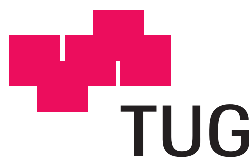
Diploma Thesis

Analysis and implementation of the position-pitch source localization algorithm on a hybrid reconfigurable CPU

Wolfgang Jäger

Signal Processing and Speech Communication Lab
Graz University of Technology

Research Lab Computational Technologies and Applications
University of Vienna



universität
wien

Assessor: Univ.-Prof. Dipl.-Ing. Dr.techn. Gernot Kubin

Supervisors: Dipl.-Ing. Dr.sc.ETH Harald Romsdorfer
Dipl.-Ing. Dr.techn. Manfred Mücke

Graz, March 2011

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ____________________ ____________________

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, ____________________ ____________________

Ort, Datum Unterschrift

Abstract

Source localization is an important component in speech processing systems, as automated source localization in combination with beamforming makes it possible to enhance the speech of individual speakers in reverberating environments, hence providing signals with improved signal-to-noise ratio for subsequent speech processing algorithms. The source localization algorithm used in this thesis consists of four main parts: Gammatone filterbank, cross correlation, Position-Pitch decomposition and final evaluation with interframe tracking. This thesis presents an analysis of the source localization algorithm with the aim to find reasonable calibrations for its accuracy and time performance. To achieve a real-time implementation the algorithm is ported to reconfigurable FPGA based hardware. The hardware used is a Stretch S6 hybrid reconfigurable CPU, in which certain computationally expensive parts of the code can be outsourced. The cross correlation is identified as the most computationally expensive part of source localization, and several approaches of it have been implemented at the hardware. In order to elaborate on advantages and disadvantages of the different approaches, measures of time performance are compared.

Kurzfassung

Die Lokalisierung von Schallquellen stellt einen wichtigen Baustein im Gebiet der Sprachsignalverarbeitung dar. Automationsgestützte Schallquellenlokalisierung in Kombination mit Beamforming ermöglicht es, Sprachsignale aus bestimmten Richtungen zu verstärken und raumbedingte Schallreflexionen, die vorallem in halliger Umgebung auftreten, zu unterdrücken. Dadurch wird der Signal-Rausch Abstand des Signals für nachfolgende Sprachsignalverarbeitungsstufen verbessert. Der hier verwendete Algorithmus zur Schallquellenlokalisierung besteht aus vier Hauptbestandteilen: Gammaton Filterbank, Kreuzkorrelation, Position-Pitch Dekomposition und abschließender Auswertung mit frameübergreifendem Tracking. Um den Schallquellenlokalisierungsalgorithmus zu kalibrieren, bedarf es einer umfassenden Genauigkeits- und Geschwindigkeitsanalyse. Der kalibrierte Algorithmus wird anschließend auf eine rekonfigurierbare FPGA basierte Hardware portiert, um ein echtzeitlauffähiges System zu erhalten. Bei der Hardware handelt es sich um eine Stretch S6 hybride rekonfigurierbare CPU, auf der rechenintensive Programmcodeteile ausgelagert werden können. Für den untersuchten Algorithmus stellt die Kreuzkorrelation den rechenintensivsten Programmteil dar. Es wurden unterschiedliche algorithmische Lösungen für die Kreuzkorrelation auf der zugrundeliegenden Hardware entwickelt. Die jeweiligen Lösungen wurden in einem abschließenden Schritt einer Geschwindigkeitsanalyse unterzogen und miteinander verglichen.

Danksagung

Ein besonderer Dank gilt meinen beiden Betreuern Harald Romsdorfer und Manfred Mücke für die Betreuung und Unterstützung während dieser Diplomarbeit. Bedanken möchte ich mich auch bei Thang Huynh Viet und Tania Habib, die mir für themenbezogene Fragen jederzeit zur Verfügung standen. Vielen Dank an Boris Clénet für die wertvolle Zusammenarbeit. Für die freundliche Arbeitsatmosphäre im Labor möchte ich mich bei Anna Fuchs, Martin Schickbichler, Nikolaus Hammler, Susanne Rexeis und Florian Krebs bedanken. Ein Dankeschön an Anna, Barbara und Franz für das Gegenlesen dieser Arbeit. Ein großer Dank geht an meine Freundin Manuela, die während der Entstehung dieser Arbeit auf viel gemeinsame Zeit verzichten musste. Besonders danken möchte ich meinen Eltern, Eva und August, die mir dieses Studium ermöglicht haben.

Graz, March 2011 Wolfgang Jäger

Contents

1 Introduction **11**

2 Source localization **13**

2.1 Introduction to source localization 13
2.2 Microphone arrays 14
2.3 Principle of source localization 14
2.4 Spatial aliasing 17
2.5 Near and far field assumptions 18
2.6 Position-Pitch based source localization 19
2.6.1 Gammatone filterbank 19
2.6.2 Cross correlation (CC) 22
2.6.3 Position-Pitch (PoPi) decomposition 24
2.6.4 Evaluation of the PoPi matrix and the tracker 27
2.7 Interaction between parts of the source localization algorithm 29
2.8 The frame mechanism and real-time requirements 30

3 Performance analysis and optimization of the algorithm **32**

3.1 Time performance of different implementations 32
3.2 Accuracy vs. time performance 36
3.2.1 Frame shift 36
3.2.2 Frame size 37
3.2.3 Sampling frequency 39
3.2.4 Filterbank adjustments 41
3.2.5 Resolution of input data 42
3.2.6 Number of microphone pairs 43

4 Stretch S6 hybrid reconfigurable CPU based development **46**

4.1 Overview of the S6 Xtensa ISEF board 46

4.2 The ISEF Xtensa processor . . . 47
4.3 S6 Instruction pipeline structure . . . 48
4.3.1 Issue Rate . . . 49
4.4 S6 Programming . . . 50
4.4.1 Defining and using Extension Instructions . . . 50
4.4.2 Handling the wide registers . . . 51
4.4.3 Handling the IRAM . . . 51
4.4.4 DMA transfers and resolution . . . 52
4.4.5 The BIOS . . . 53
4.5 S6 Software build process . . . 55
4.5.1 Report files . . . 56
4.6 Stretch integrated development environment . . . 57
4.6.1 Pipeline view . . . 57
4.6.2 Profiling . . . 58

5 Porting the source localization algorithm to reconfigurable hardware 59
5.1 Mathematical description of the CC . . . 59
5.2 CC decompositions . . . 61
5.2.1 Square decomposition . . . 61
5.2.2 Linewise decomposition . . . 64
5.2.3 Diagonal decomposition . . . 65
5.3 Data flow of the CC implementations . . . 67
5.3.1 Straightforward implementation without the ISEF . . . 67
5.3.2 Using the ISEF and WRs . . . 67
5.3.3 Using the ISEF and IRAM . . . 69
5.4 Experiments and results . . . 70
5.4.1 Comparison of 8 bit implementations . . . 71
5.4.2 Comparison of 16 bit implementations . . . 72
5.4.3 Comparison of 24 bit implementations . . . 73
5.4.4 Performance observations of further code parts . . . 74

6 Conclusion 77

A RIFF-WAVE file format 79

B Figures 81

B.1 Additional Figures to chapter 4 81

C Extension Instructions of the CC 85

C.1 Square Decomposition 85
C.2 Linewise Decomposition 86
C.3 Linewise decomposition with IRAM 87
C.4 Diagonal Decomposition 88

List of References 93

List of abbreviations

ALU	Arithmetic Logic Unit
AR	ALU's associated register
AU	Arithmetic Units
CC	Cross Correlation
CPU	Central Processing Unit
D-CACHE	Data-Cache
DATARAM	Dual Port RAM
DMA	Direct Memory Access
DOA	Direction Of Arrival
EI	Extension Instruction
ERB	Equivalent Rectangular Bandwidth
ER	Extension Register
EUs	Extension Unit cycles
FFT	Fast Fourier Transform
FPU	Floating Point Unit
FR	FPU's associated register
GCC	Generalized Cross Correlation
GIB	Generic Interface Bus
GMAC	Ethernet Media Access Controller
IDE	Integrated Development Environment
IIR	Infinite Impulse Response
IRAM	Inherent ISEF's RAM
IR	Issue Rate
ISEF	Instruction Set Extension Fabric
ISS	Instruction Set Simulator
MCC	Matlab compiler

MU	Multiplication Units
PA	Processor Array
PE	Processor Entities
PoPi	Position-Pitch
SBIOS	Stretch BIOS
SCC	Stretch-C Compiler
SCP	Software Configurable Processor
TDOA	Time Difference Of Arrival
TOA	Time Of Arrival
WR	Wide Register

1 Introduction

This thesis describes the development process of accelerating a signal processing algorithm to allow for an execution in real-time. Figure 1.1 depicts the flow diagram of the acceleration development process.

As a first step, the signal processing algorithm is analyzed with regards to time performance. The time consuming parts of the algorithm are optimized in order to reduce the execution time of the algorithm. A further reduction in execution time can be achieved by investigating the parameters of the algorithm. If it is possible to reduce the complexity of the algorithm without loosing accuracy, computational expenses can be reduced. A compromise between time and quality performance has to be made.

The resulting algorithm is reduced to its essentials. If a real-time execution is not

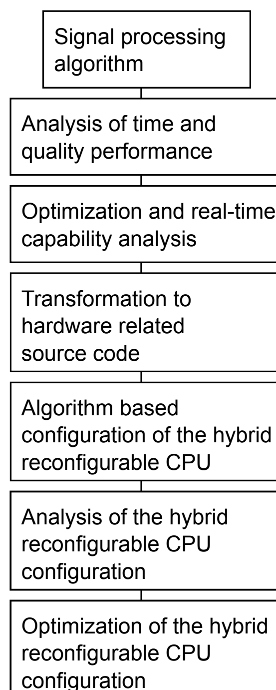


Figure 1.1: Development process of accelerating a signal processing algorithm on a hybrid reconfigurable CPU

possible at this point the computational power of the signal processing hardware has to be increased. In this thesis a hybrid reconfigurable Central Processing Unit (CPU) is used in order to speed-up the most expensive computational part of the algorithm. Therefore, the algorithm has to be transformed to hardware related source code. A configuration for the hybrid reconfigurable CPU has to be found where computations can be carried out in parallel. In order to find an optimal solution, an iterative analysis and optimization process for the hybrid reconfigurable CPU solution is necessary.

The source localization algorithm serves as a good example for testing the acceleration development process concept. Chapter 2 introduces the principles of the source localization algorithm and describes constituent sub algorithms. In chapter 3, time/quality performance analysis and the optimization of the source localization algorithm are discussed. Chapter 4 explains the hybrid reconfigurable CPU environment used. Observations discussed in chapter 3 are performed on a hardware abstract layer. The transformation of the main parts of the algorithm to the hybrid reconfigurable CPU is shown in chapter 5. To conclude this thesis, results of the acceleration process are verified.

2 Source localization

This chapter provides an overview of the source localization algorithm used in this thesis. After an introduction to microphone array-based acoustic source localization strategies, several aspects of microphone array processing that are important for implementation are discussed in more detail. Section 2.6 introduces the four particular parts of the source localization algorithm. In section 2.7 the interaction between the parts of the source localization algorithm is presented. Parallel structures essential for distributed computations are located. The segmentation of the input data into frames is explained in section 2.8 together with an exact definition of the real-time constraint. [1]

2.1 Introduction to source localization

A source localization system can be placed in a conference room where several speakers are talking. Ideally, it localizes all of the active speakers and returns the positions of the sources as output. Audio signals are recorded with microphones, arranged in a microphone array. Figure 2.1 depicts an exemplary scenario using a circular microphone array. The process of obtaining the positions of the speakers is called *source localization*.

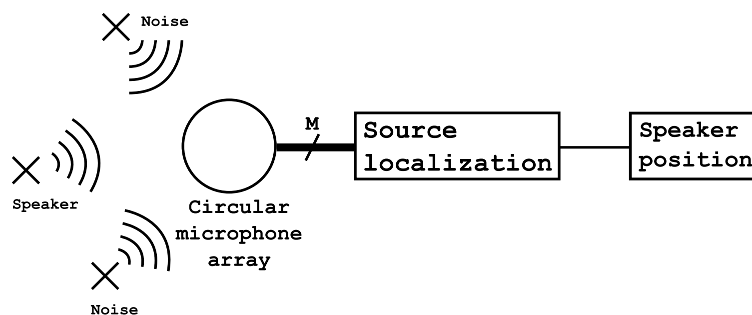


Figure 2.1: Exemplary source localization scenario, using a circular microphone array with M microphones.

[1]Parts of this chapter have been developed in cooperation with Boris Clénet and appear in his master thesis too. [1]

The output – the positions of the speakers – can be used to steer a beamforming algorithm which is able to enhance speech from defined directions and suppress ambient noise (e.g. there is no need of a personal microphone). [2] [3]

2.2 Microphone arrays

A microphone array is defined as an arrangement of multiple spatially separated microphones. Several configurations exist, which can be divided into three groups: Linear, planar and volumetric arrays. Each group has its own limitations regarding the covered spatial range. For example a linear array covers a azimuthal range of 180 ° while a planar array covers a azimuthal range of 360 °. [4]

In this thesis a planar array has been used. The circular shape of the planar array, with a diameter of $d = 0.55\,\mathrm{m}$ consists of 24 equally-spaced microphones and is depicted in Figure 2.2. The main advantages of this array are that it covers a range of 360 ° azimuth, and that it is less affected by spatial aliasing than linear arrays would be (cf. section 2.4).

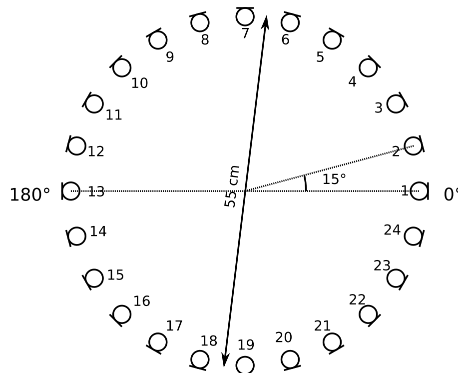


Figure 2.2: Array configuration of the applied circular microphone array with $M = 24$ equally-spaced microphones.

2.3 Principle of source localization

Microphone arrays can be used to estimate the position of a sound source relative to the position of the array, especially when a dominant source and several interfering sources overlap. The cross correlation (CC) operation is the core of this source localization algorithm. [5]

In order to understand the basic concepts of source localization and why the CC operation is important for it, a simple scenario is assumed. Two microphones receive the speech signal from a single source in the far field of an array (cf. section 2.5). No other source is present. Figure 2.3 depicts this scenario.

Depending on the speaker's position, the Time Of Arrival (TOA) of the speech's sound waves at each microphone is different, because the relative distance to each microphone is not the same.

From the Time Difference Of Arrival (TDOA)($\tau_{M_1,M_{13}}$) between microphones M_1 and M_{13}, the angular Direction Of Arrival (DOA)(θ) of the speech can be derived, according to (2.2). Microphone pairs are always chosen between opposite microphones. A straight line through the origin and M_1 of the circular microphone array of Figure 2.2, results in the opposite microphone M_{13}. Microphone M_1 and microphone M_{13} are forming a microphone pair. If M_1 is fixed as the reference, the wavefront arrives at M_{13} with a relative time delay. Therefore $\tau_{M_1,M_{13}}$ has a negative value and

$$\tau_{M_1,M_{13}} = -\left(\frac{d}{c}\right)\cos(\theta) \ , \quad \text{hence} \tag{2.1}$$

$$\theta = \arccos\left(-\tau_{M_1,M_{13}}\frac{c}{d}\right) \ , \tag{2.2}$$

where c is the speed of sound in the environment of the array ($c \approx 343\,\mathrm{m/s}$, at $20\,^{\circ}\mathrm{C}$). The cosine term in (2.1) can vary in the range of $[-1, 1]$. Therefore, the maximum possible value for the TDOA is (d/c). TDOA measurements are performed with discretized input signals. The sampling frequency (f_s) of the discretization has to be considered.

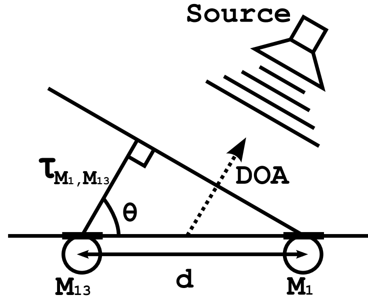


Figure 2.3: Simple scenario of source localization: M_1 and M_{13} are the recording microphones, separated by the distance d. The DOA θ states the angular Direction Of Arrival. $\tau_{M_1,M_{13}}$ is the TDOA of the plane sound wave between microphone M_1 and microphone M_{13}.

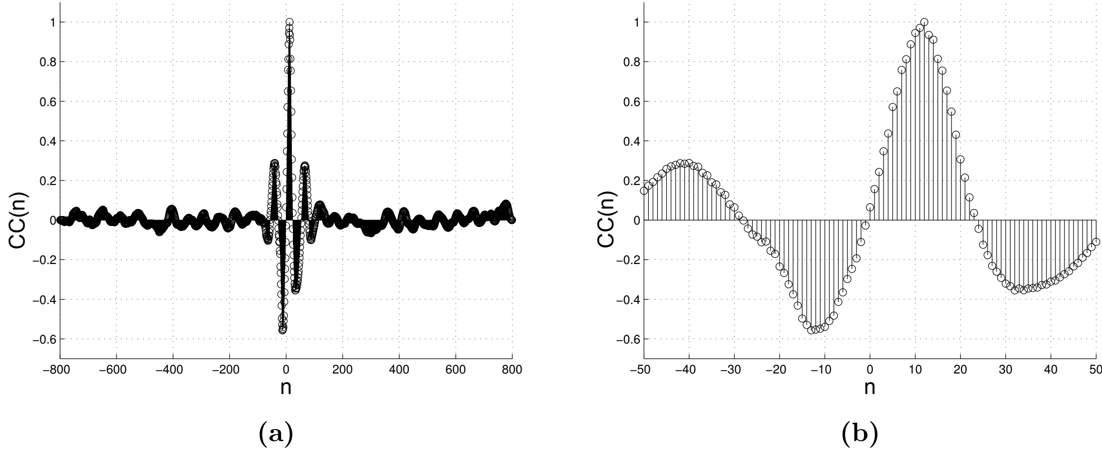


Figure 2.4: **(a)** Exemplary result vector of the CC between microphone M_1 and M_{13}. Speech signal analysis time = 100 ms; Sampling frequency = 8 kHz; **(b)** Enlarged area around sample ±50, with the first peak at $n_{\max} = 12$.

The TDOA values are multiples of the sampling period. The discretized maximum TDOA is (df_s/c).

The CC indicates the likeness of two signals. It reaches its maximum when the signals match the most. Figure 2.4a shows the CC between the outputs of microphones M_1 and M_{13} when only one speaker is present. In Figure 2.4b the x-axes range between the samples $n = \pm 50$ is depicted in more detail.

