range control allows the regulation of a signal with the loudness of another signal.

Another application of a DRC device is the linking of different audio channels where the level of a channel is controlled by the level of another one. There are different applications for such devices. For example, in audio broadcast studios a musical signal can be regulated directly with the voice signal of the speaker. The control chain and the main audio chain are separated as shown in Fig. 4.9.

4.4. Manipulation in Time

After the modifications of the spectral and dynamic characteristic of the audio signal the delay device offers the third direction of signal manipulation as depicted in Fig. 4.2. It allows us to include a variable delay into an audio signal. This relatively new element became important with the first digital mixing consoles, since a delay in the analog field is rather complicated and costly compared to the digital domain. The number of intermediate stored values and therefore the delay time is controlled in real-time. Although the computational aspect of this device is rather simple, its application can be powerful. The human perception of signals with different delays is extremely sensitive. One task is the time compensation of different signals, as it is

necessary, when a recording with a large number of signal acquisition devices (microphones) is performed. Another application is the artificial room modeling by installing a delay in every incoming signal. Finally, delays in sound reinforcement applications allow a selective orientation for the listener when the signals of the loudspeakers are individually postponed.

4.5. Real-Time Controlling

The controlling of processing modules in real-time leads to a central topic: Processing modules must be able to react on parameter changes in a sufficiently short time and without introducing audible distortion. A communication concept for transferring parameters from a control device to a processing module was discussed in section 3.6. It is shown that the update–time of parameters is about 10ms and considerably slower than the audio sampling rate. This update–time is enough in terms of real-time requirements in a recording studio. However, abrupt changes of parameters can incur high steps in the amplitude of the audio signal which leads to audible distortion. Especially the regulation of a parameter directly, with the ruler on the control-desk, can provoke massive changes of the parameter. This effect is called *zipper-noise* since, with some imagination, the tugging up and down of a zipper produces a similar sound. Other intolerable effects in the audio signal occur by changing the characteristic of a filter. Altering parameters such as cut-off frequency or gain factor imply the switch to new coefficients for the appropriate filter. Especially IIR filters can be extremely sensitive to such changes and even become instable. A next problem arises when more than one audio channel is available as input to a certain module. Switches from one input channel to the next produce unwanted, perceptible clicks. Finally, adjustable delays, where the delay time can be modified in real-time, can not be switched directly from one delay to another without misshaping the original audio signal. As a consequence a solution that allows smooth transitions from one characteristic to another must be found to eliminate these effects.

We can differentiate the following situations, where audible artifacts can occur:

1. Abrupt changes of a gain factor
2. Switches between different filters
3. Selection of audio signals
4. Changes of a delay

The different effects are discussed in details in the following sections.

4.5.1. Gain Factor Change

An abrupt change of a gain factor p has to be interpolated. A simple and powerful method is the filtering of p with a first order low-pass filter. Fig. 4.10 demonstrates the behaviour of a scaled signal with and without an interpolated gain factor.

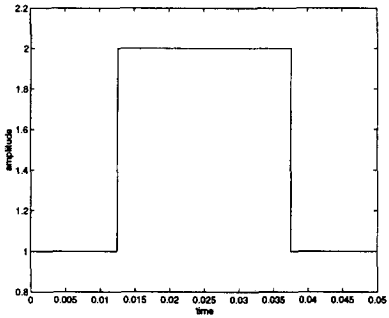
The time-domain expression of the applied first order filter can be written as

$$k_n = (1 - T)p_n + Tk_{n-1} \tag{4.13}$$

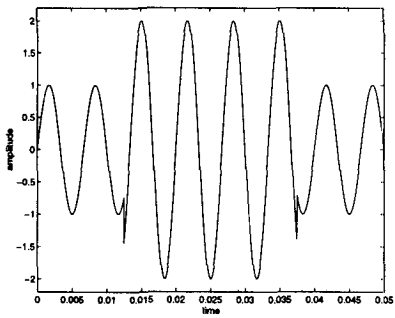
where p_n is the unfiltered gain factor and the time constant T satisfies $T < 1$. The audio signal can then be scaled directly by the resulting interpolated gain factor k_n. The performance of this method can be expressed in terms of processing overhead, that is needed compared to a non-interpolating solution. The realization of Eq. 4.13 requires one delay element, two multiplication and one accumulation. A minimum of three processing cycles are required on a DSP.

4.5.2. Filter Switch

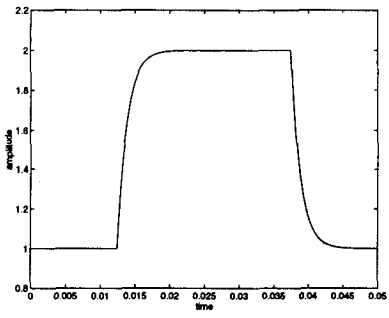
There are many discussions to be found in the literature about switching digital audio filters. Zölzer et al. [ZRB93] propose a filter switching method by calculating up to 60 intermediate filters to get a smooth transition from one filter to the next. This procedure, however, is very expensive in terms of processing cycles. Direct switches between different filter types turn out to be complicated. For example, the switch from a low-pass filter to a peak filter is done by a first switch to the ideal transfer function with $H(z) = 1$ and then change to the final filter. Another variant can be achieved by a state vector transformation [RC85]. This approach is quite costly, too, and numerical problems can occur.



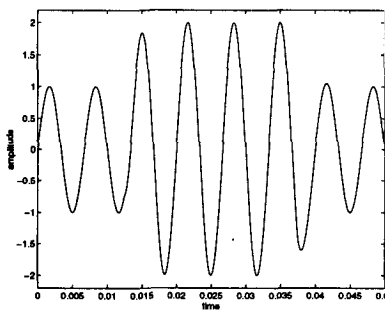
(a) Unfiltered gain change



(b) Signal with unfiltered gain



(c) LP filtered gain change

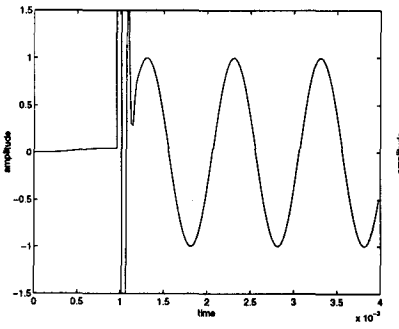


(d) Signal with LP filtered gain

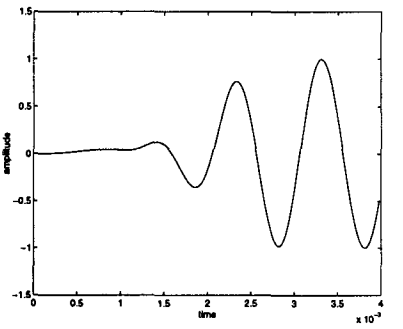
Figure 4.10: The gain factor is changed abruptly from 1 to 2 on a signal of 150Hz, sampled with a frequency of 48kHz.

Probably the easiest way for a filter switch is to fade out the signal by tuning the gain to zero, change to the new filter coefficients and finally fade the signal in. Although this method provides good acoustic results and requires only a minimum of processing overhead, the temporary loss of the signal is intolerable. Close to this idea is the interpolation by crossfading [MM94]. A filter switch is performed by calculating the old and the new filter simultaneously. Then, using a crossfade function $f(t)$, the signal from the old filter is faded out while the gain of the

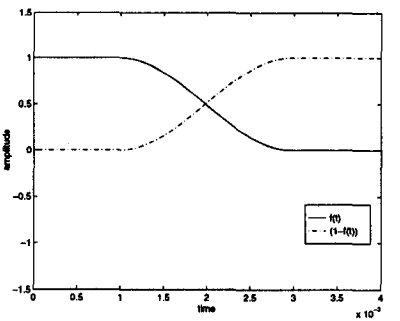
new signal is scaled by $(1 - f(t))$. The crossfade function $f(t)$ can principally be any function $f(t)$ with a smooth transition in the interval from 1 to 0. However, it has to be considered that the new filter needs a recovery time to become stable until it can be crossfaded with the old filter. Good results were obtained by using $f(t) = sin(t)$.



(a) Direct filter change



(b) Filter change with crossfade



(c) Crossfading function

Figure 4.11: The cut-off frequency of a 2^{nd} order low-pass filter is changed directly (a) and with a crossfading function (b) from 100 Hz to 10 kHz. The frequency of the filtered signal is 1 kHz.

Fig. 4.11 depicts a filter switch between two low-pass filters with the cut-off frequencies 100 Hz and 10 kHz, respectively. A direct switch

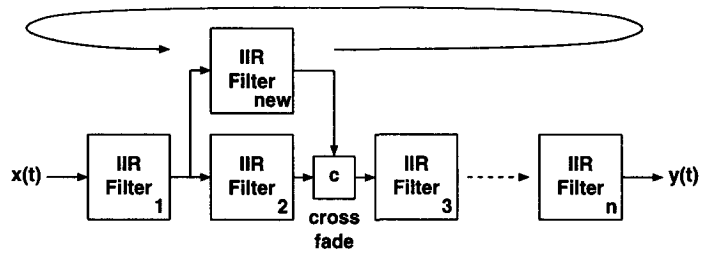


Figure 4.12: One extra filter is used for the interpolation of n sequentially ordered IIR filters. The latency for an complete update is then nt_c where t_c is the crossfade time.

leads to a high impulse in the 1 kHz signal. In this example a maximal amplitude of over 160 instead of 1 is reached. The distortion can totally be eliminated by crossfading. Of course, this variant is quiet costly since two complete filters have to be calculated instead of one. Filters in mixing-engines are mainly connected in a sequential order. As an example, an equalizer consists, in general, of four IIR filters serially connected. Therefore, processing cycles are reduced when only one filter is interpolated at a time. This way one supplementary filter is implemented and used for the interpolation of several connected filters (Fig. 4.12). If n filters are switched it takes a maximum time nt_c when t_c is the time for one complete crossfade. In Fig. 4.11 t_c is $2ms$ which is equivalent to 96 audio values with $f_s = 48kHz$. Shorter crossfade times turn out to be critical, because the new filter has to be stable before it can be tuned in. For example, an equalizer section with four individual IIR filters connected in series can be interpolated within $8ms$ and needs a processing overhead of 25%. Compared to other implementations [ZRB93] no extra processor is needed to prepare intermediate filter coefficients.

4.5.3. Signal Selection

The selection of a signal out of a number of channels means to replace the input–channel at a certain module with a new channel. This procedure is also parameter controlled and performed in real-time. When selecting an audio signal, audible distortion is added to the resulting sig-

nal. It originates from the brusque transition from one signal to another at the exact switch time and can be perceived as a "click". Smoothing the rest of the old signal and the beginning of the new signal at the transition will remove this effect. The same crossfade algorithm can be applied as described above. A simple fade-out/fade-in procedure may be convenient, if a short break in the signal is acceptable. However, the fading time should be sufficiently long, because the spectrum of the two involved signals can be completely different. A minimum of 2 to $5ms$ is ideal. The processing overhead for this method is low. The switched signal needs one extra multiplication with a fading factor.

4.5.4. Delay Change

A delay, in principle, is a static element, which postpones the audio signal for a certain time. Changing the delay will leave out samples, when the time is shortened, and it will introduce new, nonexisting samples, when the time is prolonged. It leads to a misshaped signal. Since during the change the deformed audio signal is not of interest, a short break in the signal is acceptable until the new delay is installed. Again, a fade-out/fade-in procedure with a short pause can be applied because of the tolerable loss of signal during change. The processing overhead is one extra multiplication, as discussed above.

4.6. Summary

Audio signal manipulation in a digital mixing console is performed in three different characteristics of the signal: The frequency domain, the dynamic range and the delay time. The corresponding processing modules are digital filters, dynamic range control devices and variable digital delays. The design and implementation of each module type on digital signal processors is presented. All processing modules are controlled in real-time. Since the update-time of parameters is slower than the audio sampling time, four concepts for interpolating real-time parameters are introduced. They eliminate perceptible distortion due to abrupt changes of gain factors, switches between different filters, selections of audio signals as input to a processing module and changes in digital delays. The concepts include simplicity and low processing

overhead when used in a digital mixing console.