The position of the maximum $n_{\max}$ and $\tau_{M_1,M_{13}}$ are linked by

$$\tau_{M_1,M_{13}} = \frac{n_{\max}}{f_s} \ . \tag{2.3}$$

By combining (2.2) and (2.3), the DOA can be derived. In Figure 2.4, $n_{\max} = 12$, which results in a DOA of $\theta \approx 160\,°$.

Because $\tau_{M_1,M_{13}}$ can not be greater than (d/c), the first peak of the CC vector has to lie between $[-df_s/c\ ;\ df_s/c]$. Depending on the sign of $n_{\max}$, the speaker is closer either to M_1 or M_{13} (i.e. if $\theta < 90\,°$ or $\theta > 90\,°$). Indeed if $\theta < 90\,°$, as Figure 2.3 shows, the signal has a time advance at M_1. Therefore the CC between the outputs of M_1 and M_{13} has a maximum that is retarded ($n_{\max} > 0$).

With a linear microphone array, DOAs in the range of $[0\,°; 180\,°]$ can be derived. Nevertheless it is impossible to distinguish if θ or $-\theta$ is detected. A source positioned

at θ and a source positioned at $-\theta$ return the same CC and the same TDOA (*mirrored source*). In the case of a circular array the DOAs range is widened to $[0\,°; 360\,°]$, since uncertainties in the sign are eliminated by the results of further microphone pairs.

2.4 Spatial aliasing

Similar to the *Nyquist criterion* in temporal sampling [6], which has to be fulfilled in order to avoid temporal aliasing, there are restrictions for spatial sampling [7]. The studied scenario of section 2.3 is considered. A sinusoidal signal, with a planar wavefront and a frequency $f_{\max}$ serves as source signal for investigations presented in this section. To avoid spatial aliasing the phase difference between the two microphones has to remain in the range $\pm\pi$. Otherwise it is not possible to determine whether the calculated $\tau_{M_1,M_{13}}$ is induced by the actual phase shift, or from a modulo π version of it:

$$2\pi f_{\max}\frac{d}{c}cos(\theta) \leq \pi \ . \tag{2.4}$$

An ambiguous determination of $\tau_{M_1,M_{13}}$ results in multiple possibilities for the DOA (θ). The relation

$$d_{\max} \leq \frac{c}{2\ f_{\max}} \tag{2.5}$$

provides the maximal distance $d_{\max}$ between microphones that will not lead to spatial aliasing for a known $f_{\max}$. It assumes the worst case, which is $\theta = 0\,°$.

In contrary to (2.5), the diameter of the microphone array needs to be large enough to determine the angular speaker positions accurately. The accuracy in determining a speaker position increases with increasing values of the term df_s/c (section 2.6.3 gives detailed information about speaker detection accuracy). If the sampling frequency is fixed, an increasing diameter increases the possible accuracy of speaker detection. Therefore a compromise between the effects of spatial aliasing and speaker detection accuracy has to be made by choosing the diameter of the circular microphone array.

The applied microphone array has a diameter of $d = 0.55\,\text{m}$. In order to be certain that no spatial aliasing occurs for a single microphone pair of the microphone array, a maximal frequency of $f_{\max} \approx 312\,\text{Hz}$ can be derived from (2.5). Speech signals contain frequencies higher than 312 Hz, but for a circular microphone array the DOA is different

for every pair of microphones.

$$f_{\max} \leq \frac{c}{2\ d\ cos(\theta)}\ , \tag{2.6}$$

considers the DOA. Table 2.1 shows the relationship between the DOA and $f_{\max}$. The used microphone array consists of $M = 24$ equally-spaced microphones (Figure 2.2). The angular distance between two microphone pairs is $15\,°$. Therefore the worst possible DOA for a speech signal at the circular microphone array used is $15\,°/2 = 7.5\,°$. Subtracting $7.5\,°$ from the best case scenario with $\theta = 90\,°$ leads to $\theta = 82.5\,°$ and $f_{\max} = 2389$ Hz for the used microphone array, which is sufficient for speech processing purposes.

θ	$0\,°$	$15\,°$	$30\,°$	$45\,°$	$60\,°$	$75\,°$	$82.5\,°$
$f_{\max}$	312 Hz	323 Hz	360 Hz	441 Hz	624 Hz	1205 Hz	2389 Hz

Table 2.1: Relationship between the DOA and $f_{\max}$ of a sinusoidal signal in relation to spatial aliasing for the scenario in Figure 2.3.

2.5 Near and far field assumptions

If the distance between a source and the microphone array is significantly larger than the dimensions of the array, the source is in the *far field* of the array. In this case the curvature of the arriving wavefront is negligible. The wavefront appears to be planar when it arrives at the microphones.

Inversely, the source is said to be in the *near field* of the microphone array when the source is so close to the array that the curvature of the arriving wavefront has to be considered. In this case, the signal arrives at each microphone of the microphone array with a different DOA.

According to [8], a rule of thumb for the distance r from the origin of the microphone array to where the *far field* approximation begins to be valid is

$$r \geq \frac{2L^2}{\lambda}\ , \tag{2.7}$$

where L is the largest physical dimension of the microphone array and λ the wavelength of operation. The main analysis frequency range of the used source localization algorithm reaches from 50 Hz to 600 Hz. This frequency range leads to an approximate distance

$r = 0.088\,\text{m} - 1.06\,\text{m}$, where the *far field* approximation begins to be valid. Assuming that speaker recordings are taken at distances of $r \approx 2\,\text{m}$, the *far field* approximation holds for all implementations considered in this thesis.

2.6 Position-Pitch based source localization

The Position-Pitch (PoPi) based localization algorithm consists of four main parts. First the recorded audio signals are digitally sampled using A/D converters. Then the following sequence of analysis steps as shown in Figure 2.5 is applied in a frame wise manner:

1. Gammatone filterbank: The Gammatone filterbank filters the audio signals into frequency bands. After the filtering process the following cross correlation is less affected by wide-band noise.

2. Cross correlation (CC): The CC is the core of the source localization algorithm. It detects the TDOA between the microphones of one microphone pair. If a time domain approach is chosen, the CC can be performed directly at the speech signals. If a frequency domain approach is chosen, a Fourier Transform has to be made before the CC, and an inverse Fourier Transform after the CC.

3. PoPi decomposition: Transforms the one dimensional CC output vector into a two dimensional matrix (DOA vs. fundamental frequency).

4. Analysis of the PoPi matrix and the tracker: Smoothing of outliners by tracking over time using several consecutive PoPi analysis frames.

Figure 2.5: Structure of the PoPi based source localization algorithm.

2.6.1 Gammatone filterbank

The audio signals recorded by the 24 microphones serve as input for the algorithm (Figure 2.2). A filterbank with 64 filters (cf. Figure 2.6) generates 64 filtered output signals in order to improve reverberation suppression.

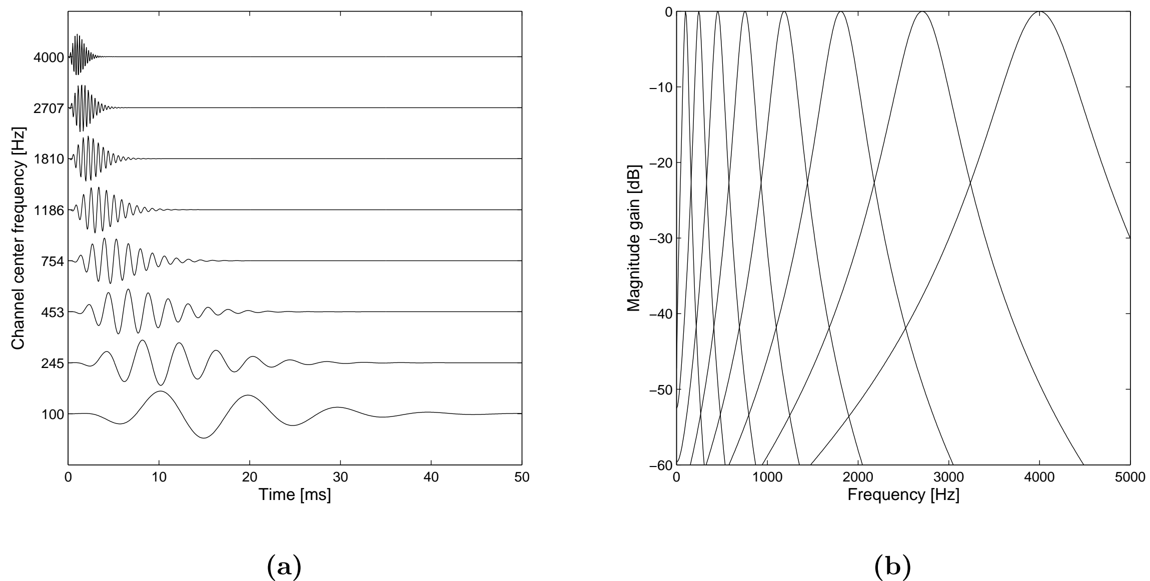
For auditory purposes, [9] recommends a Gammatone filterbank due to its good reproduction of the model for the impulse response function of auditory nerve fibers. The analytical formula for one channel of the filterbank is given by

$$g(t) = at^{n-1}\cos\big(2\pi f_c t + \phi\big)e^{-2\pi b(f_c)t} \,, \tag{2.8}$$

where a is the amplitude of the filter, n is the order of the filter, f_c is the filter center frequency, $b(f_c)$ is the center frequency dependent bandwidth of the filter and ϕ is the phase.

The bandwidth of the filter bank increases with increasing frequency, as shown in Figure 2.6. The relation between center frequency and bandwidth is defined by the Equivalent Rectangular Bandwidth (ERB) given by

$$\mathrm{ERB}(f) = 24.7 + 0.108f. \tag{2.9}$$



(a) **(b)**

Figure 2.6: Gammatone filters. (**a**) Impulse responses for eight Gammatone filters with center frequencies between 100 Hz and 4 kHz. (**b**) Frequency responses of the filters shown in panel **a**. [9]

For a fourth-order implementation the ERB is correlated to the filter bandwidth

$$b(f) = 1.019 \text{ ERB}(f) \ . \tag{2.10}$$

Cooke [10] derived the time discrete transfer function from the analytical transfer function by performing an Impulse Invariant transformation [11]. Ma [12] improved the efficiency further: First the signal is shifted to the base band, where it is filtered by a Gammatone filter. Then it is shifted back to its original frequency band. Down/up shifting is performed by the complex exponential functions $e^{\mp j2\pi ft}$. The exponential functions are calculated using the relations to trigonometric functions:

$$e^{-j2\pi ft} = \cos(2\pi ft) - j\sin(2\pi ft) \ , \tag{2.11}$$

$$e^{j2\pi ft} = \cos(2\pi ft) + j\sin(2\pi ft) \ . \tag{2.12}$$

Due to the computational complexity of shifting the actual frequency band to the base band, the exponential functions are rearranged:

$$e^{-j2\pi ft} = e^{-j2\pi f} e^{-j2\pi f(t-1)} \ . \tag{2.13}$$

Rearranging reduces the computational complexity considerably, as exponentials only have to be calculated once ($e^{-j2\pi f}$) and consecutive results can be derived from previous elements ($e^{-j2\pi f(t-1)}$).

Two implementations for the time discrete transfer function of the Gammatone filter have been available. Implementation 1 uses the inherent Matlab *filter* function, implementation 2 has been developed by Ma [12] and uses a MEX subroutine. Implementation 2 is implemented as an Infinite Impulse Response (IIR) filter with a *Direct Form II*, also called a *Canonic Direct Form* structure (cf. [13]). The advantage of *Direct Form II* is that in comparison to *Direct Form I* only one delay line is needed, on the other hand the number of necessary accumulators doubles in comparison to *Direct Form I*. The filter graph of this implementation is depicted in Figure 2.7.

During the course of this thesis, implementation 2 has been modified to be able to buffer values of the filter delay line and the exponential elements for frequency band down/up shifting between consecutive frames. An implementation in C-code has been developed to run on the Stretch board. Ma's MEX subroutine [12] served as the basic framework for this procedure.

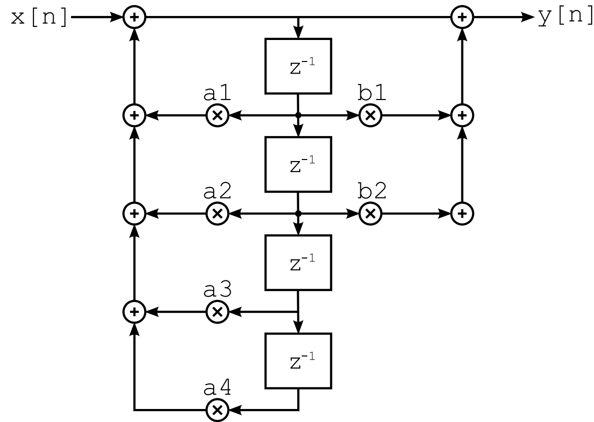


Figure 2.7: *Direct Form II* implementation of one Gammatone IIR filter channel.

Due to the fixed point data-type approach of the following computation of the CC, the result of the filter has to be a fixed point data-type. Therefore a study of the impulse responses

$$y_M[n] = \sum_{k=-\infty}^{\infty} h_M[k]\delta[n-k] \ , \qquad \delta[n] = \begin{cases} 1, & \text{if } n = 0 \\ 0, & \text{if } n \neq 0 \end{cases} \quad , \tag{2.14}$$

of the filter bank is necessary in order to prevent integer overflows (cf. [14]).

Internally the filter bank algorithm is calculated with floating point digits. In order to get fixed point numbers at the output of the filter bank, the result has to be truncated appropriately. An evaluation of the output range is possible by adding up the absolute values of the impulse response at the filter-structures point in question:

$$c = \arg \max_{M=1\ldots 64} \sum_{n=-\infty}^{\infty} |y_M[n]| \quad , \tag{2.15}$$

where M is the number of filterbank channels. Outputs of the filter bank have to be truncated by the factor c in order to prevent an overflow at the floating–to–fixed point conversion.

2.6.2 Cross correlation (CC)

Within this thesis the cross correlation has been computed with two different methods. The straightforward approach computes (2.16), which is the shifted version of the inner

product of two input vectors $x[n]$ and $y[n]$.

$$CC[n] = \langle x^*[m]y[m+n] \rangle \tag{2.16}$$

The benefit of this approach is its low mathematical complexity. Therefore it is possible to estimate the computational expenses exactly which is important to implement the CC on reconfigurable hardware. Due to the low complexity of the time domain approach the range of values of the CC vector can also be determined exactly. The value range is important for the use of fixed-point data types in order to avoid truncations. The time domain approach is used to investigate solutions regarding the Stretch hardware (the structure of the source localization algorithm when using the time domain approach is depicted in Figure 2.13).

A more flexible approach is available with the Generalized Cross Correlation (GCC) [15] [16]:

$$\mathrm{GCC}(f) = X(f)Y^*(f) \tag{2.17}$$

Here the result is computed by multiplications in the frequency domain. However, algorithms transforming the available microphone signals into the frequency domain, such as the Fast Fourier Transform (FFT), are necessary.

Brandstein [17] recommends to use the GCC together with the so called “Phase Transform”

$$\mathrm{GCC}_{\mathrm{PHAT}}(f) = \frac{X(f)Y^*(f)}{|X(f)Y^*(f)|} \ , \tag{2.18}$$

where the envelope of the frequency domain signal is constant, and information available in the phase term only. This approach tends to be more robust, and is used under sub-optimal conditions (e.g. rooms with a lot of reverberation).

The applied frequency domain CC uses a combined version of GCC and $\mathrm{GCC}_{\mathrm{PHAT}}$ in order to increase localization accuracy and pitch estimation reliability (Figure 2.8). The range of the CC result vector, which is important for the localization of the sound sources (middle sample $\pm$ (df_s/c)), consists of the corresponding result of the $\mathrm{GCC}_{\mathrm{PHAT}}$. The $\mathrm{GCC}_{\mathrm{PHAT}}$ result vector flattens at its boundaries, and periodicities can not be determined as well as with the GCC. Therefore, the range of the CC result vector exceeding the middle sample $\pm$ (df_s/c) is important for periodicity and consists of the corresponding results from the GCC (the structure of the source localization algorithm when using the frequency domain approach is depicted in Figure 2.14).

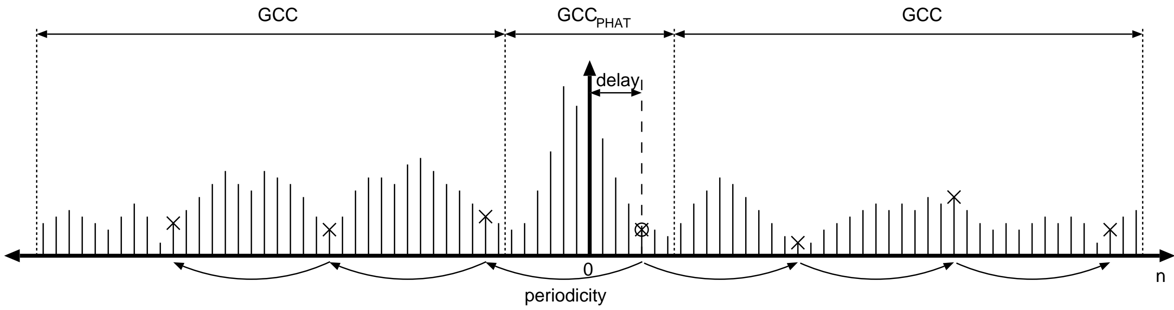


Figure 2.8: POPI decomposition: First the delay for a certain DOA is set. Then the value for the pitch is calculated. Therefore, three harmonics of the signal are evaluated by summing up the samples with the defined periodicity.

2.6.3 Position-Pitch (PoPi) decomposition

After performing the CC between the 64 corresponding filterbank channels of two microphones from a microphone pair, the 64 result vectors are summed up into one CC result vector for each frame (the frame mechanism is explained in section 2.8).

This result vector serves as input for the POPI algorithm, and contains two important pieces of information. The delay between the input signals of two microphones affects the delay of the first peak to the middle sample of the vector. In Figure 2.4 the first peak is at $n = 12$. In Figure 2.8 the first peak is at $n = -2$. Physically, the delay between the two input signals corresponds to the DOA of the sound waves (cf. section 2.3). Further

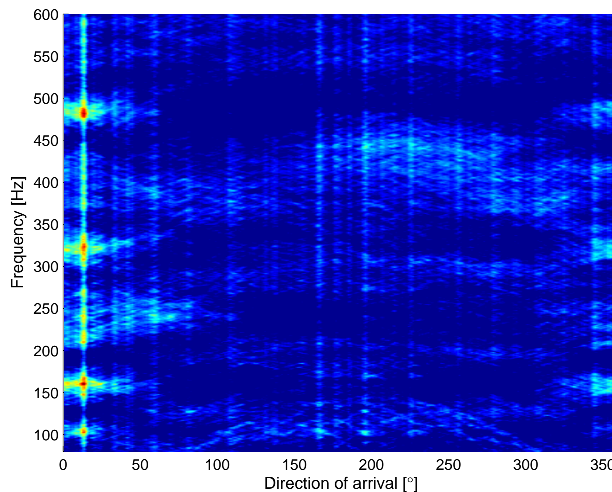


Figure 2.9: Visualisation of the Position-Pitch matrix.

peaks in the CC result vector represent occurring periodicities in the input signal. In Figure 2.4 periodicities can be observed for example at $n \approx -40$, $n \approx -13$, and $n \approx 33$. In Figure 2.8 periodicities can be observed for example at $n \approx -25$, $n \approx -13$, $n \approx 10$, and $n \approx 26$.