Chapter 5

Implementation

5.1. Hardware-Platform MUSIC

The digital audio mixing-engine, described in this thesis, has been developed using the MUSIC[1] parallel computer. This system was built at the Electronics Laboratory of the Swiss Federal Institute of Technology (ETH) [GMS+92b, GMS+92a]. The MUSIC project started in 1990. A parallel computer with a processing performance comparable to large supercomputers was developed. The first application running on this system was the simulation of neural networks [Mue93]. Other applications like molecular dynamics [SGB+93, SMPvG94] and image processing [BKG95] were successfully implemented. All these applications are realized based on the standard MUSIC programming model. It is a SPMD[2] model with asynchronous communication among the processors. This model, however, cannot be used in digital audio processing. Nevertheless, using the MUSIC as a platform for a digital mixing-engine is still ideal because of the following reasons:

- Its main processor is a floating–point digital signal processor.
- The interprocessor network is not a fix implementation. It can be modified and adapted to individual needs.
- The system is fully scalable.

The advantage of using the MUSIC lies on one hand in the existing hardware that supplies all requirements in terms of parallelism, type of processor and scalability. On the other hand, the communication network is only given in its electrical characteristics of a ring bus. It still allows any implementation of specialized protocols.

[1] **Multiprocessor System with Intelligent Communication**
[2] Single Program Multiple Data

This section first describes the hardware architecture of MUSIC. Then the communication principle and the new synchronous communication on MUSIC is explained. Discussions about the implemented memory concept and digital data I/O conclude the section.

5.1.1. MUSIC's Architecture

Figure 5.1 gives an overview of the MUSIC hardware. As an example, a system with two processor boards and one I/O board is shown. The system is scalable and allows any combination and number of boards. Three processing elements fit on a standard board (22cm by 23cm) and up to 63 PEs can be connected together in a regular 19 inch rack. The power consumption of a fully assembled system is around 700 watts. A special I/O board contains one PE and a separate interface with an own communication controller. This opens the opportunity to connect hardware modules directly to the interprocessor network. Apart from the initialization of the communication controller no further processing is required for this I/O node. Fast data throughput is therefore guaranteed, which is required by real-time applications such as audio or image processing.

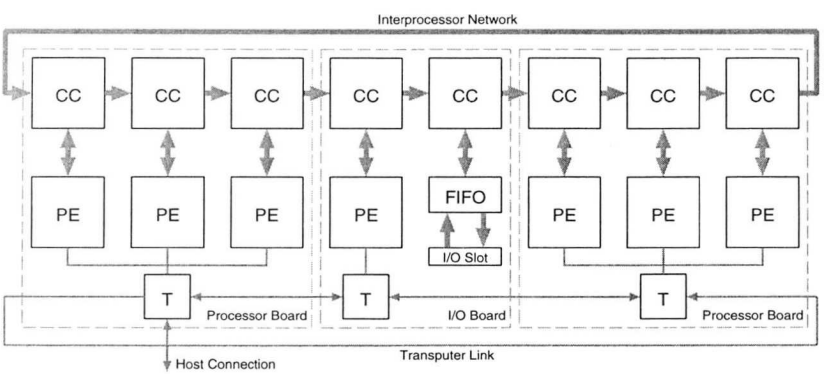


Figure 5.1: The MUSIC hardware architecture.

Each board also encloses a manager (Inmos T805 Transputer) connected to the host interface of the DSPs. It is responsible for data and program code up-loading and down-loading, respectively. Also time

measurements are performed using the transputer. Managers of different boards are connected trough their transputer links and form a standard transputer network. One Transputer is also connected with a host computer, which is a personal computer or a workstation. Via a transputer link adaptor or a SCSI interface data and program code is exchanged between host computer and MUSIC. A small kernel is running on the host computer and is responsible for the operation of the MUSIC system.

In Fig. 5.2 the main MUSIC board is shown. Three processing elements are arranged vertically on the board. Processor and communication controller of each PE are recognized. The board manager is located on top of the board.

Fig 5.3 demonstrates the block diagram of a single PE. It consists of a DSP, memory and a communication controller. The Motorola DSP-96002 is the main processor. It runs at 40MHz and is able to perform up to three floating point instructions per cycle. Two clock periods are necessary for one instruction cycle. It results a peak performance of 60 MFlops. The DSP provides two 32 Bit wide parallel bus interfaces. One is used as host interface to the board manager. The other interface connects the memory to the DSP. Four memory blocks are directly accessible. Two blocks of Static RAM (SRAM), 128 kWords each, serve as program and fast data memory. The other two blocks consist of dual-ported video RAM (VRAM), 256 kWords each. They also provide the interface to the interprocessor network. According to the data-flow to and from the interprocessor network they are called "producer" and "consumer" memory. The used VRAM is divided into two areas. One is random accessible dynamic RAM (DRAM). It builds a matrix of 512×512 memory locations and is addressed by the DSP. However, the access time is rather slow, compared with static RAM. The other area is serial access memory (SAM), a linear shift register with up to 512 data locations. It can be arranged either as input or as output port and it has the same size as one line in the DRAM part. The VRAM architecture allows to copy a complete line of the DRAM part into the SAM and vice versa within a very short time.

Each PE has its own communication controller (CC), which is responsible for the data-flow between the PE and the interprocessor net-

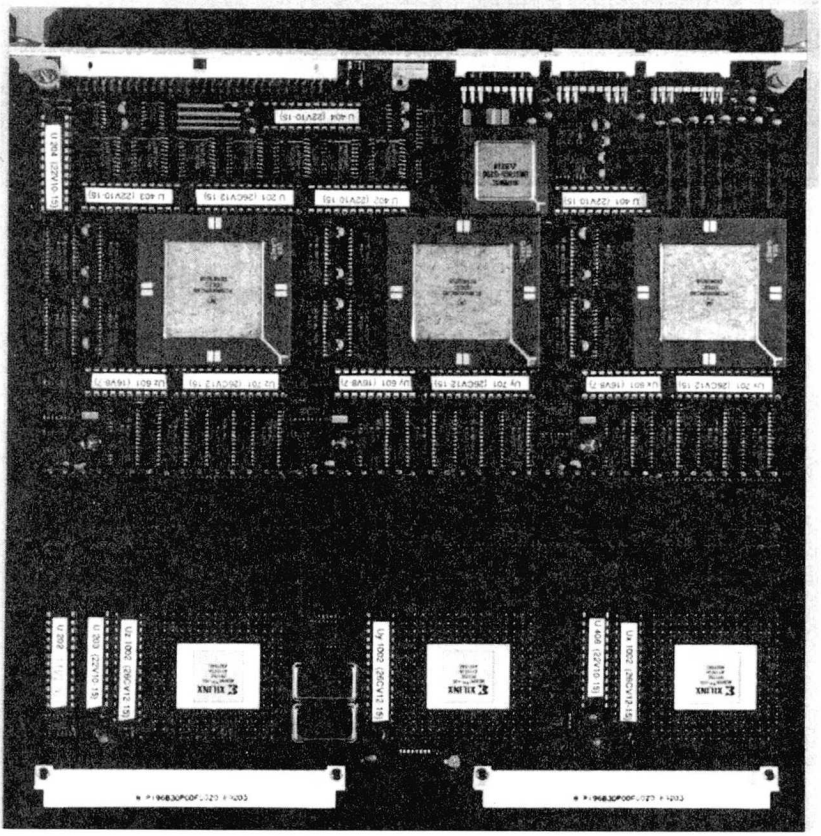


Figure 5.2: The main MUSIC board. Three PEs, each includes DSP, SRAM, DRAM and CC, are arranged vertically. The board manager, an Inmos T805, is located on top.

work. The CC is implemented in an FPGA[3] Xilinx XC3090. It fetches data from the SAM of the producer VRAM and writes arriving data into the SAM of the consumer VRAM. The CC transmits data from the PE to the network independently as long as there is valid data in the SAM left. If the producer SAM is empty, an interrupt is generated at the DSP. The processor initiates a transfer from a specified

[3]Field Programmable Gate Array

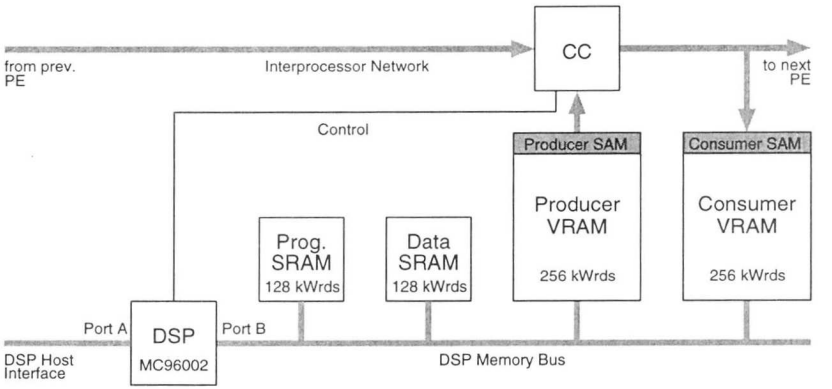


Figure 5.3: The layout of one processing element.

row of the DRAM part to the SAM. Then the CC can insert the next 512 values into the data stream of the interprocessor network. If the consumer SAM is full, the CC also interrupts the processor and the same procedure works in the other direction. Thus the processor is able to concentrate mainly on its program while data is communicated trough the network. The interrupt time is short. A VRAM transfer from or to the SAM is performed in one instruction, which lasts 200ns.

5.1.2. Communication Principle

The interprocessor network is a pipelined ring bus architecture. All data communicates trough the entire network and passes every node in the ring once (Fig. 5.1). The bus is 40 bit wide: 32 data bit and 8 token bit. The tokens are produced by the CCs and contain a unique identification of every node. This way a CC recognizes its own values, after they have traveled trough the network. Such values are outdated and removed from the bus. It is done by marking this data as invalid and a token with value 0 is passed to the next node. If a CC does not remove his data from the bus, the the next node will recognize this value as valid and will pass it again to the next node. The network will be blocked immediately because a CC may not overwrite a value with valid token. To keep track of the position, the valid data values are counted by the CC. The standard implementation of the MUSIC allows an asynchronous

communication of a three dimensional data space a maximum size of $2^{14} \times 2^{14} \times 2^{10}$. One counter is implemented for each dimension. Before a communication cycle starts, a new data space is defined and the controllers are initialized. The standard communication principle on the MUSIC is called intelligent communication (IC). Further discussions about IC are found in [Mue96].

5.1.3. Synchronous Communication on MUSIC

The implementation of an audio mixing-engine on the MUSIC required a redesign of the communication controller. A new synchronous communication controller (SCC) was designed in the same FPGA of the MUSIC processing element and works at 25MHz. It is able to transfer a maximum of 512 data values of 32 bit within one master-clock cycle. In contrast to the normal MUSIC communication, the transferred data field can be interpreted as a one dimensional set of global audio channels. As depicted in Fig. 5.4, the communication model is an all-to-all broadcasting where every processor produces only one part, but consumes all communicated channels. These parts are called producer window and consumer window, respectively.

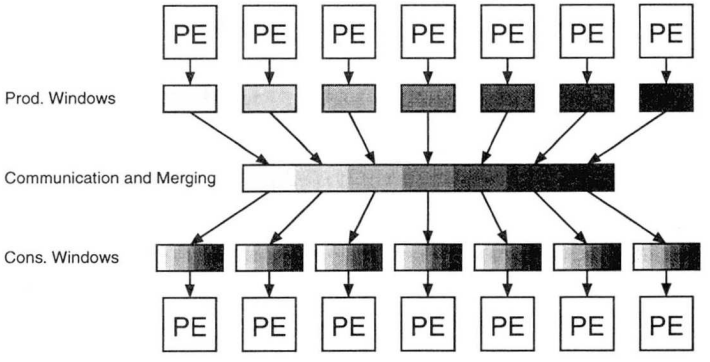


Figure 5.4: The synchronous communication on the MUSIC is an all-to-all broadcasting communication.

Let a communication cycle be the time representation of a defined master-clock period (Fig. 5.5). It is denoted by $A, B, \ldots$. On the rising

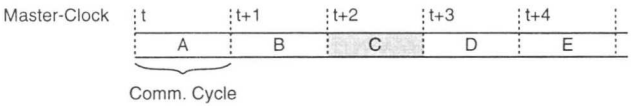


Figure 5.5: A communication cycle is a time representation of a defined master-clock period.

edge of the master-clock a new communication cycle is initiated. The first node produces a contiguous block of global audio channels – its producer window – to the network. The neighboring node on the right passes this first block with a delay of one network clock to the third node. At the same time it also consumes this data into his memory. After the second node has transmitted the last data word of the first block, it begins to produce its data block right behind the first block. In the meantime the third node transmits the first and the second data block, again delayed with a network clock. It also copies it into his memory. Thus all data values communicate through the network strictly ordered, beginning with the first value and ending with the last.