The PoPi decomposition creates a matrix

$$\hat{\mathbf{X}} = \begin{bmatrix} \hat{x}_{1,1} & \cdots & \hat{x}_{1,360} \\ \vdots & \ddots & \vdots \\ \hat{x}_{521,1} & \cdots & \hat{x}_{521,360} \end{bmatrix} \tag{2.19}$$

from the CC vector (depicted in Figure 2.8). The PoPi matrix evaluates the angular DOA over frequency. Instead of detecting peaks directly in the correlation vector, the PoPi algorithm generates a plot of the current speaker scenario. The DOA is varied between 1 ° and 360 °, with a step size of one degree. Frequencies between 80 Hz and 600 Hz are applied to the y-axis of the plot. The energy intensity is depicted in terms of color (cf. Figure 2.9).

The procedure of calculating an entry of the PoPi matrix is demonstrated in Figure 2.8: First the delay for a certain angular DOA is set (indicated by a circle in Figure 2.8), and then the value for the pitch is calculated. Therefore, three harmonics of the signal are evaluated by summing up the defined samples (indicated by crosses in Figure 2.8). The higher the energy of the samples, the more likely a speaker is talking at the current position and pitch.

Due to the physical dimensions of the circular microphone array used, speakers are positioned outside of the microphone array. Nevertheless the DOA is projected onto the interconnection between a microphone pair, and is available as a time difference Δt. The projection follows equation

$$\Delta t = \frac{\cos(\theta)\ f_s\ d}{c} \ , \tag{2.20}$$

where θ states the angular DOA, d is the distance between the two microphones of one microphone pair, and c the speed of sound.

The sampling frequency f_s rasterizes the time difference, and therefore only a certain angular accuracy can be achieved. For a first rough estimate, the accuracy is determined by

$$\Delta\theta = \frac{180\ c}{2\ f_s\ d} \ , \tag{2.21}$$

where the number of possible time lags between the two microphones is distributed equally over a semi circle. However, the projection includes a cosine term and is therefore nonlinear. After sampling, the projection is more accurate at the center and not as accurate at the lateral (Figure 2.10 and Figure 2.11). The relationship between the angular DOA and the accuracy is associated via

$$\Delta\theta = \arccos\left(\frac{\tau[n]\ c}{f_s\ d}\right) - \arccos\left(\frac{\tau[n-1]\ c}{f_s\ d}\right), \quad \Delta\tau[n] = \Delta t(nT)\ , \tag{2.22}$$

where the difference in angle for two consecutive samples states the accuracy. In Table 2.2 resolutions for given sampling frequencies are listed.

Sampling frequency	Rough estimate	Accuracy lateral	Accuracy central
48 kHz	1.17 °	3.89 °	0.74 °
32 kHz	1.75 °	6.66 °	1.12 °
16 kHz	3.51 °	7.71 °	2.23 °
8 kHz	7.02 °	10.26 °	4.47 °

Table 2.2: Angular accuracy for different sampling frequencies.

To recap, as a consequence of the PoPi algorithm the semi circle of the DOA is divided into segments of one degree. The projection of these segments onto the interconnection between the microphones would lead to a nonlinear raster. Because sampling is linear and determined by the sampling frequency, the consequence is a mismatch affecting the angular accuracy. Resolution is more accurate at the center and less accurate at the lateral.

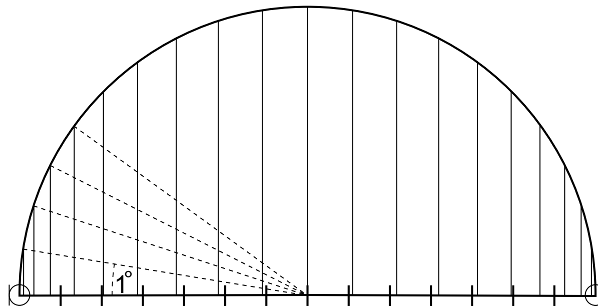


Figure 2.10: Projection mismatch between the equally spaced semi circle of DOA and the sampled interconnection between two microphones.

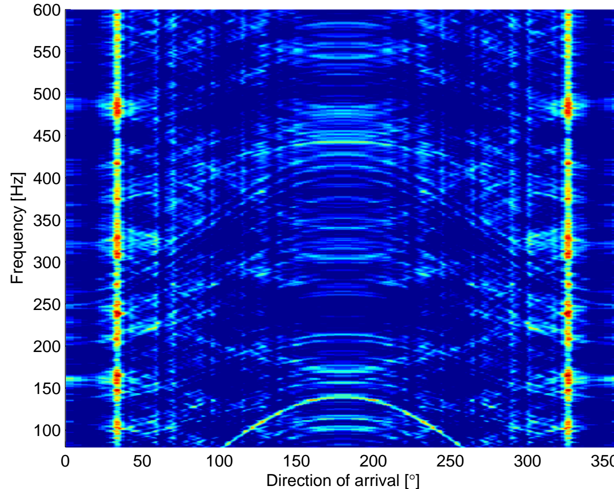


Figure 2.11: PoPi matrix for one microphone pair. The PoPi matrix is symmetric for one microphone pair (*mirrored source*, cf. section 2.3). The resolution of the PoPi matrix is higher in the area of 90 °, and lower in the area of 0 ° and 180 °.

2.6.4 Evaluation of the PoPi matrix and the tracker

For human observers it is relatively easy to evaluate a graphical representation of the PoPi matrix. Automatic approaches are more demanding. In the course of this thesis a basic evaluation module has been developed. Research has shown that it is only capable of detecting a single speaker. Further investigations will be necessary in order to improve the quality.

The PoPi algorithm delivers a 521×360 sized matrix (2.19), representing frequencies between 80 Hz and 600 Hz in one dimension, and the circle (1 ° to 360 °) for the azimuth DOA in the other dimension. The evaluation algorithm sums up all frequency elements of the PoPi matrix

$$\mathbf{x} = x_j = \sum_{i=1}^{521} \hat{x}_{ij} \;, \qquad (\hat{x}_{ij}) = \hat{\mathbf{X}} \;. \tag{2.23}$$

Hence the vector $\mathbf{x}$, representing the energy intensity at a certain DOA, is the basis for the detection process.

The first decision to make is, whether a speaker is active or not in the present frame. Thus, a threshold parameter for the actual speaker scenario is needed. This parameter needs to be trained before the algorithm can be executed. The current implementation determines the threshold as a fraction of the maximally available energy intensity. Be-

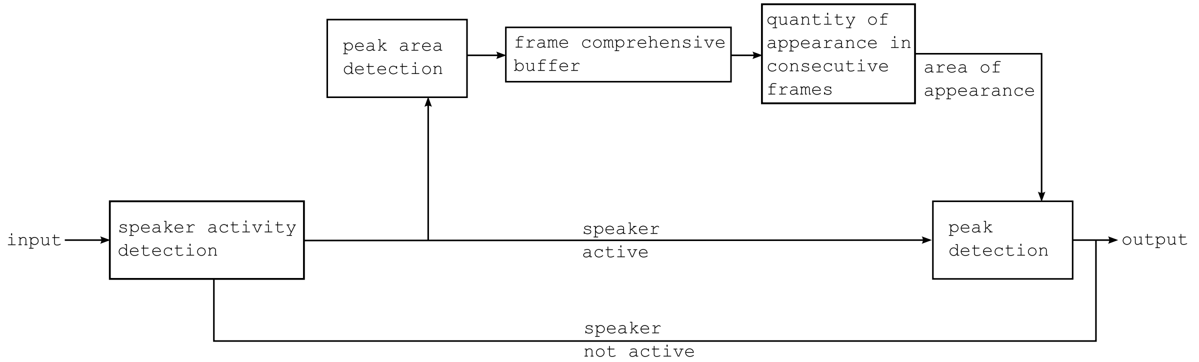


Figure 2.12: Processing chain of the tracker.

cause the absolute energy intensity is only necessary for thresholding, $\mathbf{x}$ is normalized afterwards

$$\overline{\mathbf{x}} = \overline{x}_j = \frac{x_j}{\arg\max_{k=1\ldots360} x_k} , \qquad j = 1\ldots360 . \tag{2.24}$$

Peaks in the vector $\overline{\mathbf{x}}$ which exceed a certain level are marked as active speaker areas, and are inherited to the tracking.

The tracker analyzes results from consecutive frames. The number of frames which are considered at the analysis can be varied. If L frames are considered by the tracker, a certain DOA has to appear in more than $L/2$ frames to be taken into account at the evaluation. A 10° range for each of the tracker's detected DOAs is considered at the actual peak detection, where the maximum of the current frame is searched for and the particular DOA detected. The 10° range is based on the width of two side by side standing speakers at a distance of approximately 1.5 m to the origin of the microphone array. Figure 2.12 shows the processing chain.

For the thresholding approach of the tracker, training is necessary in order to get meaningful results. To avoid the training, another tracker has been developed where thresholding is not used. The maximal magnitude of the PoPi matrix is tracked over consecutive frames. If a speaker is active, the angular DOA does not change randomly. If a certain DOA appears many times within a certain time interval, it can be assumed that a speaker is positioned at this DOA. The frequency of occurrence for certain DOAs is evaluated. The advantage of this approach is that no training is necessary. The constraint is that only single sources can be located.

2.7 Interaction between parts of the source localization algorithm

Parallelism is important in order to find efficient realizations for a reconfigurable hardware based implementation. This section describes the determination of parallel structures of the PoPi based source localization algorithm.

Figure 2.13 shows the assembly plan if the CC is performed in the time domain. Exploring parallel structures shows that Filterbank–CC–PoPi channel strips for different microphone pairs are independent of each other and may be computed in parallel. The CC is the most expensive computational operation (cf. Figure 3.2). This is important knowledge, as channel strips can be computed on different arithmetic units.

Figure 2.14 computes the CC in the frequency domain (GCC (2.18)). In this case a Fourier Transform with prior windowing is necessary before the CC, and an inverse Fourier Transform is necessary after the CC. After the inverse Fourier Transform and before the PoPi decomposition, results around the middle sample of the GCC vector are replaced by $\mathrm{GCC}_{\mathrm{PHAT}}$ results (cf. section 2.6.2). The parallel structure is nevertheless maintained for the frequency domain implementation. Computations can be carried out on different arithmetic units.

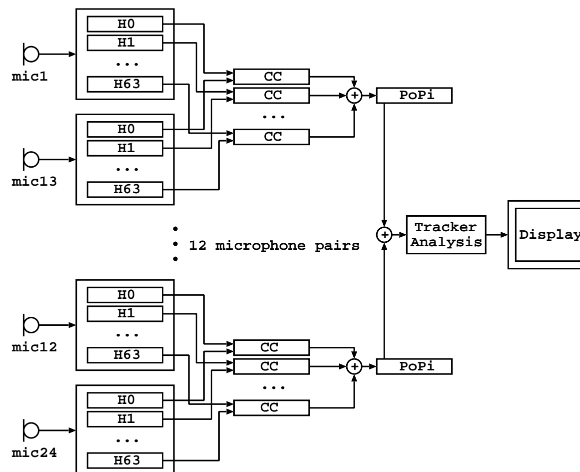


Figure 2.13: Structure of the source localization algorithm, if the CC is performed in the time domain. mic1 – mic24: Microphones of the circular microphone array; H0 – H63: Filter channels of the Gammatone filterbank;

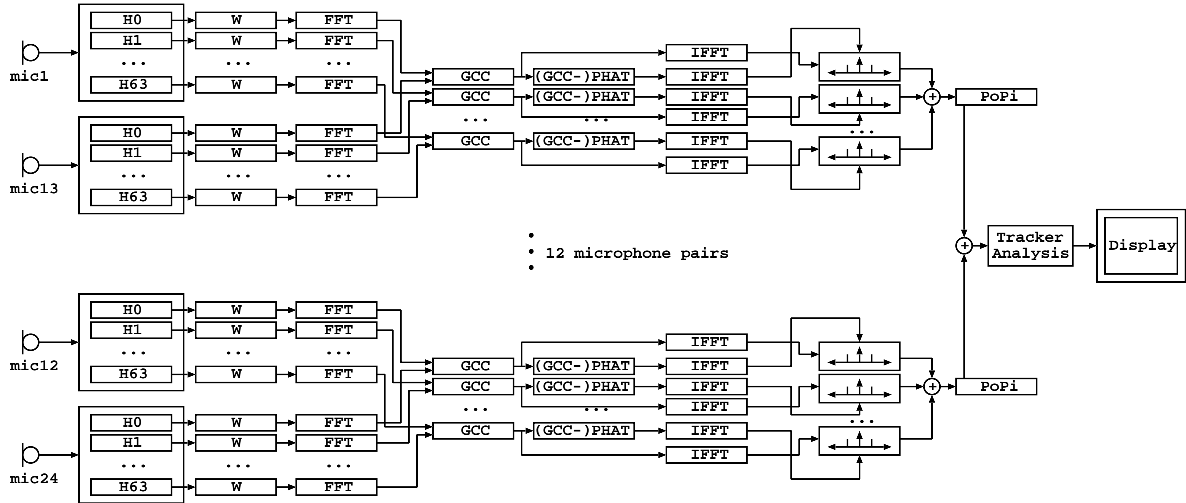


Figure 2.14: Structure of the source localization algorithm, if the CC is performed in the frequency domain. mic1 – mic24: Microphones of the circular microphone array; W: Hamming window; H0 – H63: Filter channels of the Gammatone filterbank;

2.8 The frame mechanism and real-time requirements

The source localization algorithm has to be computed as soon as a certain amount of data (referred to as *frame size*) is available at the outputs of the A/D converters of the microphones.

Determining the optimal frame size is essential for the analysis. If the frame size is too small the DOA can not be obtained, because the frames of the two microphones do not have correlating samples. In this case the first peak in the CC result vector does not appear (cf. section 2.6.3 and Figure 2.8). The frame size should not be too large either, because computational complexity – of the time domain CC algorithm – rises quadratically with an increasing frame size. In order to get continuous results it is necessary to overlap the frames. The time which elapses between the start of two consecutive frames is called the *frame shift*. Because frame size and frame shift are specified as time values, they are both dependent on the sampling frequency. A higher sampling frequency results in larger frames and more computational effort.

Figure 2.15 shows the principle of the frame mechanism, which is used synchronously for all microphones involved in the source localization algorithm. After splitting the audio stream into frames with a fixed frame size N, subsequent computations can be performed.

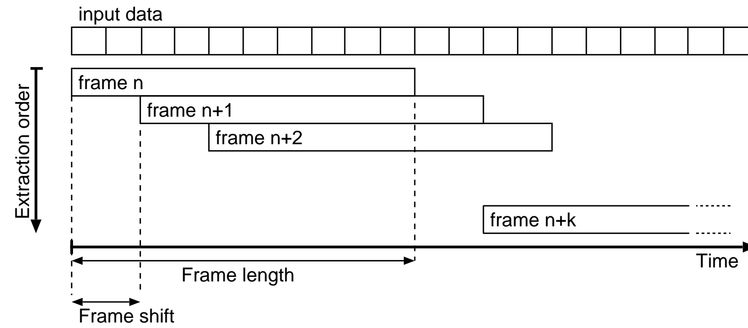


Figure 2.15: The frame mechanism: The input signal data stream is split into frames of a certain length (frame size). Frames overlap, the time elapsing between two consecutive frames is the frame shift.

The real-time requirement has no fixed, general definition. It has to be defined for the PoPi based source localization algorithm, and is related to the frame shift. By the time the input data completing a frame n is available, the computation of the source localization algorithm is started. The computation has to be completed until data for the computation of the next frame $n + 1$ is available in order to avoid interferences. The time between frame n and frame $n + 1$ is the frame shift. Therefore the source localization result has to be computed within the duration of the frame shift. If this requirement is not fulfilled the algorithm is not capable of delivering results in real-time.

3 Performance analysis and optimization of the algorithm

Investigations concerning the time performance of different PoPi based source localization algorithm approaches are presented in section 3.1 of this chapter. It will be shown that an optimization of the single parts of the algorithm is not sufficient to perform calculations in real-time. Therefore, the change in accuracy by varying several parameter of the algorithm are discussed in section 3.2. If it is possible to reduce the parameters complexity without loosing accuracy the execution time can be reduced further.

Since Matlab is hardware independent and offers comprehensive analysis tools, it has been used for experiments in this chapter.

3.1 Time performance of different implementations

Different Matlab software approaches for the time consuming parts of the algorithm result in significantly different execution times. Three parts of the speaker detection algorithm are the most extensive:

- Gammatone filterbank
- Cross correlation
- PoPi decomposition

The Gammatone filterbank is available as a vector oriented Matlab code, as well as a MEX subroutine written in C. Matlab provides an inherent CC function (*xcorr*), which executes computations in the frequency domain. Additionally, a time domain approach programmed as a MEX subroutine has been implemented. Three implementations are available for the PoPi algorithm: A loop based detection algorithm, a vector oriented approach, and a MEX subroutine.

The simulations took place on an "Intel Core i5-650 Processor (4MB Cache, 3.20GHz)". In order to explore the Matlab inherent multi threading possibilities, simulations have been carried out once in *single thread* mode and once in *multi thread* mode. In multi thread mode as many threads as arithmetic units available at the processor are started by Matlab, which is two for the processor used. In order to be able to compare results, and as a consequence of the available hardware resources, time performance results for the *explicitly parallel* approach have been carried out consecutively and extrapolated. The following approaches have been tested:

	Gammatone filterbank	Cross correlation	Position-Pitch decomposition
Matlab			
A	MEX	xcorr	MEX
B	Vector based	xcorr	Vector based
C	Vector based	xcorr	Loop based
D	MEX	MEX	MEX
Matlab runtime environment			
E	MEX	xcorr	MEX
Using explicitly parallel threads by calling Matlab via shell script			
F	MEX	xcorr	MEX

Table 3.1: Programming methods for Matlab code approaches. The letters in the left-hand column denote the individual software configurations in the rest of this chapter.