5.1.4. Pipelining Communication

As mentioned in section 3.4.2, filling and emptying a ring bus reduces communication bandwidth and should be avoided. In the MUSIC system each node consumes its data directly after the output of a node (Fig. 5.6).

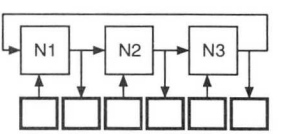


Figure 5.6: Each nodes consumes its own data directly after the output.

To use the full bandwidth, communication cycles can be interleaved. It requires that two successive communications are nested within one master-clock. On the other hand, a non-interleaved communication does not take advantage of the full communication bandwidth because the

ring bus has to be filled (but not emptied) during one master-clock. But, each communication cycle is coherent and the software management is easier. As an example, consider a pipelined ring bus with 3 nodes. Node one produces 1 value, node two produces 2 values, and node three produces 3 values (Fig. 5.7). The two possible communication methods

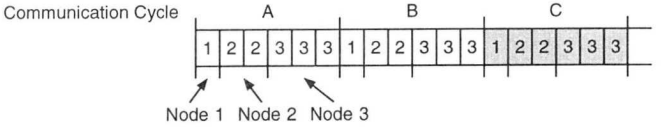
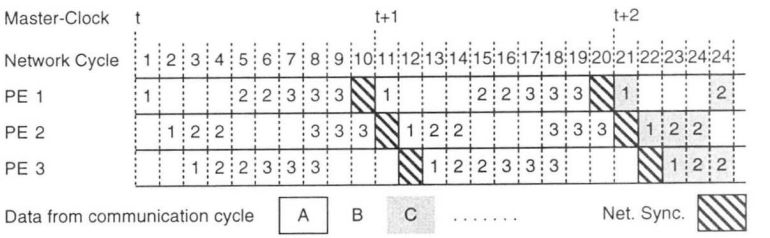


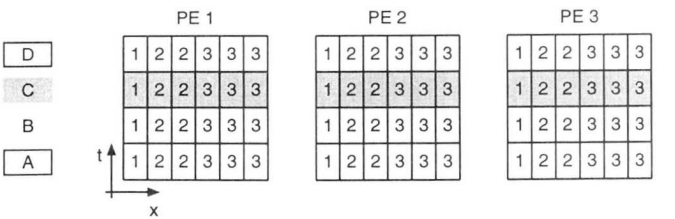
Figure 5.7: Example: In a system with three nodes 1, 2, and 3 values are produced by node one, two, and three. It is repeated every communication cycle.

are explained in the following paragraphs.

Non-interleaved Communication Non-interleaved communication is applied, when one coherent communication cycle is performed within one period of the master-clock. Fig. 5.8(a) depicts a non-interleaved communication on MUSIC. During master-clock t data from communication cycle A is transferred, during master-clock $(t+1)$ data from communication cycle B. A processor calculates new data from communication cycle A while B is transferred. The new data will then be part of communication cycle C. The ring bus is filled during each master-clock. For a system with n nodes, n unused channels appear in the data stream. The MUSIC can have up to 63 nodes installed. A theoretical maximum of 520 communication channels are possible at a network frequency of 25MHz and 48kHz sampling frequency. Therefore, the efficiency of a full MUSIC system with non-interleaved communication and 63 nodes is 87% or better, depending on the number of nodes. The communicated data is serially copied into the SAM of each PE. During the network synchronization, a SAM/RAM transfer is performed. The entire contents of the SAM is then copied into a row of the DRAM part. The transmitted data appears ordered in the local consumer memory. Each row corresponds to one defined communication cycle. The consumer memories of all PEs have the same data values at



(a) During one master-clock one communication cycle is transmitted.

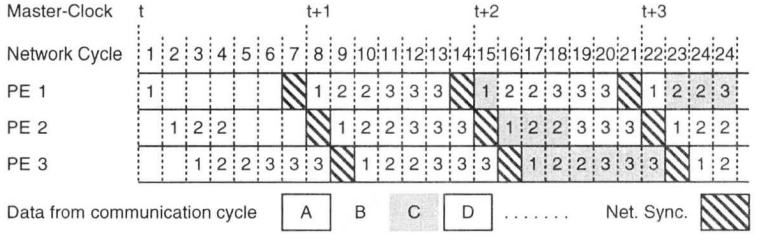


(b) The local consumer memory of each PE.

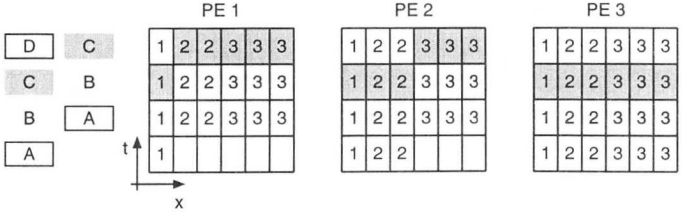
Figure 5.8: Non-Interleaved communication on MUSIC.

the same positions. The management for accessing the data is simple. This advantage is bought with a less usable bandwidth.

Interleaved Communication Interleaving means, that two successive communications cycles X and Y are mixed during one master-clock. Fig. 5.9(a) depicts the communication. After a filling the pipeline during master-clock t the full bandwidth is utilized (apart from the network synchronization time). However, during master-clock $(t+1)$ data from communication cycles A and B is transferred, during master-clock $(t+2)$ data from communication cycles B and C, etc. As a result, at each node parts from two successive communication cycles are transferred. A processor has to calculate data before it is communicated. When communication cycle B and C is transferred through the network, a



(a) During one master-clock two successive communication cycles are transmitted.



(b) The local consumer memory of each PE.

Figure 5.9: Interleaved communication on MUSIC.

PE can process new data from communication cycle A. The resulting data will then be part of communication cycle D. Thus, applying an interleaved communication requires to store two communication cycles at each processor. The latency through the entire system is prolonged. The local consumer memory of each PE is depicted in Fig. 5.9(b). Each row contains data from two successive communication cycles (except of the last node). The column, where communications X and Y switch in the memory is different on each PE and depends on the producer part of each PE.

The current implementation on MUSIC operates with a non-interleaved communication. This is done because of the following reasons:

- The complexity of the software is drastically reduced compared to an interleaved solution.
- The hardware controller becomes less complex.
- The lower efficiency is tolerable.

5.1.5. Synchronous Communication Controller

The synchronous communication controller (SCC) is configured once before the first communication cycle has started. Each processor writes four data values into the register file of its SCC (Fig. 5.10). The first

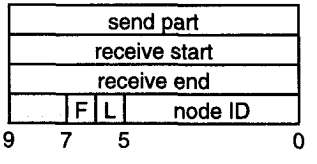


Figure 5.10: The configuration parameters of the communication controller.

value is an identification word. It is appended to each data value produced by this node. The SCC examines the ID and recognize its own data values after they have traveled once through the net. 6 bit contain a unique node ID. Two special bit are used to mark the first (F) and the last (L) node of the ring. The node, which has the F bit set, starts the communication always at the rising edge of the master-clock. The last node appends the L bit to each of its produced data values. The other nodes will then determine the end of the communication cycle after the last value with the L bit set has passed. The next two configuration words define the consumer (receiver) window. The start and end position, respectively, is specified. 10 bit are used for 512 audio channels. The last configuration word defines the producer (sender) window. Only the amount of channels to produce is indicated, since the start position of the producer window is given implicitly: During a communication cycle, every node begins to produce its data when the first empty channel arrives from the neighbor at the left.

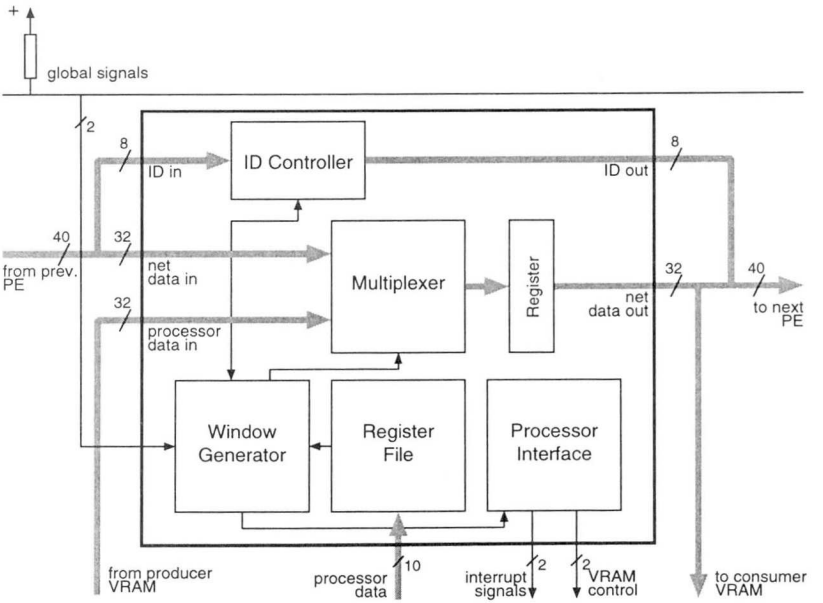


Figure 5.11: The functionality of the synchronous communication controller.

The architecture of the SCC, implemented in the FPGA on the MUSIC, is shown in Fig. 5.11. Its main functionality is a large data multiplexer. It decides, whether data values from the neighbor on the left are passed to the next node or if new values from the own PE are inserted into the data stream of the ring network. Two *global signals* are used by the network. One is the network clock running at 25MHz. Each SCC is transmitting data synchronously to this clock. The second global signal is the master-clock. Is has a clock rate of 48kHz and is generated by the audio interface board. Other clock rates, for example 44.1kHz or 38kHz, are also possible. The *register file* contains the configuration data for the controller, as described above. The *window generator* builds the heard of the controller. Based on the values in the register file, it determines, if the currently transmitted values are in the consumer window. In this case, a copy is then stored in the consumer VRAM. To arbitrate producing, the *ID controller* checks the F bit which indicates

the data block of the first node. Then it informs the window generator when the first invalid data word from the left neighbor appears. It means that all preceding nodes have transmitted their data values and this node can start to produce. The multiplexer is then switched and values from the PE are inserted in the data stream. After the last value of the own producer VRAM has been transmitted, the multiplexer is switched back. Again, it transmits data from the left neighbor until the communication cycle has finished. The ID controller checks each ID. Valid values are copied into the own VRAM. In the case where the ID is from the own node, an ID with value 0 is passed to next node, to indicate an invalid datum.

5.1.6. Memory Concept

The video memory of the PE is able to exchange data at high clock rates through the SAM. Additionally, the depth of the SAM is large enough to store all global audio channels. Therefore, with the end of a communication cycle a SAM-DRAM transfer is executed in the consumer memory and a copy of the entire SAM is placed into a specific row of the DRAM part. For the next cycle a new line is used. As a result, a cyclic ring-buffer appears in the consumer memory. It has a history of the last 512 communications (Fig. 5.12). Each line corresponds to

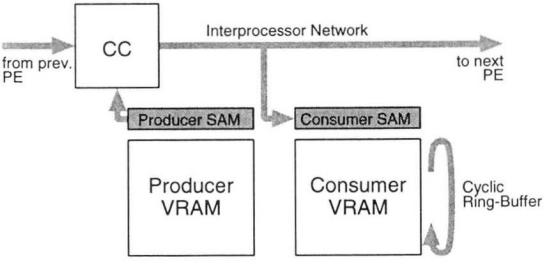


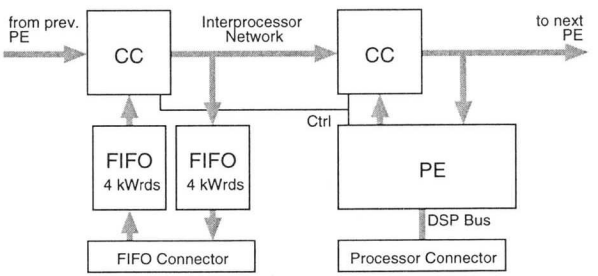
Figure 5.12: In the consumer memory the communicated channels are stored in a cyclic ring-buffer, where all data of the last 512 communications is accessible.

one master-clock, each row to one channel. Discussions about the practical usage of this memory concept for digital audio are described in

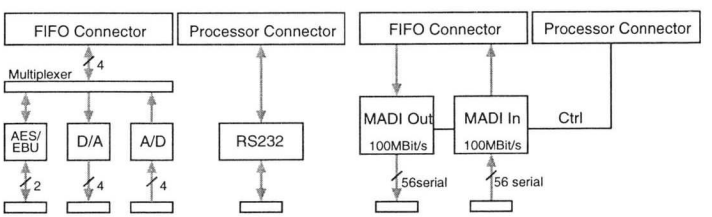
section 3.5.1 and 3.6.