All simulations are performed with invariant algorithm parameters, listed in the following Table 3.2:

Resolution of input data	16 bit
Number of microphone pairs	12
Sampling frequency	48 kHz
Frame size	0.1 s
Frame shift	0.02 s
Filterbank frequency range	50 Hz – 8 kHz
Number of filterbank channels	64
Length of tracker	40

Table 3.2: Parameters for time performance simulation.

Results of the performance tests are depicted in Figure 3.1. The eye-catching approach **D** computes the CC in time domain. Apparently there are very efficient Fourier Transform implementations available, so the frequency domain approaches are more efficient, especially if they use the FFTW framework [18] as Matlab does. Approaches **A**, **B**, **C**, **E** and **F** are using the same frequency domain based CC. Variations in the Gammatone filterbank and PoPi decomposition parts (approaches **A**, **B**, **C**) show that MEX subroutines (**A**) are faster than vector based solutions (**B**), which are again faster than loop based Matlab routines (**C**).

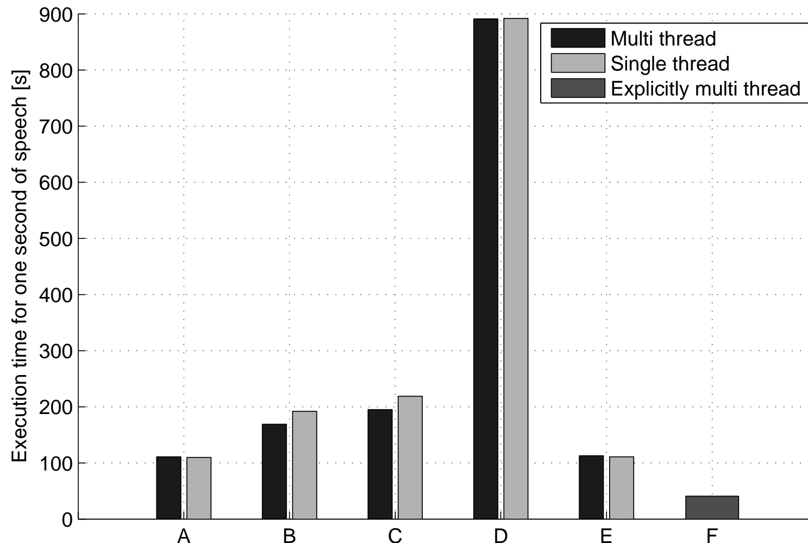


Figure 3.1: Time performance measurements using different programming methods (Annotations are explained in Table 3.1).

Multiple thread adjustments do not affect MEX implementations (**A**, **E**, **F**), as Matlab is only aware of parallelizing Matlab inherent functions. Approach **E** is compiled as stand alone executable. The compilation process uses the Matlab compiler (MCC) and at execution time Matlab runtime libraries are also necessary. Contrary to expectations the stand alone compilation is not faster than approach **A**.

Approach **F** separates the initialisation, the computation of each microphone pair channel (twelve channels), and the terminal analysis stage. Computations of the microphone pair channels are executed in parallel on individual CPUs. A shell script controls the execution chain. The independence of single microphone pair channels is beneficial in regard to parallel computing (cf. section 2.7), and allows for easy and efficient code separation, which is reflected in a significantly shorter execution time.

Matlab's profiling tool allows to take a closer look at the distribution of time duration used at certain parts of the code in approaches **A**, **B**, **C** and **D** (Figure 3.2). In approach **A**, most of the time is spent at the CC followed by the Gammatone filterbank and the PoPi decomposition. Approach **B** and **C** demonstrate the sizable computational effort

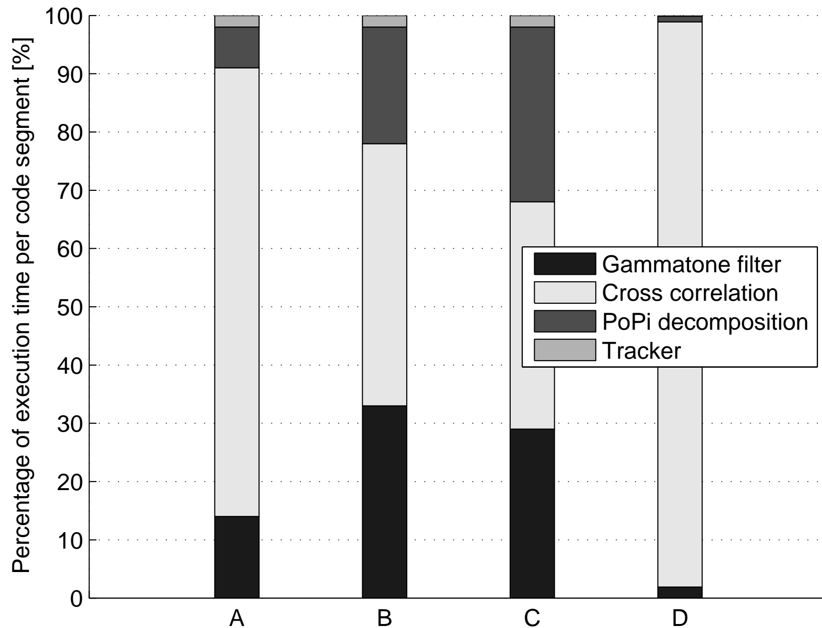


Figure 3.2: Relative time consumption in percent for the Gammatone filterbank, the cross correlation, the PoPi decomposition, and the tracker (Annotations are explained in Table 3.1).

when not using MEX subroutines. As shown by the results of approach **C**, loop based programming should be avoided and replaced by vector based solutions. Performing the CC in the time domain exorbitantly increases the computational costs for this section of code. The measurements in Figure 3.2 are performed with the single thread adjustment.

Figure 3.1 shows that approaches are not able to be carried out in real-time. Nevertheless, an increase in computational efficiency could be achieved. Investigations in the next section will analyze the acceleration in execution time when changing parameters of the algorithm.

3.2 Accuracy vs. time performance

This section provides insight into the speaker localization algorithm in regard to accuracy and time performance when varying single parameters such as: Frame shift, frame size, sampling frequency, filterbank adjustments, resolution of the input data, number of microphone pairs, and length of the analysis tracker.

Computational performance is measured as the duration of time needed to calculate the data for one second of recorded speech. If the result of the variously adjusted algorithm is within $\pm 5^\circ$ of the real result, the output is marked as accurate. The level of accuracy states the percentage of correctly detected frames within the analyzed voice recording.

3.2.1 Frame shift

The procedure of analyzing parameters of the algorithm is performed by varying only single parameters, whereas the other parameters are fixed. Adjustments are listed in the following table 3.3:

Fixed parameters		Varied parameters
Frame size	85.3333 ms	Frame shift
Sampling frequency	48 kHz	Length of tracker
Filterbank frequency range	50 Hz – 8 kHz	
Number of filterbank channels	64	
Resolution of input data	16 bit	
Number of microphone pairs	12	

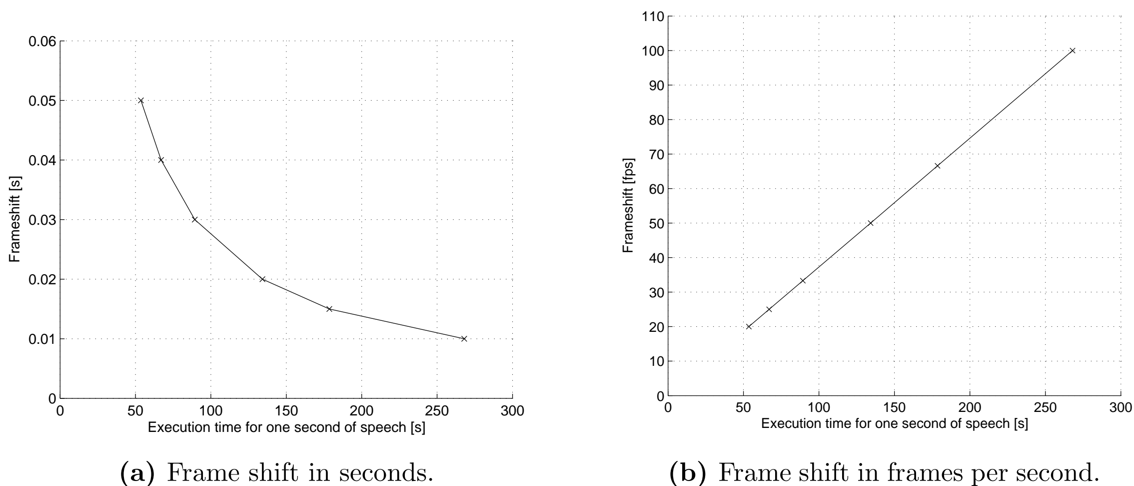
Table 3.3: Frame shift: Simulation parameters.

Beginning with the frame shift, it can be observed in Figure 3.3a that in correspondence to decreasing time-lags between the consecutive frames the execution time increases. The time performance curve follows a simple rational function $f(x) = 1/x$. The inversion of this function is a linear relationship between the frame shift (in frames per second [fps]) and the computational speed (Figure 3.3b). The result is expected as a doubling in frames to calculate, causes a doubling of computational complexity.

The quality of the PoPi matrix is not affected by variations in the frame shift. Therefore, the accuracy over different frame update intervals should stay the same. Figure 3.4 inspects this behaviour. In Figure 3.4a the accuracy diminishes with increasing frames per second. This effect is caused by the fixed tracker length. Fixed tracker length means that the evaluation algorithm always uses the result of the actual and seven elapsed frames. At an update rate of 100 fps, eight frames are only the 8/100 part of one second, whereas at an update rate of 20 fps, eight frames are 8/20 parts of a second. Therefore the tracker has more information over time available (e.g. the pauses between two spoken words are better covered). Adapting the tracker length with an increasing number of frames removes this effect. The overall accuracy stays the same (Figure 3.4b).

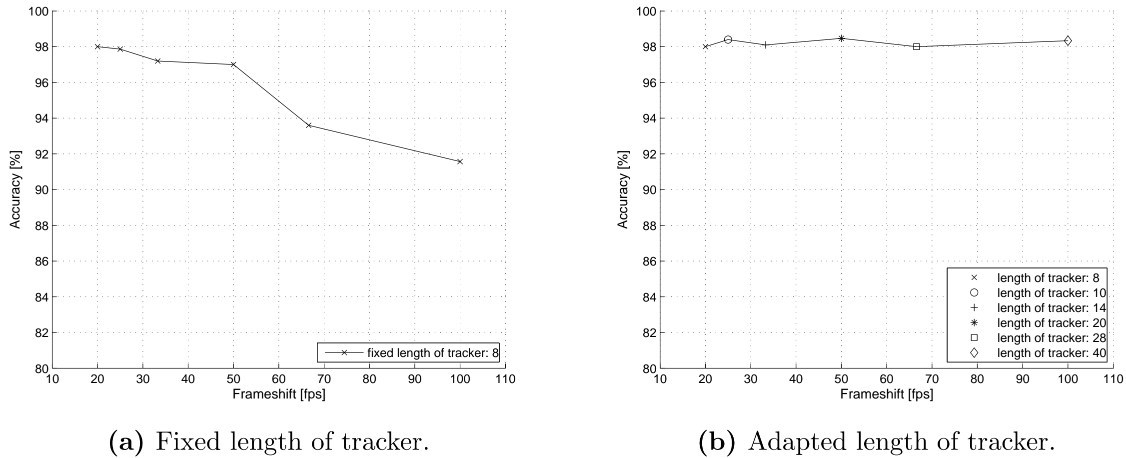
3.2.2 Frame size

Modifications in the frame size mean that the algorithm is considering more audio samples for one frame. Therefore pauses between words or phrases are less troublesome for



(a) Frame shift in seconds. **(b)** Frame shift in frames per second.

Figure 3.3: Time performance: Variation of frame shift.



(a) Fixed length of tracker.

(b) Adapted length of tracker.

Figure 3.4: Accuracy vs. time performance in regard to the frame shift.

the speaker detection, although the beginnings and endings of speaker active parts are more blurred.

Fixed parameters		Varied parameters
Frame shift	20 ms	Frame size
Sampling frequency	48 kHz	Length of tracker
Filterbank frequency range	50 Hz – 8 kHz	
Number of filterbank channels	64	
Resolution of input data	16 bit	
Number of microphone pairs	12	

Table 3.4: Frame size: Simulation parameters.

An increase in frame size goes hand in hand with an increase in execution time. The relationship is linear. This linearity suggests that the calculation of the CC (as shown in Figure 3.2, execution time is mostly needed during the CC) is performed in the frequency domain, where the amount of multiplications grows linearly with the length of the input vectors. Performing the CC in the time domain would quadratically increase the amount of multiplications in reference to the length of the input vector (cf. section 2.6.2).

Figure 3.5 shows the dependencies of accuracy and time performance with varying frame sizes. Two different datasets have been used: Dataset A is a recording of a single professional male speaker without any speaking breaks. In dataset B, two non-

professional speakers (one male and one female) are talking one after another with short breaks in between.

The detection works considerably better with the professional speaker, and increases with the growing frame size. Beginning with a frame size between 100 ms and 140 ms, saturation in accuracy can be observed. Exploring the differences between tracker lengths used for computing dataset A leads to the conclusion that for small frame sizes longer tracking is to preferable, whereas for large frame sizes an increased length of the tracker does not lead to more accurate results, because boundaries between speaker active and speaker inactive areas are softened.

3.2.3 Sampling frequency

Performance and accuracy observations have also been carried out with an alternating sampling frequency.

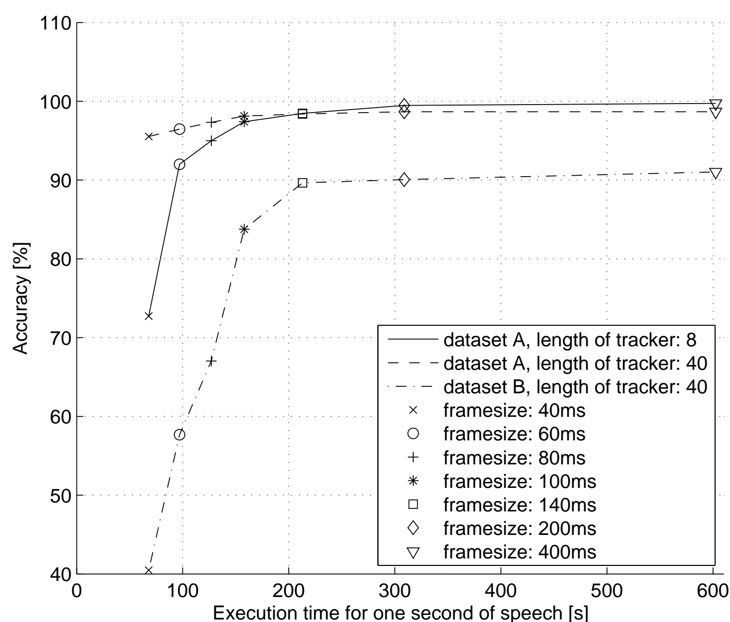


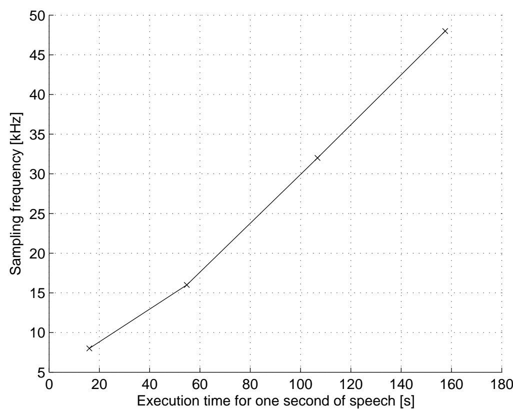
Figure 3.5: Accuracy vs. time performance in regard to the frame size.

Fixed parameters		Varied parameters
Frame shift	20 ms	Sampling frequency
Frame size	100 ms	Filterbank frequency range
Resolution of input data	16 bit	Number of filterbank channels
Number of microphone pairs	12	Length of tracker

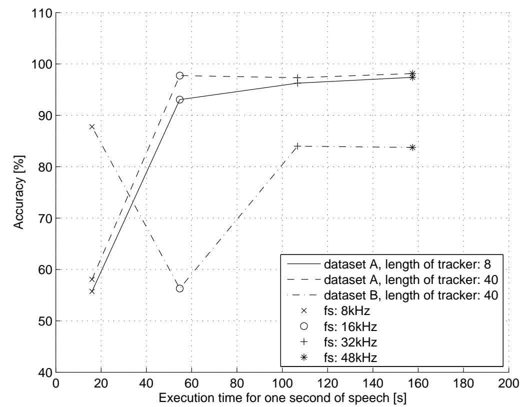
Table 3.5: Sampling frequency: Simulation parameters.

Figure 3.6a shows the relationship between increasing sampling frequency and the resulting increase in execution time. The relationship is linear. The 8 kHz case is faster than expected. This is a consequence of the reduced number of Gammatone filter channels (32 instead of 64 channels; Nyquist frequency: 4 kHz).

Accuracy in dependence of execution time (Figure 3.6b) shows expected results for dataset A. Accuracy is low in the 8 kHz case and higher for 16 kHz, 32 kHz and 48 kHz. As the highest center frequency of the filterbank is at 8 kHz, there are no improvements for sampling frequencies of 32 kHz and 48 kHz. On the contrary, results from dataset B are showing unexpected behaviour. The results for a sampling frequency of 8 kHz are very accurate, whereas the results for the 16 kHz case are very poor. Looking at result matrices from the PoPi decomposition suggests that a reduced resolution of the filterbank at higher frequencies leads to a more stable result for the 8 kHz case.



(a) Performance



(b) Accuracy vs. time performance

Figure 3.6: Accuracy vs. time performance in regard to the sampling frequency.

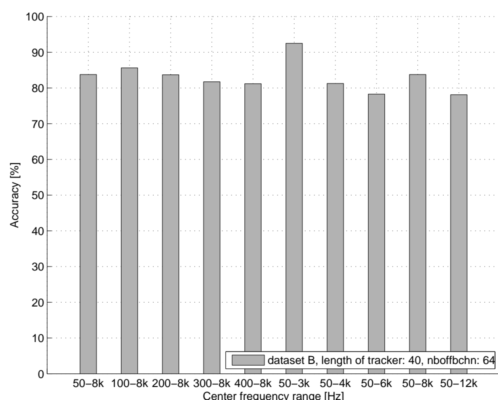
3.2.4 Filterbank adjustments

For higher sampling frequencies results are not getting more accurate because of the center frequencies range of the filterbank. Therefore, this section analyzes adjustments of the filterbank.

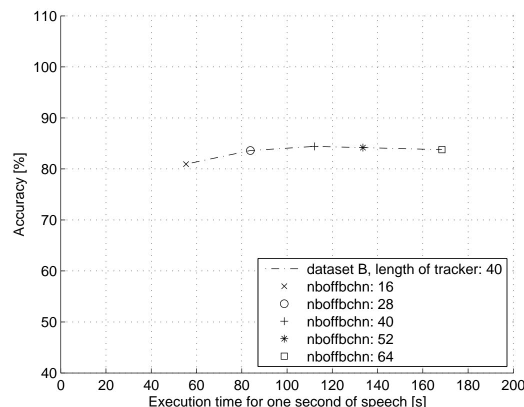
Fixed parameters		Varied parameters
Frame shift	20 ms	Filterbank frequency range
Frame size	100 ms	Number of filterbank channels
Sampling frequency	48 kHz	
Resolution of input data	16 bit	
Number of microphone pairs	12	
Length of tracker	40	

Table 3.6: Filterbank adjustments: Simulation parameters.

The results from different center frequency ranges are shown in Figure 3.7a. Reducing the upper limit of the center frequencies to 3 kHz, and lifting the lower limit of the center frequencies to 100 Hz leads to higher accuracy. Figure 3.7b depicts results from a reduced number of filterbank channels (Range of center frequencies: 50 Hz – 8 kHz). Simulations reveal that 28 channels lead to the same level of accuracy as 64 channels. It is possible to reduce the number of filterbank channels, with the benefit being saved computational load. For simulations of Figure 3.8, the center frequencies range of the filterbank has been reduced to 100 Hz – 3 kHz. Simulations have again been performed for different numbers of filterbank channels. The number of channels can be reduced to 10 without a significant loss of accuracy. The reduction of filterbank channels goes hand in hand with a tremendous acceleration in execution time.



(a) Accuracy with varied filterbank frequency range



(b) Accuracy vs. time performance: Varied number of filterbank channels. Filterbank frequency range: 50 Hz – 8 kHz.

Figure 3.7: Accuracy and time performance in regard to filterbank adjustments.

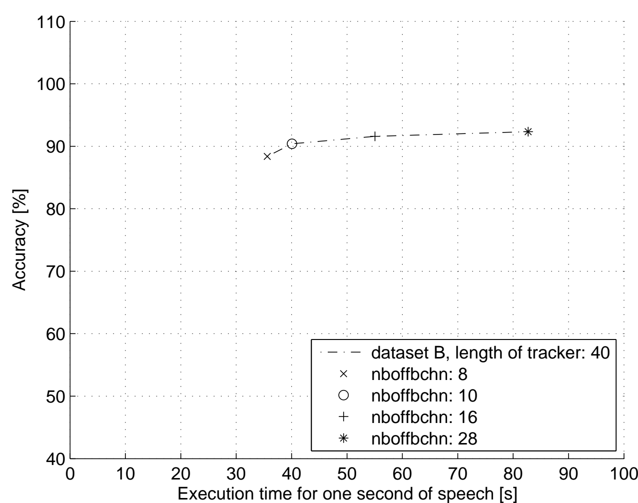


Figure 3.8: Accuracy and time performance in regard to optimized filterbank adjustments. Filterbank frequency range: 100 Hz – 3 kHz.

3.2.5 Resolution of input data

Matlab is working internally with double precision floating point numbers. Therefore differences in input data resolution can only vary accuracy, not time performance. Simu-

lation results show (Figure 3.9) that a resolution of 16 bit is both necessary and sufficient.

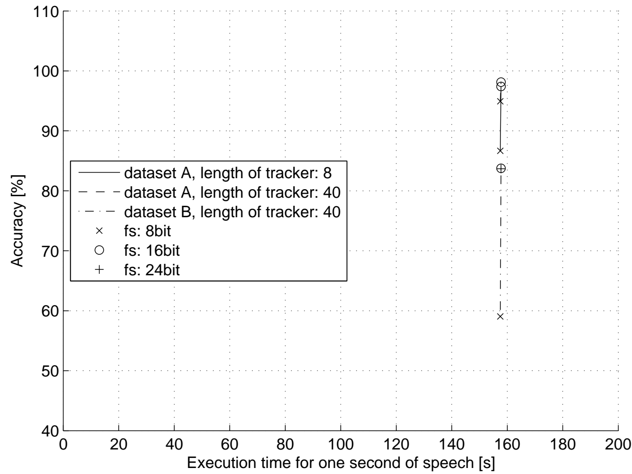


Figure 3.9: Accuracy vs. time performance in regard to the resolution of the input data.

3.2.6 Number of microphone pairs

Changing the number of microphone pairs linearly increases the execution time for one second of speech.

Fixed parameters		Varied parameters
Frame shift	20 ms	Number of microphone pairs
Frame size	100 ms	Length of tracker
Sampling frequency	48 kHz	
Filterbank frequency range	50 Hz – 8 kHz	
Number of filterbank channels	64	
Resolution of input data	16 bit	

Table 3.7: Number of microphone pairs: Simulation parameters.

Depending on the dataset, results begin to increase in accuracy with at least four pairs. Using only one pair of microphones results in a symmetric PoPi matrix, and it is not possible to determine the speaker's location (*mirrored source*, cf. section 2.3). Increasing the number of microphone pairs adds the real source of the source–mirrored source pair in the PoPi matrix and suppresses the mirrored source. Therefore the ratio between the real and the mirrored source rises with each additional microphone pair.

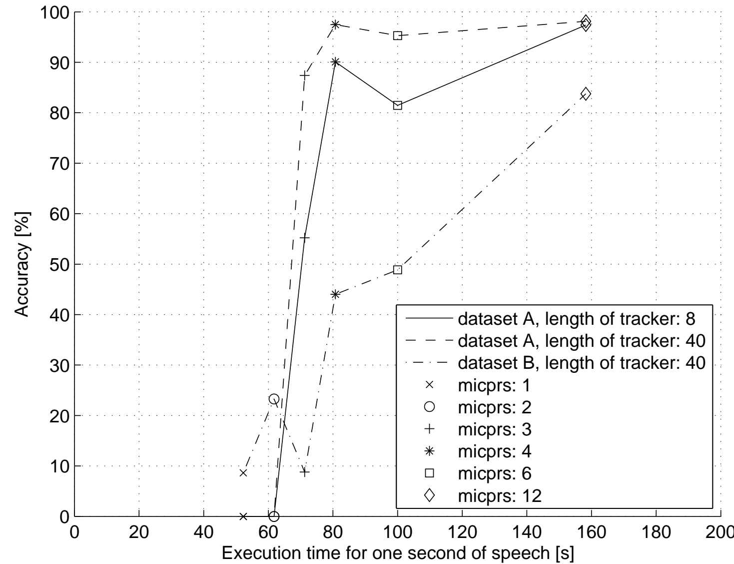


Figure 3.10: Accuracy vs. time performance in regard to the number of microphone pairs.

As there is no inactive speaker sequence in dataset A, thresholding does not come into effect and speaker detection fails as a whole for one and two microphone pairs. Proper results for four and six microphone pairs of dataset A have not been expected after a visual interpretation of the PoPi matrix. Instead an increase in accuracy has been expected such as for the results of dataset B.

Summary

In the preceding sections the effects on time/quality performance by changing parameters of the source localization algorithm have been analyzed.

Increasing the frame shift linearly increases the computational expenses. Accuracy is not affected if the length of the tracker is adapted. A shorter frame shift results in a shorter update interval for the DOA output of the algorithm. Together with real-time considerations in section 2.8 the frame shift has to be chosen appropriately.

According to simulations in Figure 3.5 a frame size between 100 ms and 140 ms is sufficient. Keeping in mind that the computational complexity is increasing quadratically for the time domain approach of the CC with increasing frame size the lower limit (100 ms) should be chosen.

The adjustments for the filterbank and the sampling frequency are connected. It is

not necessary to sample with a high frequency if the upper boundary of the filterbank center frequency range is low. The sampling frequency is also affecting the central and lateral resolution of the PoPi matrix (cf. section 2.6.3). The filterbank adjustments have to be considered in regard to the used tracker (cf. section 2.12), which is not analyzing the frequency component of the PoPi matrix in detail. For the actual implementation of the tracker it is possible to reduce the sampling frequency, the center frequency range of the filterbank, and the number of filterbank channels. These reductions go hand in hand with a significant reduction of computational expenses.

Simulations analyzing the resolution of the input data require a resolution of 16 bit. Simulations varying the number of microphone pairs are inconclusive for 4 and 6 microphone pairs. To be on the save side 12 microphone pairs should be used for computations.

Reductions of computational expenses as a result of varying the parameters of the PoPi source localization algorithm are significant (cf. Table 3.8). Nevertheless, it is still not possible to execute the algorithm in real-time. Therefore, the algorithm is ported to a hybrid reconfigurable CPU, where the most computational expensive part of the algorithm – the CC – can be computed in parallel. Chapter 4 introduces the hybrid platform, chapter 5 presents investigations concerning the implementation of the CC on the hybrid platform.

Parameters	
Frame shift	40 ms
Frame size	100 ms
Sampling frequency	16 kHz
Filterbank frequency range	100 Hz – 3 kHz
Number of filterbank channels	10
Resolution of input data	16 bit
Number of microphone pairs	12
Length of tracker	40
Execution time	
7.55 s for one second of speech	
Accuracy	
89.2 %	

Table 3.8: Simulation with the optimized parameter configuration.

4 Stretch S6 hybrid reconfigurable CPU based development

This chapter introduces the hardware platform which has been used to proof the concept of accelerating parts of the source localization algorithm on reconfigurable hardware.

In contrast to a pure FPGA based solution a hybrid hardware structure has been used. The Stretch processing unit has both a classical Arithmetic Logic Unit (ALU), referred to as "Xtensa", and a reprogrammable FPGA unit, referred to as "Instruction Set Extension Fabric" (ISEF). On the ISEF it is possible to parallelize computationally expensive operations. [19] [20] [1] [2]

4.1 Overview of the S6 Xtensa ISEF board

This section describes the Stretch S6 PCIe DVR Add-in VRC6416 card that was used for this thesis. Figure B.1 shows a picture of a similar VRC6016. Figure B.2 additionally lists chip identification numbers. Figure B.3 gives the block diagram of the board. The core of the board is the Processor Array (PA). It consists of three S6105 and one S6100 Processor Entity (PE). The S6100 processor has extended features compared to the S6105, which makes it the master of the PA.

On the left side of the PEs in Figure B.3, A/D converters (Techwell TW2864B) are connected to each processor. The A/D converters are for video and audio purposes, but only the audio paths are needed in the application. The main memory of each PE appears on the right side of Figure B.3. Each PE has 128 MB of main memory. Figure B.4 focuses on the processor. The main blocks that have been used for the application

[1] Parts of this chapter have been developed in cooperation with Boris Clénet and appear in his master thesis too. [1]

[2] Knowledge concerning the design and functionality of the Stretch environment, has been acquired from the Stretch manual. The manual is part of the Stretch Integrated Development Environment (IDE). [21]

are the DDR2 controller and the S6SCP Engine. The Quad Dataport (which is for video applications only), the Low-Speed Peripherals, the Enhanced Generic Interface Bus (eGIB), and the Ethernet Media Access Controller (GMAC) have not been used.

4.2 The ISEF Xtensa processor

The Xtensa LX Dual-Issue VLIW processor features three parallel data paths, as shown in Figure 4.1. The Floating Point Unit (FPU) with its corresponding register (FR) and the Arithmetic Logic Unit (ALU) with its register (AR) are the two *classical* ones. The Instruction Set Extension Fabric (ISEF) with the ISEF RAM (IRAM) and the Wide Registers (WRs) is the reprogrammable part of the chip.

Normal C functions are executed either on the ALU or on the FPU, whereas time critical or costly parts of the code can be outsourced to the ISEF where specific calculations are performed in parallel. There are several possibilities for providing data to the three functional unit (FPU, ALU, ISEF).

- When using the ALU or FPU, data can either be transferred over the Data-Cache (D-CACHE) or over the Dual Port RAM (DATARAM).
- When using the ISEF, data can either be transferred over D-CACHE/WRs, DATARAM/WRs, or IRAM.

Direct Memory Access (DMA) enables the IRAM to be filled directly from the main memory. Otherwise data loading has to be accomplished by the WRs. It is also possible to use the Extension Register (ER) from inside the ISEF. Intermediate results can be stored there, but no access from outside the ISEF is possible.

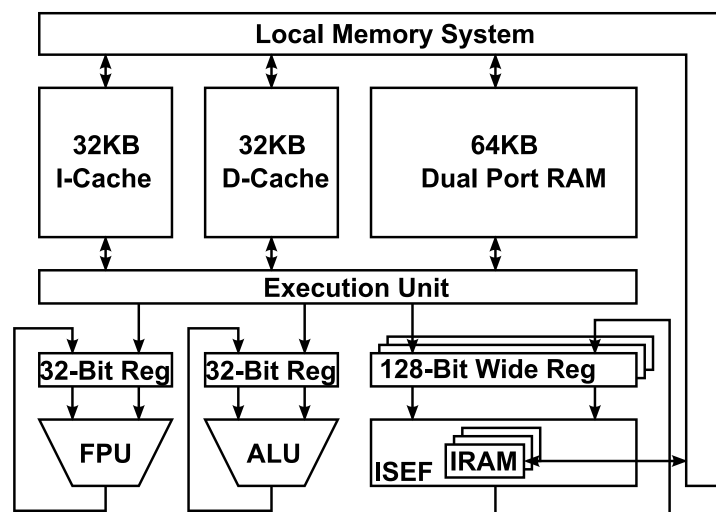


Figure 4.1: Organization of the Xtensa.

4.3 S6 Instruction pipeline structure

The Xtensa processor uses a five stage instruction pipeline. It is possible to launch at most one instruction in the pipeline per CPU cycle. The pipeline allows for efficient use of existing instruction latency. The stages are listed in Table 4.1. Figure 4.2 shows the schedule of the pipeline.

I	Instruction fetch
R	Register file read and instruction decode
E	Execute
M	Data cache read
W	Register file write
EUn	Extension Unit cycles

Table 4.1: Pipeline stages.

An ISEF configuration results in n extra cycles known as Extension Unit cycles (EUs). These cycles are depicted on the right side of Figure 4.2. The ISEF configuration gives rise to a pipeline extension. It is possible to add up to 27 EUs to the normal pipeline stages.

If dependencies between consecutive instructions exist, *stalls* appear in the pipeline. Avoiding these time-consuming stalls requires careful programming. Figure B.5 illus-

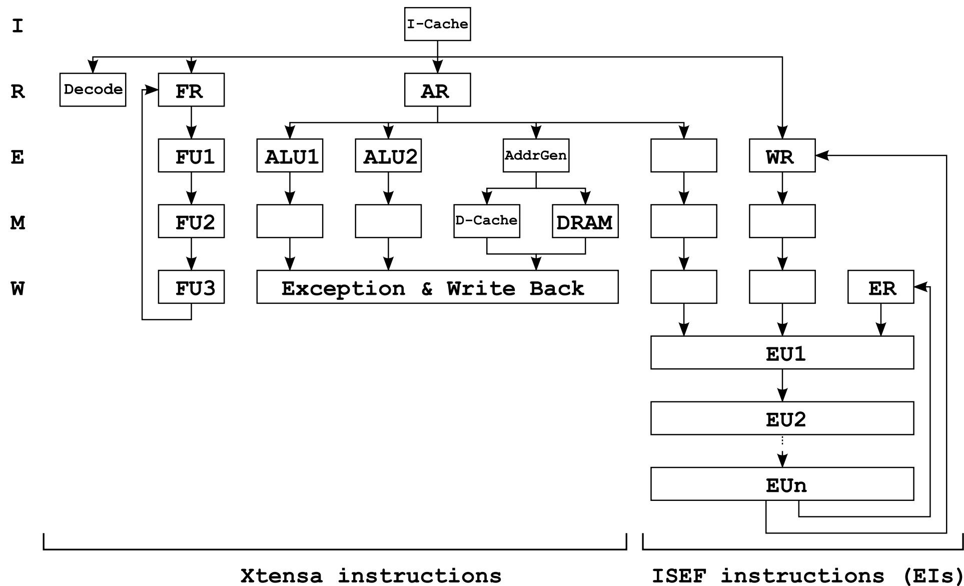


Figure 4.2: The extended pipeline structure.

trates the effect of dependencies. At address 0x4001c0db, the instruction `wraputi` requires the result of the ISEF's Extension Instruction (EI) at 0x4001c0cd. The EI is not finished when its result is needed by the `wraputi` instruction, and therefore the whole pipeline is stalled for 13 cycles.

4.3.1 Issue Rate

It is important to notice that the Xtensa and the ISEF do not necessarily run at the same frequency. The cycles of the ALU and the ISEF are linked by the *Issue Rate* (IR). It is a number which gives the ratio between the running frequencies of the Xtensa and the ISEF.

For example, an IR of 1 means that the ISEF issues an instruction every time the ALU issues one. An IR of 3 means that the ISEF issues an instruction at the beginning of every group of three instructions from the ALU. The default value for the IR is 1. If the compiler can not reach the target frequency, it indicates the achieved frequency in a report file (cf. section 4.6). The IR then has to be increased so that the target frequency for the ISEF is achievable.

4.4 S6 Programming

The Stretch Software Configurable Processor (SCP) is programmable in ANSI-C, although there are special hardware-related parts of the code (especially those that command the ISEF) which have to be programmed in Stretch-C. Stretch-C varies from ANSI-C. There are additional, more width-flexible data types and certain hardware-related functions, definitions, and intrinsics.

4.4.1 Defining and using Extension Instructions

Apart from the standard ANSI-C files (∗.c), Stretch-C files (∗.xc) are used to define EIs. The following code example demonstrates the construction of an EI.

```
1  #include <stretch.h>
2  static se_sint<64> sumver[4];
3  SE_FUNC void CROSSONISEF
4      (SE_INST CC_MAC, SE_INST CC_INIT_MAC, SE_INST CC_FIN_MAC,
5      WRA A, WRB B, WRA *Y_1, WRB *Y_2)
6  {
7      se_sint<16> b;
8      ....
9      *Y_1 = (sumver[1], sumver[0]);
10 }
```

In line 1, the preprocessor includes the standard stretch library for Stretch-C files. It is necessary to use the WR data type, as well as all of the specific instructions. Line 2 defines a static array. If a variable is defined as static, it is stored in the ERs. The data type `se_sint<64>` defines a signed integer with a width of 64 bit.

Lines 3, 4, and 5 define an EI. The keyword `SE_FUNC` identifies the function as EI, and `SE_INST` gives it its name (i.e. how the function can be called from ordinary C code). Common parts of several EIs can share the same hardware resources of the ISEF. In this case, the instructions `CC_MAC`, `CC_INIT_MAC` and `CC_FIN_MAC` differ slightly from each other but the main part is nevertheless the same. Therefore, all three instructions are outlined in one function.

In total, each EI call can transfer data from 3 WRs to the ISEF and output data to 2 WRs. Line 5 defines four variables located in the WRs (2 input values, 2 output values). It is possible to tell the compiler which WR (A or B) should be used. Line 7 defines a local signed integer with a width of 16 bit. Data types `se_sint<n>` and `se_uint<n>` refer to signed and unsigned integers with a bit-width of n respectively. In line 9, the two return values (of width 64 bit) are stored in one WR (of width 128 bit).