5.1.7. Data I/O

The MUSIC system offers good data input and output capabilities. Beside the standard processor board, a separate I/O-board was designed which allows to connect peripheral devices directly to the interprocessor network [Ros94]. A separate SCC is controlling the data exchange with



(a) I/O-Board



(b) Quad Audio Interface (c) MADI Interface

Figure 5.13: The I/O-board of the MUSIC system provides a FIFO connector for direct access to the interprocessor network through a separate communication controller. The processor connector is used for initialization and control of peripheral devices.

the network. The architecture of the I/O-board is shown in Fig.5.13(a). Two 96 pole connectors supply the interface to the hardware extension

modules. To provide fluent data transfer 4 kWords of FIFO memory is implemented. It is located in both directions from and to the MUSIC network. The processor connector is used by the processor on the I/O-board for initialization and control of the extension module.

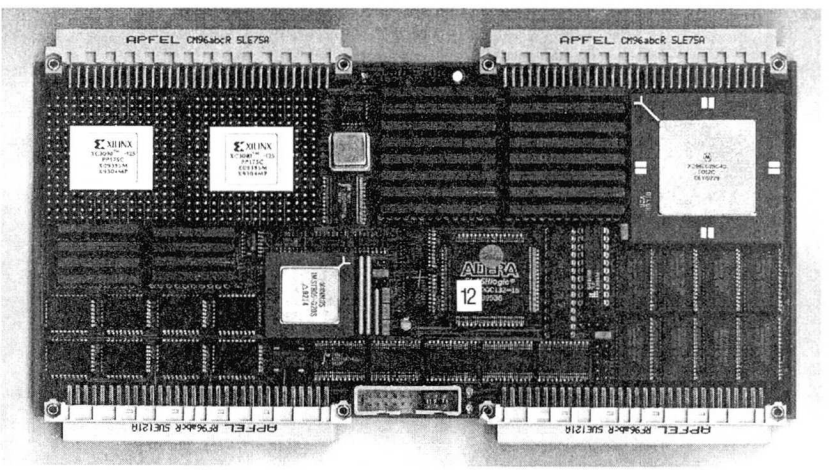


Figure 5.14: The MUSIC I/O-board. One standard PE is horizontally arranged, the board manager is located in the middle. On the left, a separate SCC and FIFO memory can be recognized. Two connectors at the bottom build the interface to an extension module.

The implementation of a mixing-engine on the MUSIC system required the design of two audio specific hardware modules. One is a quad audio interface (Fig.5.13(b)) [IK94]. It can be configured as a four channel analog audio interface. Two channels are also accessible as an AES/EBU digital interface[4]. Additionally, a serial interface (RS232) provides the possibility to transfer parameters from the control-desk to the MUSIC. The I/O-board processor collects parameters directly from the serial interface via the processor connector. The second hardware module is a multichannel audio digital interface (MADI) (Fig.5.13(c)) [RS94]. Using this standard, up to 56 digital audio channels are connected through a coaxial or fiber-optic link. The

[4]Two channel digital interface according to a standard of the Audio Engineering Society (AES) and the European Broadcasting Union (EBU)

implemented components are compatible to the FDDI standard and have a maximum throughput of 100Mbit/s. The two digital audio connection standards, AES/EBU and MADI, are introduced in section 2.6.

5.2. Software

The configuration software includes two main approaches:

- A *console definition* describes the functionality of a mixing console. Modules are chosen from a library and placed in a directed acyclic audio graph (DAAG, section 3.1.1). The resulting graph represents a full model for the entire mixing-engine. Only the qualitative operation of the modules is known. Their exact implementation is hidden.
- The objective of the *module implementation* is to create new or to replace existing modules. With a defined software interface the new modules are integrated in the module library and are used for a new console definition.

The two approaches are independent and dont influence each other. For example a module can be altered because of a some reason. If it is changed but is already used by a certain console definition a recompilation generates the code for the new mixing-engine. Fig. 5.15 depicts

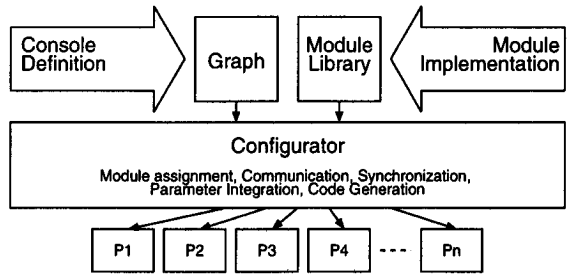


Figure 5.15: The console definition and the module implementation provide two approaches for the design of a mixing-engine. The configurator produces a code segment for every processor in the system.

the complete software architecture. The configurator builds the main

software part. All aspects of mapping a DAAG to the parallel system are integrated in this software block. As described in chapter 3, these elements are module assignment, communication and synchronization. Additionally, the communication of real-time parameters is integrated. The output of the configurator is a program code for every processor in the system.

This architecture offers a typical object-based approach. Modules appear as objects and are combined in the graph. Objects and graph together form a data-driven program and are used as input data for the configurator. Like in classical object-oriented languages, the objects itself hide their inner functionality. They are said to be encapsulated [Boo91]. The interface to each module is defined in such a way as to reveal as little as possible about its inner workings [CY91]. Consequently, the interface to an audio processing object is provided only by connecting audio channels and real-time parameters to the module (Fig. 5.16).

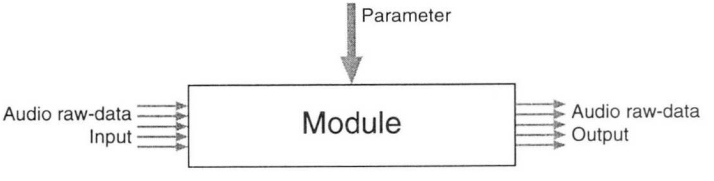


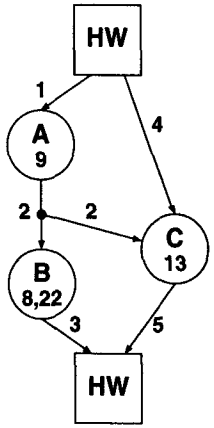
Figure 5.16: A module communicates only through audio and parameter channels.

In the next sections the three blocks of Fig. 5.15, console definition, module implementation and the configurator are examined.

5.2.1. Console Definition

The issue of this part is the specification of the DAAG. To supply a simple representation, the graph is described in plain ASCII text form. Each processing module defines its inputs and outputs. The links in the graph are represented by a unique identification number. Modules, that use a channel as input or output, add the representing number to their input or output list, respectively. A list of parameters which control the module concludes the description of one module. The specification of a

complete graph with n nodes is done in n lines.




```
A     (out: 2;    in:  1; par: 9);
B     (in:  2;    out: 3; par: 8, 22);
C     (in:  2,4;  out: 5; par: 13 );
$HW   (in:  3,5;  out: 1,4);
```

Figure 5.17: The representation of the DAAG is done in a definition file in plain ASCII text form. Each line describes the connections of one node in the graph.

As an example, the description of a graph with 3 nodes is demonstrated in Fig. 5.17. Any identification number for a channel can be chosen under the restriction that it is unique in the graph. A channel can be used as input more than once. But, only one module may produce this channel. Hardware devices are described the same way as processing modules. They are the only dedicated sources and sinks of the entire graph. Although only one line is used for their specification, they may appear twice in the graph, as input and output devices. Address handles of parameters, which control the real-time behaviour, are integrated for each module.

The syntax of the *console definition file* is described using the Extended Backus Naur Formalism (EBNF)[5]. The EBNF descriptions, used in this thesis, operate with the meta-symbols shown in Fig. 5.18.

Fig. 5.19 defines the syntax of the console definition file. The goal is to specify a graph with a minimal amount of information. Three types of *connectors* (`in`, `out`, `par`) can be attached to every module. Their

[5]This syntax description was introduced by Niklaus Wirth in his article: "What can we do about the unnecessary diversity of notation for syntactic definitions" [Wir77].

```
"..." : terminal symbols
[...] : option: occurs at most once
{...} : repetition: any number of times, including zero
(...) : grouping
|     : exclusive or
...   : increasing list
```

Figure 5.18: Meta-symbols of the Extended Backus Naur Formalism.

```
descriptor  = module "(" connector {";" connector} ")" ";"
module      = identifier | ("$" identifier)
connector   = con_type [list]
con_type    = "in" | "out" | "par"
list        = ":" number {"," number}
identifier  = letter { letter | digit }
number      = digit { digit }
letter      = "A" ... "Z" | "a" ... "z"
digit       = "0" ... "9"
```

Figure 5.19: The EBNF syntax for the console definition file.

order is of no importance. For in and out a list of identification numbers is indicated. With the connector par addresses of real-time parameters are defined. It is a list of address handles. Their values correspond with the address handles from the control-desk. A scale module, for example, is directly associated with a control device on the desk by this handle. The list size for each connector is defined in a separate *module definition file*. It is examined later. Hardware devices are handled the same way as processing modules. However, these nodes are marked with a '$'.

The definition file, as shown in Fig. 5.17, can be written directly using a text editor. To simplify this work, a graphical front-end software was designed. It allows to draw a DAAG on a workstation using a pointing device (mouse). In Fig. 5.20 a snapshot of this graphical user interface is shown. Modules are selected from a library and are placed as icons in a window. Connections between nodes are drawn with arrows. Hardware devices are symbolized by dedicated icons. The resulting definition file is then used by the configurator.

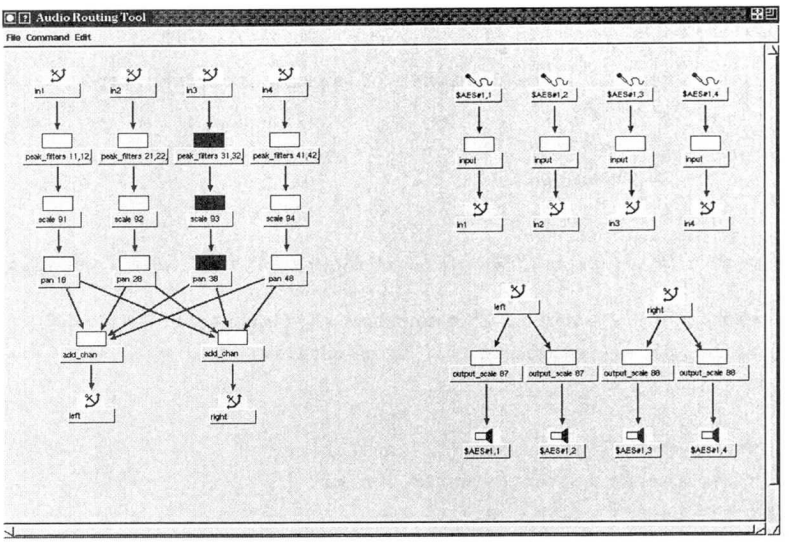


Figure 5.20: Instead of writing a console definition in plain text mode a graphical user interface on a workstation allows to draw the DAAG using a pointing device.

5.2.2. Module Implementation

During runtime a processing module should use as less processing cycles as possible. To save cycles it is inevitable to optimize especially that part of the module, which runs periodically to the master-clock. Therefore, each module is divided into three program parts: A setup, runtime and conversion code. Fig. 5.21 depicts the data-flow between these parts. The setup code is called once to prepare data for the runtime code. All information of the console definition file is prepared in that part. The runtime code is executed periodically to the master-clock. According to the setup-buffer it processes digital audio data. The conversion code is running on a separate processor where a preprocessing of real-time parameters is performed. It receives parameters directly from the control-desk and converts them to real-time parameters. A discussion about the three code segments follows.

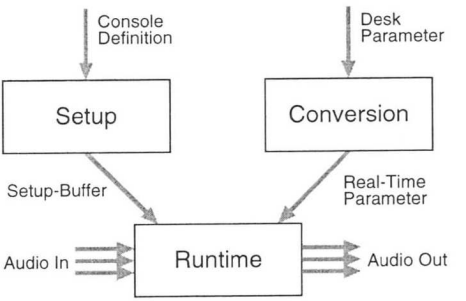


Figure 5.21: A processing module consists of a setup, runtime and conversion code. While the setup code is run once before startup, the runtime and conversion code are called periodically with a period-time (PT).

Setup code: A consequent register management is necessary when code segments are embedded in a hierarchical software structure. The standard method is to store the contents of registers on the memory stack before using them. It guarantees the correct behaviour of program parts that lie in a higher hierarchy level. After completion of a code segment, the used registers are restored by getting the old register values from the stack. This overhead becomes intolerable for small program parts which use many registers.

As a solution for this problem, a setup code is implemented for every module. It is called once after system startup. During that time, memory addresses of audio channels and parameters are stored in a setup-buffer. Addresses of filter tables, intermediate buffers and other module specific information are saved in the buffer, too. Instead of using a dynamic stack, like in a classical high level programming environment, the static setup-buffer is used. Every time the runtime code is called, it initializes registers directly from the setup-buffer. With this method, costly memory managements like register saving, stack handling, etc. can be totally omitted. As depicted in Fig. 5.22, the hierarchy of the entire software consists of two stages: A main program and the called modules. Every module is responsible for the initialization of registers. When a module is executed, it accesses its part of the setup-buffer

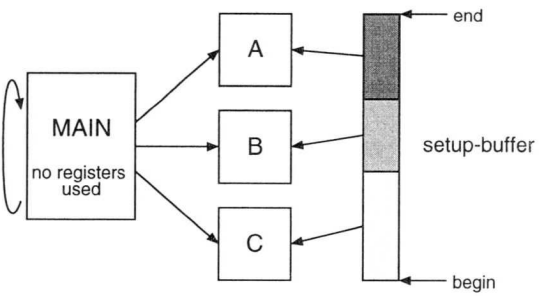


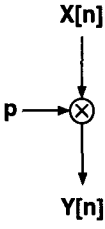
Figure 5.22: Modules initialize registers directly from a prepared setup-buffer. Register savings can be totally left out.

directly. The main program is repeated synchronously to the audio master-clock. It contains a list of modules that are called within one master-clock cycle. Since no registers are used in this code segment, their manipulation within modules does not affect the main program.

Runtime code: The functionality of a processing module is implemented in the runtime code. It is optimized and written in DSP machine language. Applying the prepared setup-buffer, all register savings are ignored. The runtime code fetches his data from the setup-buffer one by one using a global pointer and initializes the needed registers directly. The pointer is incremented each time a value is taken. After completion of the runtime code, the global pointer points to the beginning of the next segment in the setup-buffer and is used by the next module.

Consider a simple module which multiplies a signal with a gain factor p. The functionality and the corresponding program in machine language is depicted in Fig. 5.23. The overhead of data moving is still extensive. The functionality of the module is implemented in one processing cycle (line 6). Register initializations need five cycles, because each value has to be accessed indirectly. It means, an address register first has to be loaded which then allows to get the needed value. Due to the architecture of the MUSIC not more than one external memory access per cycle can be performed[6]. The complete module uses 8 pro-

[6]Only one external port of the DSP is connected to the memory.




```
1:move    y:(r5)+,n3       ;&X[n] -> n3
2:move    y:(r5)+,r2       ;&p -> r2
3:move    y:(r5)+,r4       ;&Y[n] -> r4
4:move    y:(r3+n3),d0.l   ;X[n] -> d0
5:move    y:(r2),d1.s      ;p -> d1
6:fmpy.s  d1,d0,d0         ;p*X[n] -> d0
7:move    d0.s,y:(r4)      ;Result -> Y[n]
8:rts
```

Figure 5.23: The runtime code of a scale module is shown. The initialization of all registers is performed using one global pointer (r5).

cessing cycles on a Motorola DSP 96002. The module completes within 400ns, if one cycle is used per memory access and the DSP is running at 40MHz.

Conversion code: The desk, connected to the MUSIC, transmits real-time parameters in a linear range $0 \leq x \leq MAX$ into an input-buffer of size n. A dedicated processor performs the conversion from the input-buffer into a runtime-buffer of size m. Each module can have

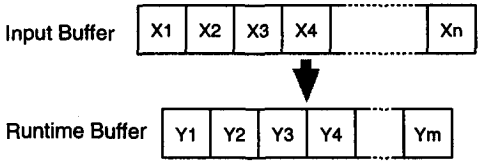


Figure 5.24: The conversion code transforms linear parameters from an input buffer into a runtime buffer. A module specific algorithm is applied.

its own conversion algorithm between a linear parameter x and a corresponding parameter y used by the runtime code. For example, a scale module needs an exponential conversion to take the perception of the human ear into account. The size of input values n is not necessarily equal to the size of runtime values m. Filter changes, for example, depend on several input values (cutoff frequency, gain, Q). The corresponding runtime value, however, is only one absolute entry of a filter-table. The parameter conversion is performed each time an input

value changes and it works as an asynchronous task on the runtime-buffer (see section 3.6.1). The concept for communicating the runtime buffer through the interprocessor network is described in section 3.6. It is transferred periodically and multiplexed to one or several global audio channels. Given from the size of the consumer memory, 512 values are transmitted using one audio channel. A refresh frequency of 10ms results for a sampling frequency of 48kHz.

Module definition: Corresponding to the console definition file, described in section 5.2.1, a separate *module definition file* exists. Information about processing requirements, number of connectable channels and type and order of parameters is stored in this file. It is used by the configurator. The syntax of the module definition file is similar to the console definition file. It is described in Fig. 5.25 using the Extended Backus Naur Formalism (EBNF).

```
descriptor  = module "(" connector {";" connector} ")"
              "[" cycles "]" ";"
module      = identifier | ("$" identifier)
connector   = (con_type [list_size]) | con_num | "SYNC"
list_size   = ":" (number | "[]")
cycles      = (number ["+" con_type "*" number]) | node_number
con_type    = "IN"      | "OUT"     | "PAR"
con_num     = "NUM_IN"  | "NUM_OUT" | "NUM_PAR"
node_number = "n" + number
identifier  = letter { letter | digit }
number      = digit { digit }
letter      = "A" ... "Z" | "a" ... "z"
digit       = "0" ... "9"
```

Figure 5.25: The EBNF syntax for the module definition file.

The following information is encoded in this file:

- The order of each `connector` (`IN`, `OUT`, `PAR`) is defined, when the module is called from the main program. This way no fix order is required in the console definition file.
- The `list_size` specifies the amount of channels or parameters at

each `connector`. Open lists are marked with '`[]`'. In this case, the module allows to connect an undefined amount of channels or parameters.

- The number of connected channels or parameters can be indicated by `con_num (NUM_IN, NUM_OUT, NUM_PAR)`. It is necessary for modules which have an open `list_size`.

- The exact number of processing `cycles` is given. If the number of cycles depend on the number of a specific connector type, a multiplier is added (`con_type "*" number`).

- In case of a hardware module the exact position of this node in the ring bus must be known. Since no processing cycles have to be indicated for hardware modules the place is used to define the `node_number`.

- With `SYNC` the module gets a synchronization list. For each input channel a number defines the relative distance to that input channel, which has the longest signal path from the source until now. It is equivalent to the size of an intermediate FIFO buffer for each input channel (see section 3.5).

Consider the example discussed in the previous section and shown in Fig. 5.17. The corresponding module definition file is represented in Fig. 5.26. Except of module `C`, all modules require exactly one input

```
A     (IN,    OUT, PAR)                 [12];
B     (IN,    OUT, NUM_PAR, PAR:[])     [27+ PAR * 11];
C     (IN:2, OUT, PAR, SYNC)            [32];
$HW   (IN:2, OUT:2)                     [n0];
```

Figure 5.26: The module definition file of the example, shown in Fig. 5.17.

and one output channel. Module `B` allows an open list of parameters. As an example, the amount of filters in an equalizer module can be specified by the number of parameters. The amount of processing cycles is depending on the amount of parameters. Variable processing times

are possible this way. The hardware device `HW` is located at position 0 and requires two inputs and outputs, respectively. Because module `C` has two inputs, the configurator will insert a synchronization scheme (delay list) for each input at the position `SYNC`.

With the described module definition file, the configurator can perform a brief error check. Too many connected channels or parameters at a specific module are recognized.

5.2.3. Configurator

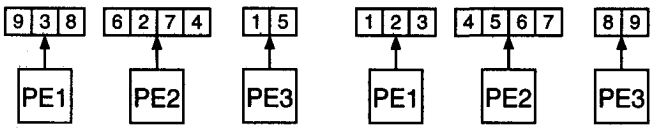
To make use of the connection-oriented model of the DAAG, the configurator is written in *C++*. It is compiled using the *g++* compiler on a SUN workstation in order to keep the program as portable as possible. An object class is defined where all necessary information of processing modules is stored. Amount and identification numbers of input and output channels are collected from the console definition file. The remaining information like time requirements and rules for calling the module is fetched from the module definition file and is also integrated in the object class. According to the configuration concept, described in chapter 3, the following steps are then performed: Modules are assigned using the FFD bin-packing heuristic, communication channels of the interprocessor network are allocated and a synchronization scheme is generated for modules, that use more than one input channel. The steps are discussed in detail in the following paragraphs.

Module Assignment: The FFD algorithm, discussed in section 3.2, is implemented for module assignment. Therefore, modules are grouped in a list with decreasing order of computing times. Then, always beginning with the first processor, each module is allocated to next processor, which still has enough processing capacity for this module. A new processor is only introduced if the current module did not fit onto one of the already tested processors.

Communication: First, a local connection is established for modules that run on the same processor and dont need a global connection. Modules with equal computing time can still be exchanged among processors to find as many local connections as possible. Then, it is checked

if there are enough global audio channels available for the remaining connections in the graph. If not, the *optimize* procedure is called which clusters small modules to larger units that have local connections. Otherwise, every remaining connection of the graph is set to one global audio channel.

Since the interprocessor network of the MUSIC allows only one producer window and one consumer window for each node, every PE has to produce one coherent data block of ordered audio channels. Every processing module has to produce the next global channels behind the channels of the preceding processing module. Therefore, after having assigned all modules to PEs, a remapping has to be done where the identification numbers of the used channels have to be reallocated in increasing order. Beginning with the first processor a renumbering of the produced channels is performed to get to a strongly increasing order (Fig. 5.27). To correspond with the new identification numbers of the



(a) Before reordering (b) After reordering

Figure 5.27: Channels are reordered to get strictly increasing channel numbers for the complete network.

output channels, the input channels of every module are set to the new values, too. All identifications numbers in the network are still unique. Every node produces exactly one coherent data window and consumes all transmitted channels.

Synchronization: Different path lengths in the DAAG are synchronized. Therefore, the entire graph is analyzed. It is necessary to know the exact distance of every node from the source nodes, the hardware nodes, of the graph. It is assumed that the incoming audio data is synchronous at all hardware connections. The entire graph is sought from the source nodes to the output nodes. If the search algorithm meets a

junction node, an intermediate synchronization is made for this node. A synchronization scheme is inserted and the distance of this junction node is set to the longest distance of all connected paths. Reaching the output nodes, all nodes are set to the same distance, the longest distance found in the graph.

Code Generation: The configurator produces a C-program for every PE in the system. It may be surprising to generate a high level C-code instead of the DSP machine language, since a program written in C on a DSP loses approximately a factor of 3 to 4 in the performance. However, it will be shown that for this special case the performance of a C-program is equivalent to the machine code and even supports better debugging facilities. Corresponding to the three code segments of each module, the generated program is divided into three parts: A setup code, a periodic runtime code for each processor and a conversion code, running on a separate processor.