4.4.2 Handling the wide registers

Figure 4.1 shows that the ISEF is accessible over the WRs, as well as over the later-discussed IRAM via DMA. In order to load the WRs with data to be processed on the ISEF, it is necessary to use a couple of special functions within the `*.c` file.

```
1  ....
2  WRGETINIT(0, p_x2);
3  WRGETOI(&wr_x1, 1);
4  ....
5  WRPUTINIT(0, p_acc);
6  WRAPUTI(wr_y_1, 4);
7  WRPUTFLUSH0();
8  WRPUTFLUSH1();
9  ....
```

Lines 2 and 3 enable data transfer to the WRs. Line 2 initializes the transfer from a memory place in the DATARAM to the WRs. The zero declares that the source pointer should be incremented after each access. Line 3 copies the data (1 byte) to the WR.

Lines 5 to 8 retrieve data from the WRs. Line 5 initializes the transfer from the WR to the destination located in the DATARAM. Again, the destination pointer should be incremented after each access. Line 6 copies 4 bytes of data from the WR to the destination. Lines 7 and 8 are also necessary in order to complete the data transfer.

4.4.3 Handling the IRAM

Figure 4.1 shows the ISEF inherent location of the IRAM. Two data paths to the IRAM are possible. The first one over the WRs is not efficient. The second one is more direct and preferable. DMA allows for direct transfer of data from the main memory to the IRAM.

```
1  *.c file:
2  se_iram_handle *hA;
3  hA = se_iram_get_handle(crossisef, A, SE_IRAM_ROW_MAJOR, 0);
```

The IRAM handle `hA` is an "access gate" to array A (line 3) in the IRAM (`crossisef` is the name of the ISEF configuration). If the array has several dimensions, one should specify along which dimension the increment is first done (with arguments `SE_IRAM_ROW_MAJOR` or `SE_IRAM_COL_MAJOR`). There are 32 banks of IRAM in total. One bank has a size of 2 kB, which leads to a total IRAM size of 64 kB. Each bank should only be accessed once per ISEF cycle, otherwise stalls of many cycles may occur. Therefore the distribution of

variables has to be wide spread over the banks. The following code example shows the possibilities defining variables within the IRAM structure.

```
1  *.xc file:
2  static se_sint<16> A[1024][8];
3  SE_MEM(A);
4  static se_sint<32> B[1024][4];
5  SE_MEM(B);
6  static se_sint<64> C[1024][2];
7  SE_MEM_LOCAL(C);
8  static se_sint<128> D[1024][1];
9  SE_MEM_LOCAL(D);
```

The maximum depth of an array is 1024 and only sizes by the power of two are possible. A group of 8 banks can be used in four different ways: the width of the data type can vary between 16, 32, 64 and 128 bit. Increasing the width of the data type decreases the second dimension of the array. After the desired array has been defined, it has to be mapped into the IRAM by using the intrinsic `SE_MEM` or `SE_MEM_LOCAL`. It is not possible to access the array over DMA with the latter.

4.4.4 DMA transfers and resolution

When using DMA, the resolution of the available audio data can be a problem since the IRAM can not handle 8 bit and 24 bit data types (cf. section 4.4.3). 16 bit and 32 bit tables are used instead. DMA transfer always considers amounts of bytes, and no information about data types is delivered.

In order to cast 8 bit data into IRAM's 16 bit tables when using DMA, the stride and skip mechanism is used (cf. section 4.4.5). In the IRAM a byte is skipped between each data-byte coming from the main memory. This is achieved using the following setup of the DMA channel.

```
1      (*p_ch1_conf).src_stride = 1;
2      (*p_ch1_conf).src_skip = 0;
3      (*p_ch1_conf).dst_stride = 1;
4      (*p_ch1_conf).dst_skip = 1;
```

`src_stride` and `dst_stride` are equal. They represent the number of data-bytes DMA transfers within one write operation. Line 2 informs the compiler that no bytes at the source's side are skipped. Line 4 tells that one byte will be skipped at the destination side after each write operation. Therefore, after one `stride` – `skip` operation each block of 2 Byte in the IRAM can be considered as a 16 bit-casted value.

The configuration is slightly different in regard to the second problem (24 bit). Here

data has to be casted into 32 bit values. Therefore a byte is skipped between each group of three bytes coming from the source.

```
1     (*p_ch1_conf).src_stride = 3;
2     (*p_ch1_conf).src_skip = 0;
3     (*p_ch1_conf).dst_stride = 3;
4     (*p_ch1_conf).dst_skip = 1;
```

`dst_skip` still skips one byte at the destination after one write operation has been performed.

4.4.5 The BIOS

The Stretch BIOS (SBIOS) provides the fundamental functionality for applications running on the S6000 family of processors. Its most important parts in terms of the thesis's applications are the DMA, the memory management, clock routines, some utility functions, and the data types. The SBIOS functions used are demonstrated in an example.

```
1  #include <sx-misc.h>
2  #include <sx-mm.h>
3  #include <sx-mmdma.h>
4  #include <sx-timer.h>
5  //create memory pools
6  static sx_int8 ddr_pool_space[567] SX_DDR;
7  static sx_int8 dram_pool_space [567] SX_DATARAM;
8  sx_mm_pool *ddr_pool;
9  sx_mm_pool *dram_pool;
10 ddr_pool = sx_mm_create(ddr_pool_space, sizeof(ddr_pool_space));
11 dram_pool = sx_mm_create(dram_pool_space, sizeof(dram_pool_space));
12 //allocate memory
13 p_samples_1 = sx_mm_zalloc(ddr_pool, framesize_100ms_inbytes);
14 p_frame_1 = sx_mm_zalloc(dram_pool, framesize_100ms_inbytes);
15 //initialize dma channel
16 sx_mmdma_chan *p_channel_1;
17 sx_mmdma_chan_config *p_ch1_conf;
18 p_ch1_conf = (sx_mmdma_chan_config *)sx_mm_zalloc(ddr_pool,
19         sizeof(sx_mmdma_chan_config));
20     (*p_ch1_conf).chan_num = 5;
21     (*p_ch1_conf).priority = 2;
22     (*p_ch1_conf).src_stride = 0;
23     (*p_ch1_conf).src_skip = 0;
24     (*p_ch1_conf).dst_stride = 0;
25     (*p_ch1_conf).dst_skip = 0;
26 init_ch1_error = sx_mmdma_chan_init(p_ch1_conf, &p_channel_1);
27 //count cycles, copy data
28 count1 = sx_get_ccount();
29 memcpy_ch1_error = sx_mmdma_memcpy(p_channel_1,
30         p_frame_1, p_samples_1, framesize_100ms_inbytes, 1);
```

```
31  while(sx_mmdma_get_num_pending(p_channel_1) != 0);
32  count2 = sx_get_ccount();
33  cycles1 = count2 - count1;
34  //close dma channel, free memory
35  close_ch1_error = sx_mmdma_chan_close(p_channel_1);
36  sx_mm_free(ddr_pool, p_samples_1);
37  sx_mm_free(dram_pool, p_frame_1);
```

The four necessary SBIOS ∗.h files are listed below:

sx-misc.h	Includes intrinsics and declares stretch data types.
sx-mm.h	Manages memory allocation.
sx-mmdma.h	Handles DMA.
sx-timer.h	Counts cycles in order to evaluate performance.

The available stretch data types are: `sx_int8`, `sx_uint8`, `sx_int16`, `sx_uint16`, `sx_int32`, `sx_uint32`, `sx_int64`, and `sx_uint64`. The numbers specify the bit width of the data type, and a "u" signifies that the data type is unsigned. In lines 6 and 7 the memory for two *memory pools* is reserved. Arrays of `sx_int8` are located in the main memory (`SX_DDR`) and the DATARAM (`SX_DATARAM`). The intrinsics `SX_DDR` and `SX_DATARAM` are provided by the file sx-misc.h.

After generating pointers to pools in lines 8 and 9, the memory pools have to be created in lines 10 and 11. Functions that create memory pools are defined in sx-mm.h. In lines 13 and 14 two variables of a certain length are initialized, each located in one pool (`zalloc()` allocates memory and initializes it with zeros).

The DMA channel now has to be initialized. First the necessary pointers (lines 16–18) and the configuration's specifications (lines 20–25) are created. The DMA channel-number can be chosen between 0 and 11 for the programmer's use (line 20). Line 21 sets the priority of each channel between 0 and 3 (0 is the highest one). The stride-skip mechanism in lines 22–25 provides the possibility to skip bytes in a recurrent pattern, either at the source or at the destination.

In line 26 the channel is initialized, and line 29 realizes the data transfer. A waiting period is imposed while the DMA transfer is pending (line 31). Afterwards line 35 closes the DMA channel. The processor cycles are counted between line 28 and line 32 and calculated in line 33. Finally, the reserved data pools have to be freed (line 37).

4.5 S6 Software build process

Figure 4.3 shows the building process for an ISEF configuration. There are two files: The ISEF configuration file (fir8.xc) and the corresponding (fir8.c) file from where the ISEF configuration is called.

The ∗.xc file needs the inclusion of stretch.h in order to use Stretch intrinsics. From the ISEF's configuration the Stretch-C Compiler (SCC) generates several files. The compilation process creates an ∗.xo object file and a ∗.xr report file.

```
1  scc -c -ms6100-3-iss fir.xc
2  scc -ms6100-3-iss -stretch-link-config fir8.xo
```

Secondly, the linker creates two additional files. libei.h (different names are possible by specifying the option -o calling SCC) has to be included in the ∗.c file, where the ISEF EIs are called from. The file libei.a is an archive file and is used for the linking process of the ISEF configuration and the ∗.c file. The linking process also requires the object code from the ∗.c file. It is generated by the SCC compiler.

```
1  scc -c -ms6100-3-iss fir8.c
```

The arising executable file can be started with the command st-run. Further compilation options are possible. The most important ones are listed in the following.

Firstly, several compilation modes are available which are called *native, simulation* and *remote*.

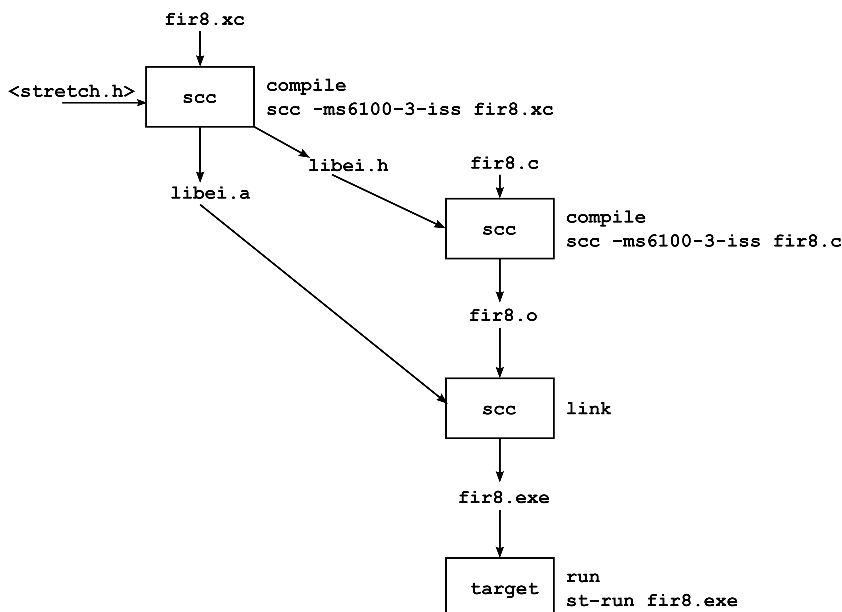


Figure 4.3: Building Process of an ISEF configuration.

-ms6-native	Compiles the EIs into native code for the host machine. Useful for executing and debugging EIs quickly before compiling them for the ISEF.
-ms6100-3-iss	Targets the S6SCP Instruction Set simulator for C/C++ and Stretch-C code.
-ms6100	Targets the S6100 and S6105 SCP processor for C/C++ and Stretch-C code.

The following commands ask for changes in default modes specificities.

-stretch-effort[0-10]	Sets the effort level for compiling EI into bitstreams. This affects the compilation of Stretch-C files only. In general, a higher effort level takes more time, but produces better ISEF resource usage.
-stretch-freq	Sets target frequency (in MHz). Default value is 300.
-stretch-issue-rate	Sets IR. Default value is 1.
-stretch-nobits	Compiles EI with no bit-file generation (cf. next section): Only information required by the Instruction Set Simulator (ISS) to run the EIs is produced.

The following commands control optimizations for advanced compilation handling.

-OPT[number]	Set optimization level to [number].
-OPT:alias = disjoint	Assume that memory references through different named pointers do not alias with each other, nor with any direct memory references.
-OPT:Olimit = size	Do not optimize functions that exceed the specified size.
-OPT:unroll = times	Do not unroll any loop more than the specified number of times. The default is 8, and unroll=1 disables loop unrolling.

4.5.1 Report files

The size of the ISEF is limited. There are 4096 units of Arithmetic Units (AU) and 8192 units of Multiplication Units (MU). The IRAM has 32 banks, each with 2048 Bytes of memory. The resource usage report of a configuration is located in the *.xr file. The final version of the report can be found at the end of this file.

```
1  /***************************************************\
2  *           Final resource usage report              *
3  *----------------------------------------------------*
4  * Configuration cross16:
5  * Total AUs                  =  3512 out of 4096
6  * Total MUs                  =  5120 out of 8192
7  * Total SHIFTs               =     0
8  * Total IRAMs                =     0 out of   32
9  * Total PRIENC bits          =     0 out of  256
10 * Target Issue Rate          =     1
11 * Target chip frequency      = 300.0 Mhz
12 * Target ISEF frequency      = 300.0 Mhz
13 * Achieved ISEF frequency    = 176.1 Mhz
14 * Maximum output write cycle =    11
15 * Warning: ISEF cannot run at the required frequency.
16 * Compile time               =  3263 seconds
17 \***************************************************/
```

The IR, the target chip and ISEF frequencies, and the execution cycles of the configuration are listed along with the resources. If the configured ISEF frequency does not meet the required frequency, a warning occurs and the achieved frequency is printed instead. The information is only available if a *bit-file* generation has been requested. Without the bit-file, values of the report file are rough approximations and there is absolutely no guarantee that the designed configuration will fit the ISEF.

4.6 Stretch integrated development environment

Stretch provides a graphical Integrated Development Environment (IDE) for code development. The IDE integrates several command line tools. For the design of efficient EIs the two most important features are the pipeline view and profiling.

4.6.1 Pipeline view

Information about possible stalls produced by an ISEF configuration could be found with the pipeline view (cf. Figure B.5). The lines of code in question are executed step by step by the debugger after the pipeline view of these lines is generated. If stalls occur, redesigning parts of the code or parts of the whole underlying model are a possible solution. Reducing stalls improves the execution speed of an ISEF configuration.

4.6.2 Profiling

Speed of execution is the main information about a configuration's performance. The process that measures the execution speed is called profiling (cf. Figure B.6). Profiling lists particular functions of the code with the number of taken cycles for each of them. Functions with a great number of cycles are preferred candidates to be implemented on the ISEF.

5 Porting the source localization algorithm to reconfigurable hardware

Chapter 3 analyzed the source localization algorithm in order to find an optimally adjusted set of parameters. It was not possible to execute the optimized implementation in real-time. In this chapter, parts of the PoPi based source localization algorithm are ported to reconfigurable hardware to minimize the execution time. The main focus is set on the core part of the algorithm – the CC.

As already mentioned in chapter 2, the CC operation is used to detect similarities between two signals. Knowing the relative position of two microphones and computing the CC between their outputs, it is possible to derive the DOA, the detected speaker's direction of emission.

Several CC approaches have been implemented on the ISEF resulting in different hardware configurations. After the different approaches are introduced, section 5.4 analyzes the performance of the configurations. [1]

5.1 Mathematical description of the CC

For two discrete signals $x[n]$ and $y[n]$ defined $\forall n \in \mathbb{Z}$, the CC is defined as [13]:

$$\Phi_{xy}[n] = \langle x^*[m]y[m+n] \rangle = \sum_{m=-\infty}^{\infty} x^*[m]y[m+n] \ , \tag{5.1}$$

with * being the complex conjugate operator.

In the case of real, finite, and discrete signals of size N $(0 \leq m \leq N-1)$ the right

[1] Parts of this chapter have been developed in cooperation with Boris Clénet and appear in his master thesis too. [1]

part of (5.1) becomes a sum between $m = 0$ and $m = N - 1$, hence

$$\Phi_{xy}[n] = \sum_{m=0}^{N-1} x[m]y[m+n] \ . \tag{5.2}$$

In this case, the CC does not exist for $n < -(N-1)$ and $n > N - 1$ and its size is $2N - 1$. From now, n will always refer to the index of the CC elements.

In order to have better understanding of what happens during the computation, Figure 5.1 depicts equation (5.2) in the case $N = 32$. Each dot of the grid represents a product between one element of $x[n]$ and one element of $y[n]$. The sum of the dots over a diagonal is one element of $\Phi_{xy}[n]$ as it is given by (5.2). For example, the longest diagonal in Figure 5.1 refers to equation (5.3), the result of the CC for $n = 0$.

$$\Phi_{xy}[0] = \sum_{m=0}^{N-1} x[m]y[m] \tag{5.3}$$

For this thesis the sum (5.2) has been subdivided in order to deal with the specific hardware architecture of the ISEF. In the following sections, the terms *square decomposition*, *linewise decomposition* and *diagonal decomposition* refer to the algorithms resulting from this subdivision.

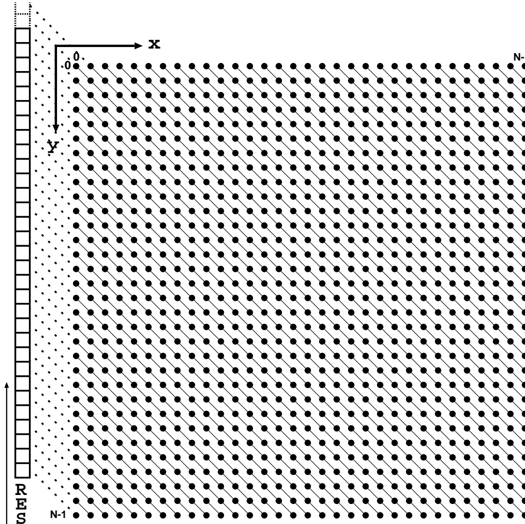


Figure 5.1: The cross correlation algorithm: Multiply-accumulate scheme for input frames $x[n]$ and $y[n]$. $N = 32$; Length of result vector: $2N - 1 = 63$;

5.2 CC decompositions

Three different EIs have been implemented which are using different types of CC decomposition. During the development process of the different approaches it became obvious that not only the hardware resources of the ISEF are the limiting factor, but the data flow from the ISEF is essential. If data has to be stored in the result vector of the CC after each EI, the execution time increases.

The first EI which has been developed is the square decomposition. The multiplications depicted in Figure 5.1 are combined to small squares. After the execution of the EI the results are added to the CC result vector. There are a lot of squares because of the small size of the squares. Therefore, a considerable amount of data transfer from the ISEF is necessary.

The second EI which has been developed is the linewise decomposition. Multiplications in Figure 5.1 are combined to larger squares which are computed in a linewise manner over several EIs. The results computed by the EI are stored in the CC vector, partly during the computation of each line of the square, and partly after the linewise computation of the square. Data transfer from the ISEF is reduced in comparison to the square decomposition, the execution time decreases.

The diagonal decomposition does not compute squares of multiplications but rhomboids. Due to the rhomboid structure of the decomposition, results have to be stored in the CC result vector only at the end of computing a diagonal of multiplications. The execution time decreases further.

5.2.1 Square decomposition

The subdivision of the CC vector into small blocks (squares) is the principle of this algorithm. Figure 5.2 ($N = 32$) describes this algorithm with a block size $b = 4$. b samples per frame are needed in order to compute all of the multiplications included in one square.

Furthermore, one square also represents the CC between b samples from $x[n]$ and b samples from $y[n]$. This function is called $c_{i,j}$. In order to navigate between the squares, two further variables are introduced: $\alpha = \left\lfloor \frac{|n|}{b} \right\rfloor$ and $\beta = |n| \bmod b$, respectively the quotient and the remainder of the division of $|n|$ by b. α can be interpreted as a block

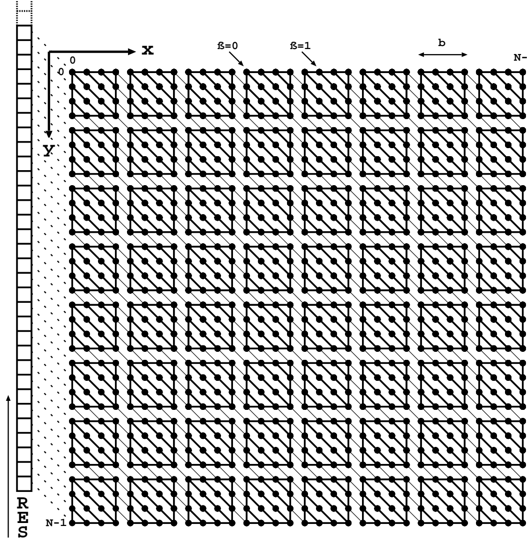


Figure 5.2: Principle of square decomposition: Multiply-accumulate scheme for input frames $x[n]$ and $y[n]$. $N = 32$; Length of result vector: $2N-1 = 63$;

index and β as an offset in this block.

$$c_{i,j}[n] = \sum_{m=0}^{b-1} x_i[m] y_j[m+n] = \Phi_{x_i y_j}[n] \tag{5.4}$$

$$x_i[m] = \begin{cases} x[m] & \text{if } i \leq m < i+b \\ 0 & \text{otherwise} \end{cases}$$

$$y_j[m] = \begin{cases} y[m] & \text{if } j \leq m < j+b \\ 0 & \text{otherwise} \end{cases}$$

A diagonal (an element of Φ_{xy}) goes through a certain amount of squares depending on n. When $\beta = 0$, n is a multiple of b and the diagonal is the assembly of the main diagonals of the squares it crosses. Only one square per square-line is crossed. The expression of the CC for the square decomposition and $\beta = 0$ can be written as:

$$\Phi_{xy}[n] = \begin{cases} \sum_{i=\alpha}^{\frac{N}{b}-1} c_{i,i-\alpha}[\beta] & \forall n < 0, \ \beta = 0 \\ \sum_{j=\alpha}^{\frac{N}{b}-1} c_{j-\alpha,j}[\beta] & \forall n \geq 0, \ \beta = 0 \ . \end{cases} \tag{5.5}$$

Otherwise, for $\beta \neq 0$ the diagonal is a combination of small diagonals with two possible lengths, that are α and $b-\alpha$. Two squares per square-line are crossed. The $\beta \neq 0$ case adds terms to the sums of (5.5) in order to take all of the extra squares that are crossed into account. The expression of the CC for the square decomposition and $\beta \neq 0$ is:

$$\Phi_{xy}[n] = \begin{cases} \sum_{i=\alpha}^{\frac{N}{b}-1} c_{i,i-\alpha}[\beta] + c_{i+1,i-\alpha}[-(b-\beta)] & \forall n < 0,\ \beta \neq 0 \\ \sum_{j=\alpha}^{\frac{N}{b}-1} c_{j-\alpha,j}[-\beta] + c_{j-\alpha,j+1}[b-\beta] & \forall n \geq 0,\ \beta \neq 0\ . \end{cases} \tag{5.6}$$

In order to show the diagonals and the squares that are crossed for different values of β, two diagonals are pointed in Figure 5.2. They represent the cases $\beta = 0$ and $\beta = 1$ ($\beta \neq 0$).

In (5.5) and (5.6) it can be noticed that i for $n < 0$ and j for $n \geq 0$ have the same role and inversely. This comes from property (5.7) of the CC (cf. [13]).

$$\Phi_{xy}[n] = \Phi^{*}_{yx}[-n] \tag{5.7}$$

Figure 5.3 shows the principle of the merging algorithm in the case $N = 32$ and $b = 4$.

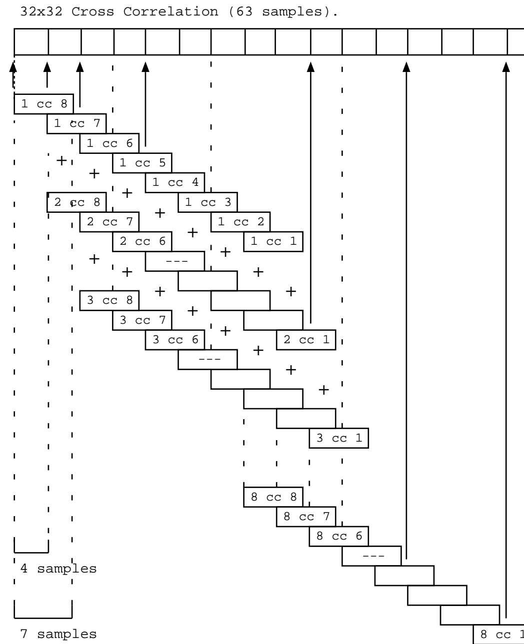


Figure 5.3: Principle of the merging algorithm for square decomposition.

Each "*i cc j*" rectangle represents the result of the CC between the ith b-samples group of the first frame and the jth b-samples group of the second frame. In other words, it depicts the non-zero results of $c_{i,j}$.

5.2.2 Linewise decomposition

The linewise decomposition differs slightly from the square decomposition. The same number of multiplications are computed, but the arrangement of the block is linear. Varying the computation flow of the algorithm results in a different data flow which is more appropriate for the hardware architecture.

A segment of length b represents multiplications between one element of $x[n]$ and b elements of $y[n]$. Figure 5.4 depicts this algorithm for a frame of $N = 32$ and $b = 16$.

After computing one line of the CC, the input sample $x[n]$ has to be updated and one final element of the actual subblock from the CC frame is available (due to the diagonal nature of the result). Repeating this task b times completes the calculation of one $b \times b$ subblock. There are four subblocks present in the actual example (5.4) which are identified by horizontal dashed lines.
In other words, a $b \times b$ CC subblock is derivable from b linear blocks.

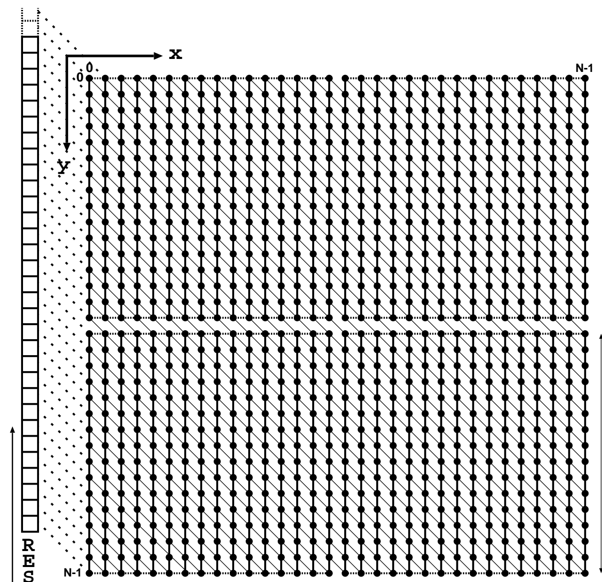


Figure 5.4: Principle of linewise decomposition: Multiply-accumulate scheme for input frames $x[n]$ and $y[n]$. $N = 32$; Length of result vector: $2N-1 = 63$;

Similar to the square decomposition case, let us call $l_{i,j}$ the function that gives the expression of such linear blocks.

$$l_{i,j}[m] = \begin{cases} x[i]y[j+m] & \text{if } 0 \leq m < b \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

Equation (5.9) gives the expression for the CC of a $b \times b$ subblock as a function of (5.8). The function $c_{i,j}$ is used as it was defined in (5.4).

$$c_{i,j}[n] = \sum_{k=0}^{b-1} l_{i+k,j}[k-n] \tag{5.9}$$

Finally, replacing $c_{i,j}$ from (5.9) in (5.5) and (5.6) provides the final expression of the whole CC frame of size $2N-1$, as was done for the square decomposition case.

In the case of linear decomposition, the same multiplications are performed as in (5.2), only their order of computation changes. Note also that for linewise decomposition, b is generally greater than the b of other decomposition algorithms.

5.2.3 Diagonal decomposition

The diagonal decomposition is similar to the square decomposition. In this case the squares are angled so that the merging of a block requires less access to the final result. Figure 5.5 depicts this algorithm for a frame of $N = 32$ and $b = 4$.

For the sake of better understanding, a replication of input frame $x[n]$ is depicted at the right half of Figure 5.5. In a mathematical sense, the replication can be expressed as "n mod N" operator for input vector $x[n]$.

Merging the blocks into the final result is simpler here, because each diagonal of a block belongs to only one diagonal of the final grid. Imagine a block-size of $b \times b$, where

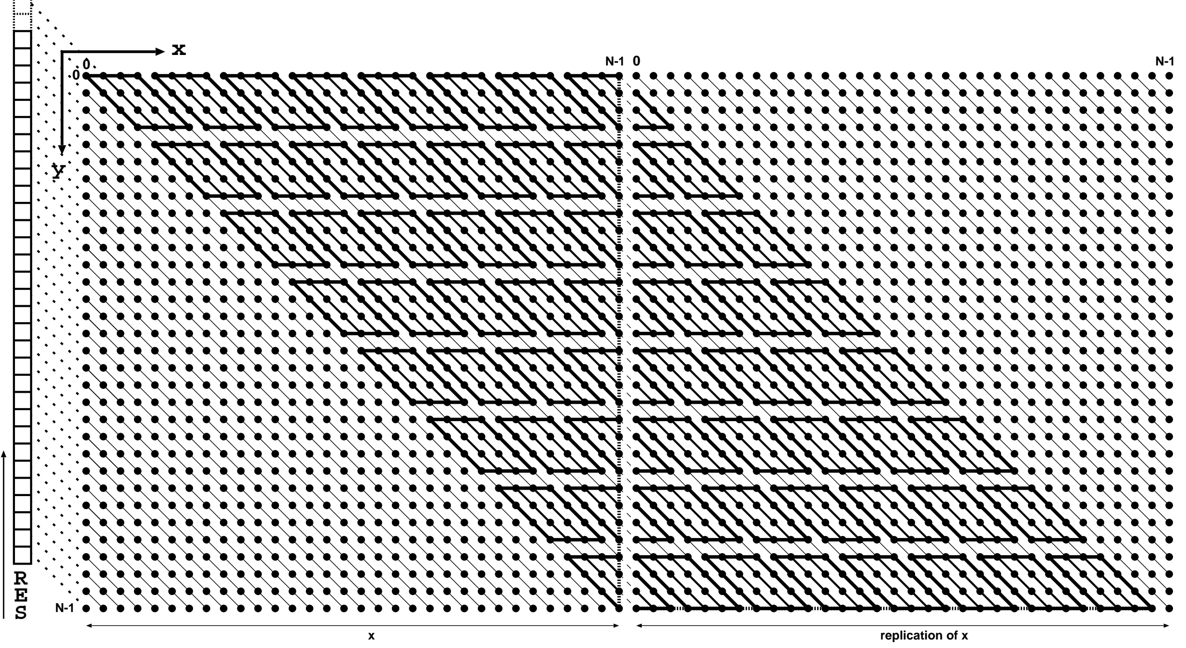


Figure 5.5: Principle of diagonal decomposition: Multiply-accumulate scheme for input frames $x[n]$ and $y[n]$. $N = 32$; Length of result vector: $2N-1 = 63$;

N is a multiple of b; (5.10) and (5.11) provide the diagonal decomposition of the CC.

$$\Phi_{xy}[n] = \sum_{m=0}^{\frac{N}{b}-1} \sum_{k=0}^{b-1} x[mb+k+n]y[mb+k] \ ,$$

$$\text{if} \quad \begin{cases} 0 \leq n \leq (N-1) \ , \ \text{and} \\ (mb+k+n) < N \ , \ \text{and} \\ \big(x[N], \ldots, x[2N-1]\big) = \big(x[0], \ldots, x[N-1]\big) \ . \end{cases} \tag{5.10}$$

$$\Phi_{xy}[n-N] = \sum_{m=0}^{\frac{N}{b}-1} \sum_{k=0}^{b-1} x[mb+k+n]y[mb+k] \ ,$$

$$\text{if} \quad \begin{cases} 0 < n \leq (N-1) \ , \ \text{and} \\ (mb+k+n) \geq N \ , \ \text{and} \\ \big(x[N], \ldots, x[2N-1]\big) = \big(x[0], \ldots, x[N-1]\big) \ . \end{cases} \tag{5.11}$$

It can be observed that the border of input frame $x[n]$ is crossed for all but the

first diagonals. The result vector of equation (5.10) is pictured at this vertical border. Together with the second result part of equation (5.11), marked with dashed lines in Figure 5.5 ("bottom right"), the CC of equation (5.2) is completely described.

5.3 Data flow of the CC implementations

The first four sections of chapter 5 gave the mathematical background for computing the CC on Stretch hardware. This section provides a detailed description of the data flow for each approach.

5.3.1 Straightforward implementation without the ISEF

This implementation provides a reference for the following versions of the algorithm. It uses the Xtensa with normal C code without the use of the extra capabilities provided by the S6 architecture. It is the slowest version, but it is also the easiest to verify. For this reason, the speed improvement quantifications and the results can always be compared to it.

The data to work on is placed in the main memory. No use of DMA or DATARAM is explicitly requested. The actual signal processing part takes place in the ALU. It computes the CC as described in (5.2), and Figure 5.1.

5.3.2 Using the ISEF and WRs

The best way to improve the performances of the straightforward approach is to move some computations to the ISEF. One way of providing the ISEF with data is to do it over the WRs.

The first step is to transfer the data from the main memory of the board to the DATARAM. This operation is achieved either with or without DMA. Both approaches have been used and compared. Figure 5.6 depicts the flow of data for all of the solutions that use the ISEF with the WRs. Depending on the implemented approach, the dashed arrows may represent data transfer with or without DMA.

Once the data is in the DATARAM, there are several ways to proceed. As announced in sections 5.1 – 5.2.3, some decompositions of the CC have been developed so that the ISEF can compute it. For each EI the correct set of samples has to be provided to the ISEF via the WRs. The choice of this set constitutes the main difference between

the algorithms. In addition, the way in which the EI's outputs are merged into the CC result vector differs as well.

Using ISEF with the square decomposition

In the square decomposition case the ISEF runs a $b \times b$ CC in one EI. This means that each EI receives $b \times b$ inputs via the WRs, and outputs $2b - 1$ samples via the WRs. The operation is repeated $(N/b)^2$ times so that each block is computed. The merging of a block is done right after its calculation, after it is issued from the WRs.

The handling of the current positions in the input frames (i.e. the indexes i and j of a "i cc j" rectangle in Figure 5.3) is done by pointers at the outside of the ISEF. The same principle is used in order to point the right place in the result frame (i.e. where the current block has to be added). The EI source code of the square decomposition is discussed in appendix C.1.

Using ISEF with the linewise decomposition

Concerning data flow in memory, the approach here is different. The first step is to transmit a whole input block from the first frame into the ERs of the ISEF over the WRs. Then, samples from the second frame are transferred one by one into the ISEF. Each EI receives one sample. After b iterations, a $b \times b$ CC is performed. Note that in this case b is greater than in the previous decompositions.

At the same time, after each EI one sample of the $b \times b$ CC is ready. Therefore its storage is handled just after the EI. That is $b - 1$ stored output samples after $b - 1$ iterations. b samples remain uncomputed in order to reach the $2b - 1$ output frame size of a $b \times b$ CC. These samples are stored back in the result vector over the WRs after the bth iteration. The process of storing back is done alternately over WRA and WRB due

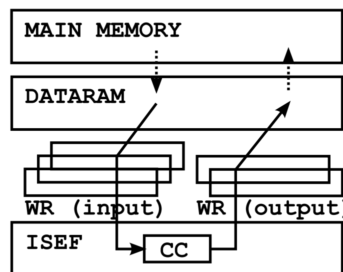


Figure 5.6: Data flow in the case of WRs use.

to performance reasons. The EI source code of the linewise decomposition is discussed in appendix C.2.

Using ISEF with the diagonal decomposition

For the diagonal decomposition (cf. Figure 5.5), the computation of a block requires $2b-1$ samples from the first frame $x[n]$ and b from the second frame $y[n]$. The first three samples of a block diagonal from vector $x[n]$ have to be initialized separately, whereas for further blocks the input samples along the x-dimension overlap and can be used from one EI to another by using the ER. The b samples from $y[n]$ are transferred through WRs and stay the same for all the blocks of a block-line. Each time one of these samples is loaded into the ISEF, a horizontal set of multiplications is done and the results are added to the right diagonal. Intermediate results from each diagonal of a block are stored in an ER table of size b. Once the computation of a block is finished, the value of the current position indicator is incremented.

Once the position indicator equals the input frame size, it means that a border frame has been reached. In this case $b-1$ samples from the beginning of $x[n]$ are loaded into the ISEF, and after computing the border block, b vertical samples are transferred from the ISEF through the WR to the final result of the CC. After performing computations on the border block, the diagonal application flow is carried on until the end of a diagonal has been reached, at which point the horizontal CC results are also stored via WR to the final result. N/b block diagonals have to be computed in order to reach the final result of the CC. The EI source code of the diagonal decomposition is discussed in appendix C.4.