Again, consider the example shown in Fig. 5.17. The setup code consists of two procedures and is shown in Fig. 5.28. One is the initialization code for the hardware module (`HW`). It is executed once by the processor on the MUSIC I/O-board where the hardware device is connected to. Later, during runtime, the hardware module runs autonomously. As required in the module definition file, shown in Fig. 5.26, `HW` is allocated at node 0. The three processing modules (`A, B, C`) are assigned to the processor at node 1. The identification numbers of the output channels are renumbered in increasing order, following the output channels of the previous node. The address handles of parameters are directly inserted as absolute memory addresses of the DSP. Module `B` also requires the number of parameters. It is calculated and inserted by the configurator. The synchronization scheme for module `C` is equivalent to the length of the FIFO buffer, which has to be implemented for each input channel. Channel 1 (channel 4 before renumbering), comming from the module `HW`, is stored in a FIFO buffer of length 1.

A runtime procedure exists for each processor (Fig. 5.29). It is an endless loop. Every module is called in the same order as in the setup code. Compiled to the DSP machine language this code segment is simply a list of jumps, where subroutines are called. No data regis-

```
void setup_HW()                          /* Node: 0 */
{
  HW(/*in*/                2,3,
     /*out*/               0,1);
}

void setup_code_0()                      /* Node: 1 Load: 93 */
{
  init_audio_blocks();
  init_B(/*in*/            4,
         /*out*/           2,
         /*num_par*/       2,
         /*par 8,22*/      269488133,  /* direct address */
                           269495301); /* direct address */
  init_C(/*in*/            4,1,
         /*out*/           3,
         /*par 13*/        269490693,  /* direct address */
         /*SYNC*/          0,1);
  init_A(/*in*/            0,
         /*out*/           4,
         /*par 9*/         269488645); /* direct address */
}
```

Figure 5.28: The setup code for the example, shown in Fig. 5.17.

```
void runtime_code_0()
{
  for (;;) {
    B();
    C();
    A();
    Sync_with_master_clock();
  }
}
```

Figure 5.29: The runtime code for the example, shown in Fig. 5.17.

ters are used and therefore register savings can be left out. At the end of one iteration, a synchronization with the master-clock is performed (`Sync_with_master_clock()`). The modules initialize their registers according to the prepared setup-buffer. The runtime code is the only program segment generated by the configurator which is time critical. Every cycle of this code lowers the processor capacity. Therefore it is interesting to compare the compiled runtime code with the same code written in machine language. Fig. 5.30 shows the compiled runtime code. The compiler has produced a list of "jumps to subroutines" (`jsr`), which cannot be further optimized. The implemented C-code has therefore the same performance as a handwritten version in machine language.

```
Fruntime_code_0
L28
    jsr     FB
    jsr     FC
    jsr     FA
    jsr     FSync_with_master_clock
    jmp     L28
```

Figure 5.30: The translation of the runtime code into machine language, generated by the C-compiler.

The processing requirements of the runtime code can be integrated into each module. One extra instruction, which necessary to call the module (`jsr`), is added. No extra processing calculation is then needed for the runtime code.

There is one possibilities to optimize the runtime code. One large program could be generated by combining processing modules automatically. Instead of the jump list (Fig. 5.29) one large machine code consisting of the assigned modules would result. This optimization is not implemented on the MUSIC.

The third segment is the conversion code. It runs on an extra processor where all parameters of the entire system are preprocessed. This segment is executed whenever parameters are changed.

```
void conversion_code()
{
  convert_B(/*num_par*/    2,
            /*par*/        8,22);
  convert_C(/*par*/        13);
  convert_A(/*par*/        9);
}
```

Figure 5.31: The conversion code for the example, shown in Fig. 5.17.

5.3. Summary

The implementation of a mixing-engine is shown. The hardware platform MUSIC has been modified and adapted for audio processing. The new communication concept allows the transmission of up to 512 digital audio channels within the system. It is shown, that the ring bus architecture of MUSIC offers on one hand a scalable system with maximum of 63 connected nodes and comfortable I/O capabilities. On the other hand, disadvantages when applying a ring bus in digital audio are discussed. They arise in terms of less usable bandwidth (empty channels) compared to a common bus.

The implemented software covers the configuration concept as described in chapter 3. Two approaches allow a reconfiguration: First, the optimized programming of digital signal processors is provided. The encapsulation of processing modules permits functional modifications and the integration in existing systems. Second, the high level definition of a mixing-engine is supplied by drawing the corresponding DAAG. A compilation of these inputs generates the code for a new mixing-engine.

Chapter 6

System Limitations

6.1. Configurations on MUSIC

The question arises, what are the possible configurations on MUSIC? In general, the number of processors and their usable performance limit the processing demands of a configuration while the bandwidth of the interprocessor network confines the communication requirements. The most important measure for the size of a configuration is the number of implementable strips. It is difficult to quantify the processing demands of a strip. A typical implementation is given in Tab. 2.2. But, since any reconfiguration is allowed, substantial differences may occur. However, this strip is taken for a raw estimation of the performance of MUSIC when used as a mixing-engine.

The number of summing busses, more precisely, the number of summing busses a strip may be connected to influences the size of a configuration. This parameter defines the dimension of the mixing matrix (Fig. 2.2) and therefore most communication needs. Industrial implementations perform a clustering of summing busses. This way, a strip can only be connected to some clusters of summing busses and the extent of communication can be regulated. However, for the following calculations always a fully connected mixing matrix is assumed and the number of summing busses is used as a variable.

Several system constraints are given by hardware components and processing modules :

p_{max}: Maximal number of processor

f_p: Processor frequency

f_s: Audio sampling frequency

f_c: Communication frequency

cpi: Clocks per instruction

i_s: Instructions per strip

i_b: Additional instructions per summing bus for each strip

g: Grain size of processing modules

l_m: Number of memory locations for one communication

t_{sync}: Synchronization time communication/processing

c_{sync}: Number of global channels for synchronizing communication/processing

An additional parameter can be seen as a constant for the special case when communication costs are ignored. It is the processor usage factor.

Usage factor of processor. The usage factor $u_p(g)$ is depending on the grain size g of the used processing modules.

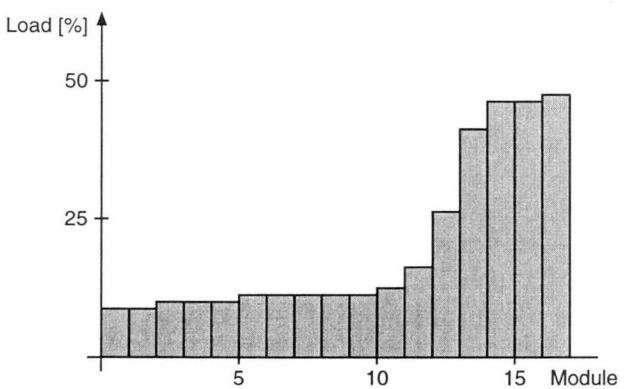
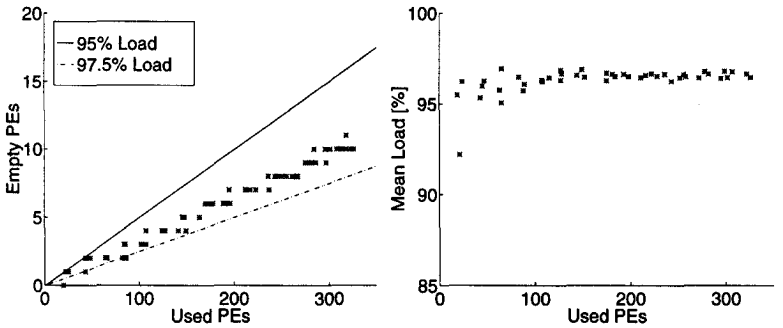


Figure 6.1: On the MUSIC, 17 modules are implemented. The percentage of their processing requirements shown.

Fig. 6.1 shows the processing demands or the grain size, respectively, of the different modules on MUSIC. A base library of 17 modules is implemented. All modules use less than 50 percent of processing cycles that can be performed during one audio sampling period. It is required



(a) Configuration prediction (b) Mean Load of PEs

Figure 6.2: Simulations of random configurations using the implemented modules on MUSIC show a linearly increasing loss of PEs. Fig. 6.2(b) depicts the mean load. 45 simulations are shown, between 100 and 1500 modules are assigned.

for good results when FFD bin-packing is applied (section 3.3.3). Three quarter of the modules need less than 20 percent of processing cycles.

Performing random assignments with FFD bin-packing and applying the module library leads to results as shown in Fig. 6.2. The loss of PEs grows linearly, as stated in section 3.3.3. The average load of PEs is in the range between 92% and 98% for all performed simulations (Fig. 6.2(b)). Using more processors the scattering is smaller and the expected load is in the neighborhood of a constant. In this implementation, it is 96.5% and the resulting usage factor is therefore set to $u_p(g) = 0.965$. If the grain size of modules remains constant, we can write $u_p(g) = u_p^* = constant$.

Maximal number of strips. Let the maximal number of strips S_{max} be the measure for the size of a configuration. $S_{max}(b, g)$ can be written as a function of summing busses b and grain size of processing modules g

$$S_{max}(b,g) = \frac{f_p}{f_s} * \frac{p_{max}}{cpi(i_s + i_b b)} * u_p(g) \tag{6.1}$$

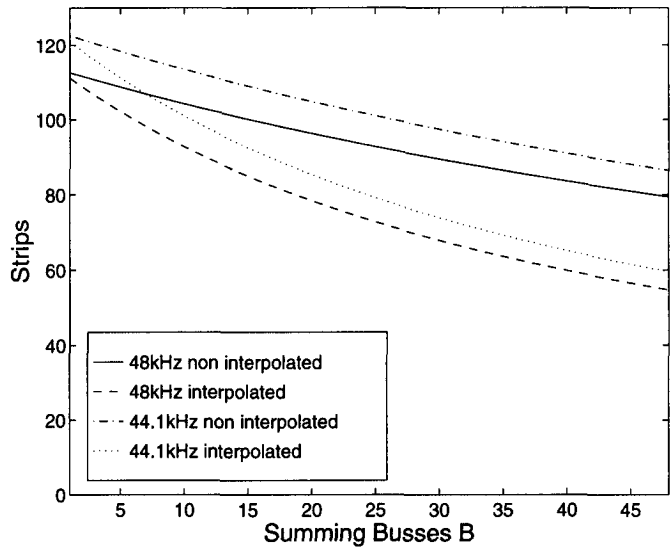


Figure 6.3: The maximal number of strips $S^*_{max}(b)$ on the MUSIC with 63 processors as a function of summing busses b. With and without interpolation refers to a parameter interpolation at each connection point in the mixing matrix.

To compute only the processing performance it is assumed, that all communication requirements can be satisfied by the interprocessor network. Consequently, modules do not need to be grouped and the grain size remains constant. Under this condition the usage factor is set to $u_p(g) = u^*_p = constant$ and

$$S^*_{max}(b) = \frac{f_p}{f_s} * \frac{p_{max}}{cpi(i_s + i_b b)} * u^*_p \tag{6.2}$$

The DSP-96002, used in the MUSIC system, is running at $f_p = 40\text{MHz}$, has a *cpi* of 2, and allows a maximum of $p_{max} = 63$ processors. The processing requirements for one typical strip are $i_s = 223$ cycles (Tab. 2.2). Using interpolation within the matrix has a major influence, because the amount of processing cycles at each connection point varies. The demands for the summing matrix are at least $i_b = 2$ instructions per connection (section 3.5.1), when no parameter interpolation is prac-

ticed. Applying a first order low-pass interpolation increases the cost to $i_b = 5$ instructions per connection (section 4.5.1). Combining these constraints leads to some maximal configurations on MUSIC. As an example, if 16 summing busses are implemented and the summing matrix is fully connected, extra 32 processing cycles per strip are necessary (without interpolation). Taking the usage factor u_p^* of 0.965 for one processor, as calculated above, a maximum of 99 strips on MUSIC results at $f_s = 48$kHz. With Eq. 6.2 the maximal number of strips is depicted in Fig. 6.3. $S_{max}^*(b)$ is shown for the sampling frequencies 48kHz and 44.1kHz. Note, that for these calculations the communication costs are not taken into account.

Maximal number of global audio channels. To evaluate the usable communication bandwidth the maximal number of global audio channels C_{max} is introduced. The communication frequency is $f_c = 25$MHz. A theoretical maximum of 520 global channels results at 48kHz sampling clock. A consumer system with 44.1kHz sampling clock has 566 channels. But, the serial access memory of the MUSIC processing element is able to store only $l_m = 512$ channels during one sampling clock (section 5.1.1). For a small number of PEs l_m limits C_{max}. Using more PEs another factor is responsible for C_{max}: Since non-interleaved communication is used in the MUSIC implementation (section 5.1.4), not the full bandwidth is utilized (Fig. 6.4 and Fig. 5.8(a) on page 87). A full system can combine $p = 63$ PEs in a VME sized cabinet. For such

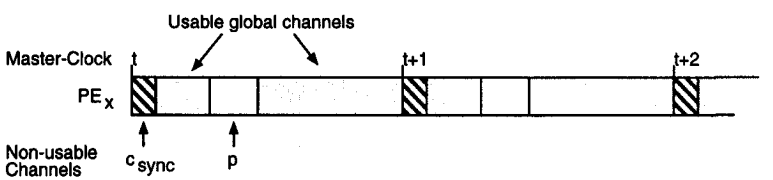


Figure 6.4: Non-interleaved communication on MUSIC. c_{sync} and p nonusable audio channels appear during synchronization and because of non-interleaved communication.

a system 63 global channels are lost. Additionally, the synchronization t_{sync} has to taken into account. It is the time used for a SAM/DRAM

transfer in the video memory. This time can be expressed in a number of global audio channels c_{sync}, which can not be used. It is evaluated by

$$c_{sync} = 2 * t_{sync} * f_c \tag{6.3}$$

and depends on the communication frequency f_c of the network. The factor 2 appears, since t_{sync} is needed for both, producer and consumer memory. The used video memories of have $t_{sync} = 200$ns, which results in $c_{sync} = 10$. C_{max} is then evaluated by

$$C_{max}(p) = \begin{cases} l_m & : \quad p + c_{sync} \leq \frac{f_c}{f_s} - l_m \\ \frac{f_c}{f_s} - (p + c_{sync}) & : \quad p + c_{sync} > \frac{f_c}{f_s} - l_m \end{cases} \tag{6.4}$$

For a full MUSIC system 447 global channels are usable at 48kHz and 493 at 44.1kHz sampling clock.

Minimal number of global audio channels. The minimal number of audio channels C_{min} is reached when only the mixing matrix is communicated globally and all strips perform their inner communication locally on the processors. It is assumed, that every channel which goes into the mixing matrix is broadcast on the interprocessor network and occupies a global channel. Additionally, each strip uses at least one global channel as input. Input and output strips use a minimum of two

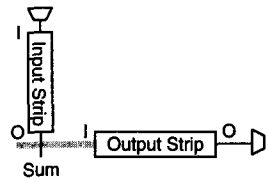


Figure 6.5: Minimum global communication requires one input and one output per strip.

global audio channels (Fig 6.5) and C_{min} is simply

$$C_{min}(b, g) = 2 * S_{max}(b, g) \tag{6.5}$$

It depends on the number of summing busses and the grain size of the distributed modules. It indicates an absolute lower bound of communication requirements.

Predicting the effectively used global channels for any configuration is not possible. It depends on the complexity of the signal-flow graph. Maximal and minimal amount of global channels, as calculated above, are upper and lower bounds. $C_{max}(p)$ depends on the number of processors and $C_{min}(b, g)$ on the number of summing busses and the grain size of the processing modules. If the maximal number of global channels is not enough to implement a specific graph, modules are grouped to save bandwidth (section 3.2). As a result, instead of increasing the network bandwidth, the processor load is decreased. Consequently a higher grain size g and a lower usage factor $u_p(g)$ results. How much $u_p(g)$ declines can not be evaluated in advance.

What is the limiting factor for a large mixing console on MUSIC, the communication bandwidth or the amount of processors? To perform an estimation on this question, the amount of additional global channels per strip can be computed.

Additional global channels per strip. This value is denoted by C_S and indicates how many global channels per strip remain when the minimal amount of global channels are occupied. C_S is used as a criterion, how fine the granularity of a strip can be.

$$C_S = \frac{C_{max} - C_{min}(b, g)}{S_{max}(b, g)} \tag{6.6}$$

It is a theoretical value to indicate the amount of remaining global channels per strip when the minimal amount of global channels is used. Suppose that the usage factor does not decline and has a constant value of u_p^*, we can combine Eq. 6.2 and Eq. 6.5 and get

$$C_S^* = \frac{C_{max}}{S_{max}^*(b)} - 2 \tag{6.7}$$

Tab. 6.1 shows the maximal configurations for 8, 16, 24, and 32 summing busses. C_S^* signifies that an average of more than 2 global channels is left per strip under the condition that only the minimal number of global channels are occupied and the usage factor is a constant u_p^*. It implies that each strip can be divided at least into two pieces even when all processing resources are used. Thus, it must be supposed that the processing resources and not the communication bandwidth are the

b	8	16	24	32
$S^*_{max}(b)$	105	99	93	88
C^*_S	2.25	2.52	2.80	3.08

Table 6.1: Maximal configurations on a full MUSIC at 48kHz sampling rate. The maximal number of strips $S^*_{max}(b)$ and the additional global channels per strip C^*_S are computed with $C_{max} = 447$ global audio channels and a constant usage factor u^*_p of 0.965.

limiting factor. Nevertheless, constructing a configuration, where the communication bandwidth is the restricting factor is always possible.

6.2. Comparison with available products

Tab. 6.2 shows a comparison of commercially available digital mixing consoles with the mixing-engine, implemented on MUSIC[1]. Of course, a comparison is restricted for signal processing and configuration capabilities, only. The number of strips is comparable for all products. A size configuration with different numbers of strips is possible in most implementations manually. Data about the time for such a configuration is not available. In contrast to such manual configuration capabilities, the automatic configuration on MUSIC takes less than 10 minutes. Moreover, the MUSIC implementation is the only one, which allows a functionality configuration like changing, implementing or removing processing modules.

[1] Data collected from product brochures and personal conversations

	Neve Capricorn	SSL Axiom	Sony OXF-R3	Lawo MC-80	StageTec Cantus	Studer D940	ETH MUSIC
Strips	32..96	32..96	48	32..96	32..96	32..96	6..100
DSP	ASIC	ASIC	DEC Alpha	M96002	AD21060	M56002	M96002
Processing Type	Fix	Fix	Float	Float	Float	Fix	Float
Size Configuration	man	man	no	man	man	man	auto
Configuration Time	?	?	-	?	?	?	< 10 min
Functionality Configuration	no	no	no	no	no	no	yes
Price [1000 US$]	300-700	400-1100	800	300-1100	400-800	400-1000	-

Table 6.2: Comparison of commercially available mixing consoles with the mixing-engine implementation on MUSIC.

Chapter 7

Conclusion

7.1. Results

The division of a digital audio mixing console into mixing-engine and control-desk allows to concentrate on each part separately. This work has investigated methods for implementing a mixing-engine on a parallel processor architecture. It has been confirmed that a mixing-engine can be implemented separately on a parallel computer system. Since high demands in terms of flexibility and cost optimization exist for such a processing system, a concept for an automatic configuration is proposed. It allows to compile an audio mixing-engine in a high-level environment. This way, profitable and user specific designs are attained. Problems dealing with parallel programming are completely hidden and the designer can concentrate on architectural issues. The implementation time for a new design can be kept extremely short which again reduces cost in the industrial production process.

Model A new model for a general description of an audio mixing console is presented (section 3.1.1). It is called a directed acyclic audio graph (DAAG) and allows to characterize multichannel audio processing in a graph, where processing modules are combined in a network of audio channels.

The parallel computer architecture fulfills two conditions: It is scalable in the number of processors and has an interprocessor network with broadcast capabilities. Such an architecture, which provides both features, is called orthogonal (section 3.1.4). The flexibility of connecting only the needed amount of processors and the communication requirements of the mixing matrix are combined this way.

Graph Mapping Mapping a DAAG onto an orthogonal architecture in order to minimize the total system cost becomes the main issue. This assignment problem is known to be NP-complete and can not be solved within polynomial computing time. But, on one side the typical signal-flow in an audio mixing-engine, which allows a raw separation into processing and communication, and on the other side the periodicity of digital audio helps to split the problem of the optimal task to processor assignment into three distinct problems: Module assignment, communication and synchronization.

To perform an optimal module assignment, heuristics of the bin-packing problem are investigated for their usage in the field of digital audio (section 3.3.2). Applying FFD bin-packing is shown to be a suitable assignment algorithm. The used processor capacity is predictable, if a small library of processing modules is used. The implemented library on MUSIC reaches 96.5% expected processor load (section 6.1). The synchronization of different path lengths in the graph and the communication of real-time parameters is performed using a new memory concept (section 3.5).

Algorithms For the interpolations of real-time parameters, efficient algorithms were developed (section 4.5). All possible artifacts, which occur in a digital mixing-engine, are eliminated.

Hardware Implementation The implementation of the presented concept on the MUSIC proves its efficiency. The hardware of the MUSIC was adapted to fulfill the requirements of orthogonality and digital audio data exchange. A new synchronous interprocessor network was realized (section 5.1.3). An I/O-board for MUSIC and two digital audio interfaces were designed (section 5.1.7).

Software Implementation The developed software allows the automatic configuration of a mixing-engine on MUSIC. A mixing-engine is created easily by specifying its definition graph. A graphical user interface was programmed for this purpose (section 5.2.1). Since algorithms are encapsulated in modules, their implementation and the definition of the graph are isolated and can be performed independently. New

or existing algorithms are programmed or modified in minimal time. These features hardly incur an overhead in terms of processing power. A syntax for the graph description (section 5.2.1) and for the module description (section 5.2.2) is presented. It is used as an interface to the configurator, the main software part. It performs the assignment and generates program codes for each processor (section 5.2.3).

With the presented concept and its realization in a prototype, it has been shown, that an automatic configuration of a mixing-engine is feasible and allows a fast adaption of digital mixing consoles. It optimizes the costs of each system and reduces costs in the production process. As a result, this work is the basis for an industrial product. Such a product is currently in development.

7.2. Outlook

The presented concept is intended to be used in a industrial environment for custom specific system designs. It helps the producer of a mixing console to offer cost optimized and individual systems for the professional user. However, the flexibility and the short time of a reconfiguration opens the opportunity to use this concept not only for a factory configuration. A new scenario is conceivable where a digital audio mixing console is configured just before a new recording session starts. This means, the console is composed or adapted every time before it is used. Only two reasons are given to show, why this consideration is innovative:

- Some recording or broadcast sessions demand special processing. This can be an unusual filter characteristic or an additional reverberation module. If it is practicable to specialize a mixing console only for such a session, new possibilities of audio signal processing are introduced and music can be made more attractive.

- The hardware cost of a mixing console is high. It is necessary to utilize the system efficiently. Audio engineers, who operate with the system, mostly have knowledge over one type of mixing console. Operating with a different kind of mixing console introduces

costly learning procedures. If the possibility exists to configure the console exactly to that type, which the operator knows, its utilization is optimized and costs can be reduced.

As a result, this work is probably only one step towards a new area in audio recording studios. The hardware of digital audio processing systems will be more and more transparent for an adaption to individual needs. Apart from the factor cost, flexibility will be the important element.

Apart from digital audio processing, fast reconfiguration procedures are also required for a large variety of other systems. For example automatic production processes, telephone switchboards, base stations for mobile communication, computer network routers, etc. have to be configured before being installed. This procedure requires high demands in terms of time and manpower. Similar concepts, as described in this work, are feasible and help to build such systems more economically.

Bibliography

[AES91] AES Recommended Practice for Professional Digital Audio Engineering - Serial Multichannel Audio Digital Interface (MADI). *AES technical paper (ANSI S4.43-1991)*, AES10, 1991.

[AES92] AES Recommended Practice for Professional Digital Audio Engineering - Serial Transmission Format for two-channel Linearly Represented Digital Audio. *AES technical paper (ANSI S4.40-1992)*, AES3, 1992.

[AHU83] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[AMD92] AMD Datasheet. *TAXIchip Intergrated Circuits Am7968/Am7969*, 1992.

[And91] G. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood, CA, 1991.

[Ant93] Andreas Antoniou. *Digital Filters, Analysis, Design and Applications.* McGraw-Hill, 1993.

[AO93] G. Andrews and R. Olsson. *The SR Programming Language.* Benjamin/Cummings, Redwood, CA, 1993.

[Bar84] Earl R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM Journal of Algebraic Discrete Methods*, 3(4):541–550, December 1984.

[Ben88] K. Blair Benson. *Audio Engineering Handbook.* McGraw-Hill, New York, 1988.

[BJ93] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *6th SIAM conf. of Parallel processing for scientific computing*, pages 445–452, 1993.

[BJL+84] J. Bentley, D. S. Johnson, F. Leighton, C. McGeoch, and L. McGeoch. Some Unexpected Expected Behavior Results for Bin-Packing. In *Proceedings 16th Ann. ACM Symp. on Theory of Computing*, pages 279–288. Association for Computing Machinery, 1984.

[BJLM83] J. Bentley, D. Johnson, F. Leighton, and C. McGeoch. An Experimemtal Study of Bin-Packing. In *Proceedings 21st Ann. Allerton Conference on Communication, Control and Computing*, pages 51–60, Urbana, Ill, 1983. University of Illinois.

[BKG95] B. Bäumle, P. Kohler, and A. Gunzinger. Interactive Parallel Rendering on a Multiprocessor System with Intelligent Communication Controllers. In José Fortes, Edward Lee, and Teresa Meng, editors, *Proceedings of the Parallel Rendering Symposium (PRS'95)*, October 30–31 1995.

[Ble78] B. Blesser. Digitization of audio: a comprehensive examination of theory, implementation and current practice. *Journal of Audio Eng. Soc. (AES)*, 26:739, October 1978.

[BLOS95] A. Burchard, J. Lieberherr, Y. Oh, and S. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, Dec. 1995.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

[Boo94] Grady Booch. *Object-Oriented Design with Applications, 2nd Ed.* Benjamin/Cummings, 1994.

[BS93] Stephen T. Bernard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructered problems. In *Sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[CGJ84] E. G. Coffman, M. R. Garey, and D. S. Johnson. *Algorithm design for computer system design*, chapter Approximation Algorithms for Bin–Packing – An Updated Survey, pages 49–106. Springer, New York, Wien, 1984.

[Che96] W. Chen. Performance of cascade and parallel IIR filters. *Journal of Audio Eng. Soc. (AES)*, 44:148–158, 1996.

[CL91] E. G. Coffman, Jr. and G. S. Lueker. *An Introduction to the Probabilistic Analysis of Packing and Partitioning Algorithms.* Wiley & Sons, New York, 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17, 1985.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice-Hall, 1991.

[ERA95] Hesham El-Rewini and Hesham H. Ali. Task scheduling in multiprocessor systems. *IEEE Computer*, pages 27–37, Dec. 1995.

[ERLA94] H. El-Rewini, T. Lewis, and H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, Englewood Cliffs, N.J., 1994.

[Fal94] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. Technical report, Brussels, 1994.

[FD92] E. Falkenaur and A. Delchambre. A genetic algorithm for bin packing and line balancing. In *IEEE Robotics and Automation*, pages 1186–1192, France, May 1992.

[FDD92] FDDI physical layer protocol. Technical report, 1992.

[FK91] S. Floyd and R. Karp. FFD bin packing for item sizes with distributions on $[0, 1/2]$. *Algorithmica*, 6:222–240, 1991.

[Fly66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, November 1966.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[FM82] C. M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[GBF+96] A. Gunzinger, B. Bäumle, M. Frey, M. Klebl, P. Kohler, R. Morel, and M. Rosenthal. *Programming Environment for a High-Performance Parallel Supercomputer with Intelligent Communication*, pages 25–32. John Wiley & Sons, 1996.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability - A Guide to the Theorie of NP-Completeness.* W. H. Freeman Co., San Francisco, 1979.

[GL81] A. George and J. W.-H. Liu. *Computer solution of Large Sparse Positive Definite Systems.* Prentice Hall, Englewood Cliffs, N.J., 1981.

[GMS+92a] Anton Gunzinger, Urs Müller, Walter Scott, Bernhard Bäumle, Peter Kohler, and Walter Guggenbühl. Architecture and realization of a multi signalprocessor system. In José Fortes, Edward Lee, and Teresa Meng, editors, *Proceedings 1992 Application Specific Array Processors*, pages 327–340. IEEE Computer

Society Press, August 4–7, 1992, Berkeley, California 1992. Invited paper.

[GMS+92b] Anton Gunzinger, Urs Müller, Walter Scott, Bernhard Bäumle, Peter Kohler, Johann vonder Mühll, René Hüsler, Florian Müller, Wilfred van Gunsteren, and Walter Guggenbühl. Achieving super computer performance with a DSP array processor. In Robert Werner, editor, *Supercomputing '92*, pages 543–550. IEEE/ACM, IEEE Computer Society Press, November 16–20, 1992, Minneapolis, Minnesota 1992.

[Gra76] R. Graham. *Computer and Job Shop Scheduling Theory*, chapter Bounds on the Performance of Scheduling Algorithms, pages 165–227. John Wiley and Sons, New York, London, Sydney, 1976.

[GY92] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling dags on multiprocessors. *Parallel and Distributed Computing*, pages 276–291, 12 1992.

[HL93] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. Technical report, 1993.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[Hwa91] Kai Hwang. *Advanced computer architecture, parallelism, scalability, programmability.* McGraw-Hill, New York, 1991.

[IEC73] IEC. Automatic gain control devices. *IEC Recommendation*, 268-8:20–22, 1973.

[IK94] M. Isler and M. Klebl. *Audiointerface für MUSIC*, 1994. Diplomarbeit, WS93/94, Institut für Elektronik, ETH.

[Jac86] Leland Jackson. *Digital filters and signal processing.* Kluwer, 1986.

[JDU+74] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974.

[JDU+75] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. NP-complete scheduling problems. *Computer and System Design*, 10:384–393, 1975.

[KA96] Y. Kwok and I. Ahmed. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[KK95] G. Karypis and V. Kumar. Multilevel graph partitioning and sparse matrix ardering. In *Intl. Conf. on Parallel Processing*, 1995.

[KL70] B. W. Kerninghan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.

[Lam79] H.Y. Lam. *Analog and Digital Filters.* Prentice Hall, Englewood Cliffs, N.J., 1979.

[LGTP90] Ken Linton, Lee Gould, Stephen Terepin, and Alan Purvis. Optimising massive parallel architectures for real-time digital audio. In *89th Audio Engineering Society Convention*, Los Angeles, USA, September 1990.

[LGTP91a] Ken Linton, Lee Gould, Stephen Terepin, and Alan Purvis. On the re-allocation of processing resources for digital audio signal processing. In *IEE Colloquium on "Digital Audio Signal Processing"*, page 7/1, May 1991.

[LGTP91b] Ken Linton, Lee Gould, Stephen Terepin, and Alan Purvis. A scalable hybrid multiprocessor for real-time digital audio signal processing. In *IEE Colloquium on "Digital Audio Signal Processing"*, page 9/1, May 1991.

[LM87a] Edward Lee and David Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[LM87b] Edward Lee and David Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sep. 1987.

[LTP91] Ken Linton, Stephen Terepin, and Alan Purvis. Taskforce scheduling strategies for digital mixing consoles. In *90th Audio Engineering Society Convention*, Paris, February 1991.

[McN84a] G. W. McNally. Dynamic range control of digital audio signals. *Journal of Audio Eng. Soc. (AES)*, 32:316–327, 1984.

[McN84b] G.W. McNally. AES Recommended Practice for Professional Digital Audio Applications - Preferred Sampling Frequencies. *Journal of Audio Eng. Soc. (AES)*, 32:781–785, 1984.

[MK91] Ch. Musialik and Th. Knaeple. A user configurable digital audio mixing console for broadcast applications. In *Convention of Audio Eng. Soc. (AES)*, number 3043, 1991.

[MM94] Th. Meier and M. Muheim. *Audioverarbeitung auf MUSIC*, 1994. Diplomarbeit, SS94, Institut für Elektronik, ETH.

[Mot89] Motorola Inc. *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, 1989.

[MT90] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations.* Wiley, cop., New York, 1990.

[MTTV93] Gary L. Miller, Shang-Hua Teng, W. Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, New York, 1993. Springer.

[Mue93] Urs A. Mueller. *Simulation of Neural Networks on Parallel Computers*, volume 23 of *Series in Microelectronics.* Hartung-Gorre, Konstanz, Germany, 1993. Ph.D. Thesis at the Swiss Federal Institute of Technology (ETH), Zürich.

[Mue96] Hansruedi Vonder Muehl. *Concept and Implementation of a Scalable Architecture for Data-Parallel Computing.* Series in Microelectronics. Hartung-Gorre, Konstanz, Germany, 1996. Ph.D. Thesis at the Swiss Federal Institute of Technology (ETH), Zürich.

[ODFB91] R. Oldehoeft, D.Cann, J. Feo, and A. Böhm. *The SISAL 2.0 Reference Manual.* Technical Report, Lawrence Livermore National Laboratory, December 1991.

[PSL90] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, January 1990.

[PSWB92] Alex Pothen, Horst D. Simon, Lie Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomuting'92 Proceedings*, pages 42–51, 1992.

[RC85] R. Rabenstein and R. Czarnach. Stability of recursive time-varying digital filters by state vector transformation. *IEEE Signal Processing*, 8(1):75–92, Feb. 1985.

[RKGT95] M. Rosenthal, M. Klebl, A. Gunzinger, and G. Tröster. A freely configurable audio mixing-engine with automatic load-balancing. In *Convention of Audio Eng. Soc. (AES)*, number 3979, 1995.

[Ros94] M. Rosenthal. Multi I/O Board for MUSIC. Technical report, Institut für Elektronik, ETH, Zürich, 1994.

[Rot95] Fritz Rothacher. *Sample-Rate Conversion: Algortithms and VLSI Implementation*, volume 44 of *Series in Microelectronics*. Hartung-Gorre, Konstanz, Germany, 1995. Ph.D. Thesis at the Swiss Federal Institute of Technology (ETH), Zürich.

[RS94] A. Rhomberg and R. Schweikert. *MADI Interface für MUSIC*, 1994. Studienarbeit, WS93/94, Institut für Elektronik, ETH.

[SG85] S. K. Skedzielewski and J. Glauert. *IF1 - an intermediate Form for Applicative Languages*. manual M-170, Lawrence Livermore National Laboratory, July 1985.

[SGB+93] W. Scott, A. Gunzinger, B. Bäumle, P. Kohler, U.A. Müller, H. Vonder Mühll, A. Eichenberger, W. Guggenbühl, N. Ironmonger, F. Müller-Plathe, and W. F. van Gunsteren. Parallel molecular dynamics on a multi signalprocessor. In *Computer Physics Communication*, volume 73, pages 65–86, 1993.

[Sho86] P. Shor. The average case analysis of some on-line algorithms for bin packing. *Combinatorica*, 6:179–200, 1986.

[Sho91] P. Shor. How to pack better than Best Fit: Tight bounds for average-case on-line bin packing. In *32nd Symposium on Foundations of Computer Science*, pages 752–759. IEEE, 1991.

[Sin87] J. Sinclair. Efficient computation of optimal assignments for distributed tasks. *Journal of Parallel and Distributed Computing*, 4:342–363, 1987.

[SMPvG94] W. Scott, F. Müller-Plathe, and W. F. van Gunsteren. Molecular dynamics study of the mixing and demixing of a binary lennard-jones fluid. In *Molecular Physics*, volume 82, 1994.

[SSNB95] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[Stu91] J. R. Stuart. High quality digital audio. In *IEE Colloquium on "Digital Audio Signal Processing"*, page 1/1, May 1991.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntatic definitions. *Communications of the ACM*, 20(11):822–823, 1977.

[Zoe89] Udo Zoelzer. *Entwurf digitaler Filter für die Anwendung im Tonstudiobereich.* PhD thesis, Techn. Univ. Hamburg-Harburg, 1989.

[Zoe96] Udo Zoelzer. *Digitale Audiosignalverarbeitung.* Teubner, Stuttgart, Germany, 1996.

[ZRB93] U. Zoelzer, B. Redmer, and J. Buchholtz. Strategies for switching digital audio filters. In *Convention of Audio Eng. Soc. (AES)*, number 3714, 1993.

Curriculum Vitae

14.10.1968	Born in Liestal near Basel
1975-1980	Primary school in Allschwil
1980-1984	Progymnasium in Allschwil
1984-1987	Gymnasium am Kohlenberg in Basel
1987	Matura Typus B
1987-1993	Swiss Federal Institute of Technology Zürich (ETH)
1993	Diploma in Electrical Engineering (Dipl. El.-Ing. ETH)
1993-1996	Research Assistant, Electronics Laboratory, ETH Zürich