5.3.3 Using the ISEF and IRAM

This solution makes use of the IRAM, a memory space which is placed within the ISEF and is also accessible from outside the ISEF. Each occurrence of an EI can process data from and to the IRAM, thus it is an alternative to the previous processing via the WRs. Furthermore, this memory space reaches 64 KB which enables the storage of a large amount of samples (e.g. entire frames).

Figure 5.7 depicts the flow of data for solutions using the IRAM. The data is transferred directly from the main memory to the IRAM via DMA. The dashed arrows represent these transfers. It is important to note that IRAM can not handle 8 bit data

types (cf. section 4.4.4), for this reason 16 bit tables are used instead.

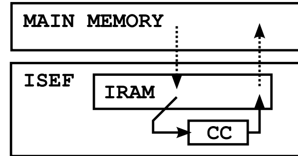


Figure 5.7: Data flow in the case of IRAM use.

Using IRAM with the linewise decomposition

Differences to the linewise decomposition using only WRs (cf. section 5.3.2) are small. Instead of loading the whole input block from the first input frame into the ER, the same block is loaded from the DATARAM to the IRAM using DMA.

Data transfer from the second input frame is performed via WRs. While calculating single lines of the result the intermediate results are again stored in the IRAM, whereas single result samples are transferred outwards by WRs. After b iterations of the algorithm, b in the IRAM remaining result samples are stored back in the final CC result vector using DMA. The EI source code of the diagonal decomposition with IRAM is discussed in appendix C.3.

5.4 Experiments and results

This section deals with the performance of the different CC approaches. Experiments have been performed for different combinations of decompositions and input data resolution. For example, the square decomposition has only been executed with an input resolution of 8 bit, because it became clear that this approach is not fast enough. The IRAM approach of the linewise decomposition has only been executed for an input resolution of 16 bit, because execution took too long. At the end of the chapter performance observations of loop unrolling, the DMA transfer, the Gammatone filterbank and the PoPi decomposition are presented.

The following references have been given in order to refer to the CC solutions easily:

i. Straightforward implementation without using the ISEF.

ii. Square decomposition with ISEF and WR.

iii. Linewise decomposition with ISEF and WR.

iv. Linewise decomposition with ISEF, WR and IRAM.

v. Diagonal decomposition with ISEF and WR.

In order to make different approaches easier to compare, some variables for the designing and simulation process have been fixed:

- EIs are, if possible, always compiled with bitfile generation.
- The issue rate is fixed with 1.
- The chip frequency was tuned until the ISEF was able to meet the target with the particular EI.
- Fixed point calculations are performed with high resolution, therefore no truncations at the CC results are necessary.
- EI are designed to use as many ISEF resources as possible.
- For cycle simulations only one microphone pair is used.
- No filterbank is arranged in front of the CC.
- No PoPi decomposition is performed after the CC.
- In order to completely load the EI to the ISEF, one dry run is necessary before starting the measurements.
- Sampling frequency = 48 kHz
- Frame shift = 20 ms
- Frame size = 100 ms

5.4.1 Comparison of 8 bit implementations

EI	f_{Xtensa}	Cycles/Frame	t_{Frame}
i	300 MHz	75.26 M	250.9 ms
ii	250 MHz	32.62 M	130.5 ms
iii	170 MHz	10.98 M	64.6 ms

Table 5.1: Performance of 8 bit Stretch simulations.

The straightforward approach i does not use the ISEF, hence no chip frequency limitations have to be made. Square decomposition ii has the disadvantage that the merging loop to add intermediate results to the final CC result has do be called very often, which slows down the algorithm. Linewise decomposition iii gets rid of some merging loops therefore performance is better.

5.4.2 Comparison of 16 bit implementations

EI	f_{Xtensa}	Cycles/Frame	t_{Frame}
i	300 MHz	207.62 M	692.1 ms
iii	190 MHz	43.91 M	231.1 ms
iv	300 MHz	591.55 M	1971.8 ms
v	230 MHz	29.13 M	126.7 ms

Table 5.2: Performance of 16 bit Stretch simulations.

Approaches with a data width of 16 bit are already significantly slower due to the increased use of ISEF resources. Nevertheless, accuracy considerations of section 3.2.5 show that it is recommendable to use an input data resolution of 16 bit. For this reason the results of table 5.2 are the most valuable.

The first approach i again states the straightforward implementation with no limitations to the chip frequency. iii computes the CC blocks linewise, although merging block results to the overall CC is still necessary.

In approach iv changes of the data flow have been made. One of the two input frames is stored directly into the IRAM via DMA. Data in the IRAM can be used directly within an EI. Unfortunately performance measurements using the IRAM are very poor, and due to an incomprehensible internal error in the Stretch compiler, bitfile generation was not possible.

Approach v uses diagonal decomposition, therefore no subsuming of CC block results is necessary any more. This implementation is the second fastest of all the approaches, with the added benefit of computing 16 bit data. It is the preferred approach.

5.4.3 Comparison of 24 bit implementations

EI	f_{Xtensa}	$^{\text{Cycles}}/_{\text{Frame}}$	t_{Frame}
i	300 MHz	874.71 M	2915.7 ms
iii	180 MHz	98.06 M	544.8 ms

Table 5.3: Performance of 24 bit Stretch simulations.

24 bit approaches conclude the performance observations with varying input resolutions. As there are no 24 bit data types available at the Xtensa, type casting is performed by an extra patch at the Xtensa. The patch slows down the execution time of this approach and makes it laborious.

Concerning data width the ISEF is very flexible, and consequently a linewise approach is available. Due to its hardware resource consuming behaviour, the performance of it is not good.

Summary

Figure 5.8 provides a time performance overview of simulations from tables 5.1, 5.2 and 5.3. Approach vi computes the time domain CC with a MEX subroutine in Matlab. It is added to allow for comparison between Matlab and Stretch implementations. According to section 2.8 the real-time requirement is given by the frame shift, which is 20 $^{\text{ms}}/_{\text{Frame}}$. It can be observed that no approach is able to execute computations in real-time. Promising approach v has a speed-up of factor 5.46 in comparison to the 16 bit straightforward approach.

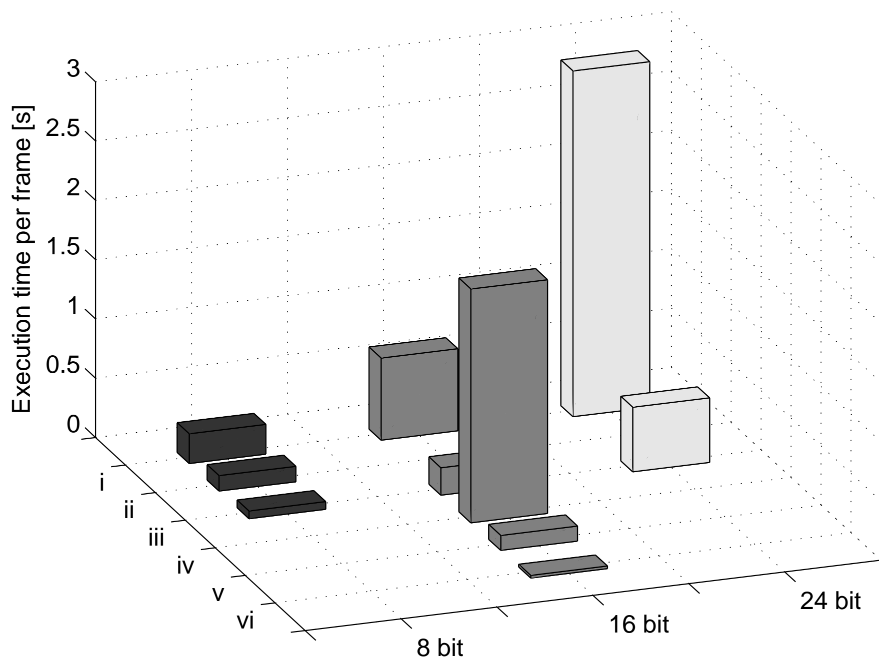


Figure 5.8: Summary of the Stretch CC simulations. Nomenclatures i – v are listed at the beginning of section 5.4. Approach vi computes the time domain CC with a MEX subroutine in Matlab.

5.4.4 Performance observations of further code parts

Loop unrolling

The speed of an EI is not only determined by its size or architecture. There are tremendous differences depending on the call of the EI. Table 5.4 shows measurements of a 32×32 CC using diagonal decomposition with different calling functions.

If the repeated call of an EI is performed via a loop, it is called a *not unrolled* EI call. If the loop of the EI call is dissolved by hand, it is called an *unrolled* approach. *Partly unrolled* approaches are for example getting rid of the inner loop of two nested loops. Investigations telling the Stretch compiler to unroll loops automatically (with compiler options explained in section 4.6) have not been successful, so *fully unrolled* means that the loops have to be dissolved by hand. Hence, programs are soon reach an unmanageable size (for both the programmer and the compiler).

EI call	f_{Xtensa}	Cycles/Frame	t_{Frame}
ALU	300 MHz	10454	34.8 μs
Not unrolled	230 MHz	11308	49.2 μs
Partly unrolled	230 MHz	1675	7.3 μs
Fully unrolled	230 MHz	1074	4.7 μs

Table 5.4: Comparison between different states of loop unrolling of the EI calling function (approach v; frame size = 32; resolution of the input data = 16 bit;).

DMA transfer

Simulation data for the EI is loaded from the main memory via DATARAM and WR. It is possible to load simulation data from the main memory into the DATARAM with the explicit use of a DMA transfer. Section 4.4.5 illustrates this procedure.

Data transfer	f_{Xtensa}	Cycles/Frame	t_{Frame}
With DMA	190 MHz	1.242 M	6.5 ms
Without DMA	190 MHz	1.070 M	5.6 ms

Table 5.5: Data transfer between the main memory and the DATARAM with/without an explicit use of the DMA (approach iii; resolution of the input data = 16 bit; sampling frequency = 8 kHz).

Contrary to prior assumptions, performance measurements of Table 5.5 show that the explicit use of DMA transfers is not efficient.

Gammatone filterbank and PoPi decomposition

Performance measurements of ALU based solutions of the Gammatone filterbank and the PoPi decomposition are depicted in Table 5.6.

Solutions for	f_{Xtensa}	$^{\text{Cycles}}/_{\text{Frame}}$	t_{Frame}
Gammatone filterbank (Double precision)	300 MHz	869.98 M	2899.9 ms
Gammatone filterbank (Single precision)	300 MHz	19.22 M	64.1 ms
PoPi decomposition	300 MHz	773.87 M	2579.6 ms

Table 5.6: Computational expense of the Gammatone filterbank and the PoPi decomposition (resolution of the input data = 16 bit).

Concerning the Gammatone filterbank: Stretch does not provide the possibility to calculate double precision floating point digits on the Floating Point Unit (FPU). Double precision calculations are emulated on the ALU and are not efficient.

6 Conclusion

This thesis investigates the development process of accelerating the PoPi source localization algorithm by parallel computation and by porting it to reconfigurable hardware.

Data recorded by a circular 24 channel microphone array has been processed to deliver the DOA for a speaker scenario. The algorithm consists of four parts: Gammatone filterbank, CC, PoPi decomposition, and evaluation of the PoPi matrix with tracking. The Gammatone filterbank, the CC and the PoPi decomposition have an independent, parallel structure for different pairs of microphones. Therefore, parallel computations at different arithmetic units are possible. Only the final evaluation and the tracking between interframe position estimates cannot be performed in parallel. An analysis of the algorithm is necessary in order to locate computational expensive parts. Computational expensive parts of the algorithm have been rewritten in order to increase efficiency. The cross correlation has been detected as the most computational expensive code segment. There are two possibilities for computing the CC: Either in time domain or in frequency domain. The time domain approach has the benefit of low mathematical complexity, therefore value range estimations are precise and algorithmic restructuring is flexible. The frequency domain approach is dependent on an efficient implementation of the Fourier Transform. However, computational complexity grows linearly with the increasing length of the input vectors. This is in contrast to the time domain approach, where a quadratical interrelation between complexity and input vector length is given. In order to find reasonable adjustments for the algorithm, accuracy vs. time performance measurements were carried out. This was done by changing one parameter of the algorithm after another. After the analysis and optimization, where the trade-off between accuracy and time performance has been explored, the source localization algorithm has been ported to a Stretch S6 hybrid reconfigurable CPU. This hardware is a hybrid of a classical arithmetic unit and a ISEF. It is the choice of the software developer whether to use the ALU, or the ISEF with its possibility of parallelizing the calculations of computational expensive code parts. For excessive use of the ISEF several approaches have

been implemented for the most complex part of the algorithm – the CC. Experiments evaluating the time performance of the different ISEF based approaches conclude this thesis.

The main goal of this thesis has been to accelerate the execution of the algorithm. It has been possible to reduce the execution time of the algorithm by optimizing the Matlab based approaches. A further reduction of the execution time has been possible by reducing the complexity of the algorithmic parameters. The analysis of the algorithmic parameters documented that the reduction of execution time goes hand in hand with a gain in accuracy. An acceleration by the factor 26 could be achieved with a 5 % gain in accuracy. The computational most expensive operation of the algorithm has been ported to a hybrid reconfigurable CPU, where an acceleration by the factor 5 has been possible. Nevertheless, it has not been possible to run the algorithm in real-time. Reasons for this are either the – after optimization still high – computational costs of the source localization algorithm, or the lack of hardware resources at the ISEF. Another aim has been, to run the source localization algorithm with the use of the microphone inputs of the Stretch board. Unfortunately it has not been possible to run applications at the hardware, only cycle accurate simulations have been possible.

The conclusion of this thesis is also an outlook for further development at the same time. At the current state of the project further investigations concerning the interframe speaker tracking are needed. The current implementation is only capable of detecting one speaker sufficiently exactly. An elaborate probability-based approach should be able to be more sophisticate. Solving hardware access problems of the Stretch environment would make it possible to test the algorithm on the hardware in order to gain further information on data flow and interaction between I/O parts and algorithmic parts of the application.

A RIFF-WAVE file format

Audio inputs have been simulated by audio “Resource Interchange File Format”-“Waveform Audio File Format” (RIFF-WAVE) files [22] while testing implemented algorithms. Their samples were coded with a resolution of 8 bit, 16 bit, or 24 bit. Possible sampling frequencies are 8 kHz, 16 kHz, 32 kHz, or 48 kHz.

From byte 0x00 a RIFF-WAVE file begins with the header which describes the file contents on a size of 44 Byte and is composed of three chunks. The elements of each are listed below in the order in which they appear in the file.

- RIFF chunk (declaration of the RIFF-WAVE format)

Name	Bytes	Description
FileTypeChunkID	4	“RIFF” (0x52,0x49,0x46,0x46)
FileSize	4	size of the file in byte (minus 8 Byte)
FileFormatID	4	“WAVE” (0x57,0x41,0x56,0x45)

- Audio format chunk

FormatChunkID	4	“fmt” (0x66,0x6D, 0x74,0x20)
BlocSize	4	chunk size in byte minus 8 Byte (0x10)
AudioFormat	2	storage format (1 for PCM, . . .)
Channels	2	nb. of channels (0 for Mono, 1 for Stereo)
Frequency	4	sampling frequency [Hz]
BytePerSec	4	nb. of byte per second
BytePerSample	2	nb. of byte per sample (among all channels)
BitsPerSample	2	nb. of bit per channel’s sample (8, 16, or 24)

- Data chunk

DataBlocID	4	“data” (0x64,0x61,0x74,0x61)
DataSize	4	nb. of byte of data

After the header, the data is stored in *little endian* sample after sample (i.e. the kth sample of channel n is written before the $k+1^{th}$ of channel n-1).

In order to handle RIFF-WAVE files and to retrieve the data stored in the header, a structure has been created.

```
1  struct wavefile
2  {
3          // RIFF - CHUNK
4          char riff_name[4];
5          long riff_length;
6          char riff_type[4];
7          // FMT - CHUNK
8          char fmt_name[4];
9          long fmt_length;
10         short formattyp;
11         short canalnb;
12         long samplerate;
13         long b_per_sec;
14         short b_per_sample;
15         short Bits_per_sample;
16         // DATA - CHUNK
17         char data_name[4];
18         long data_length;
19 }
```

This information is necessary in order to access the desired parts of the data in the right way. The following extract from the source code provides an example of how the structure can be used. It stores a frame of 100 ms in the data pool ddr_pool at the address given by p_samples_1.

```
1  struct wavefile wf1;
2  FILE *p_file1;
3  void *p_samples_1; //pointer to the frame
4
5  p_file1 = fopen("file.wav", "rb");
6  fread((void *) &wf1, sizeof(wf1), 1, pwfile1); //reads the header
7
8  //number of bytes in the frame (corresponds to 100ms)
9  frame_length_in_bytes = (int)(0.1 * wf1.samplerate * wf1.bytes_per_sample);
10
11 //allocates memory
12 p_samples_1 = sx_mm_zalloc(ddr_pool, frame_length_in_bytes);
13 fread(p_samples_1, 1, vefile1.data_length, pwfile1); //store the frame
```

It is different if samples from the RIFF-WAVE file are stored with a width of 8 bit. In this case, retrieved samples are unsigned integers with values in the range [0; 255]. In order to get the real signed values, it is necessary to subtract 128 from each sample to achieve the value range of $[-128; 127]$.

B Figures

B.1 Additional Figures to chapter 4

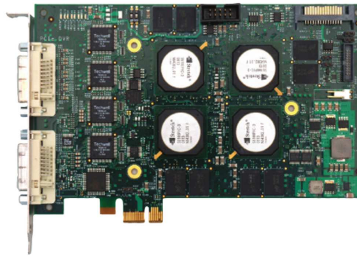


Figure B.1: Picture of the VRC6016 card

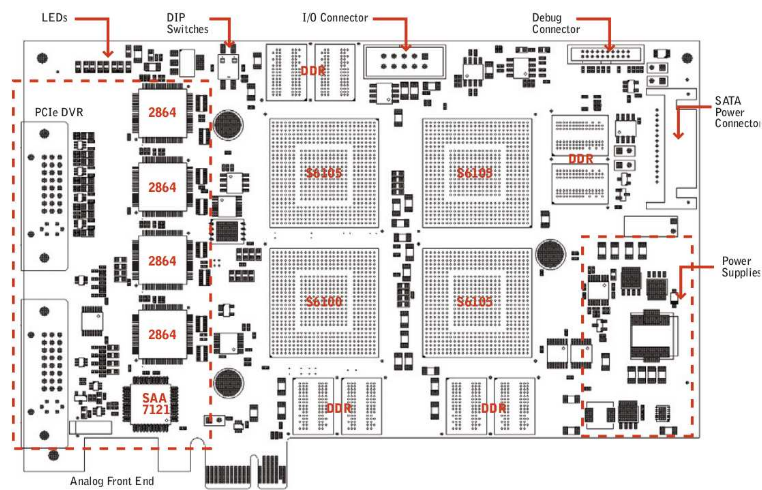


Figure B.2: VRC6016 board layout

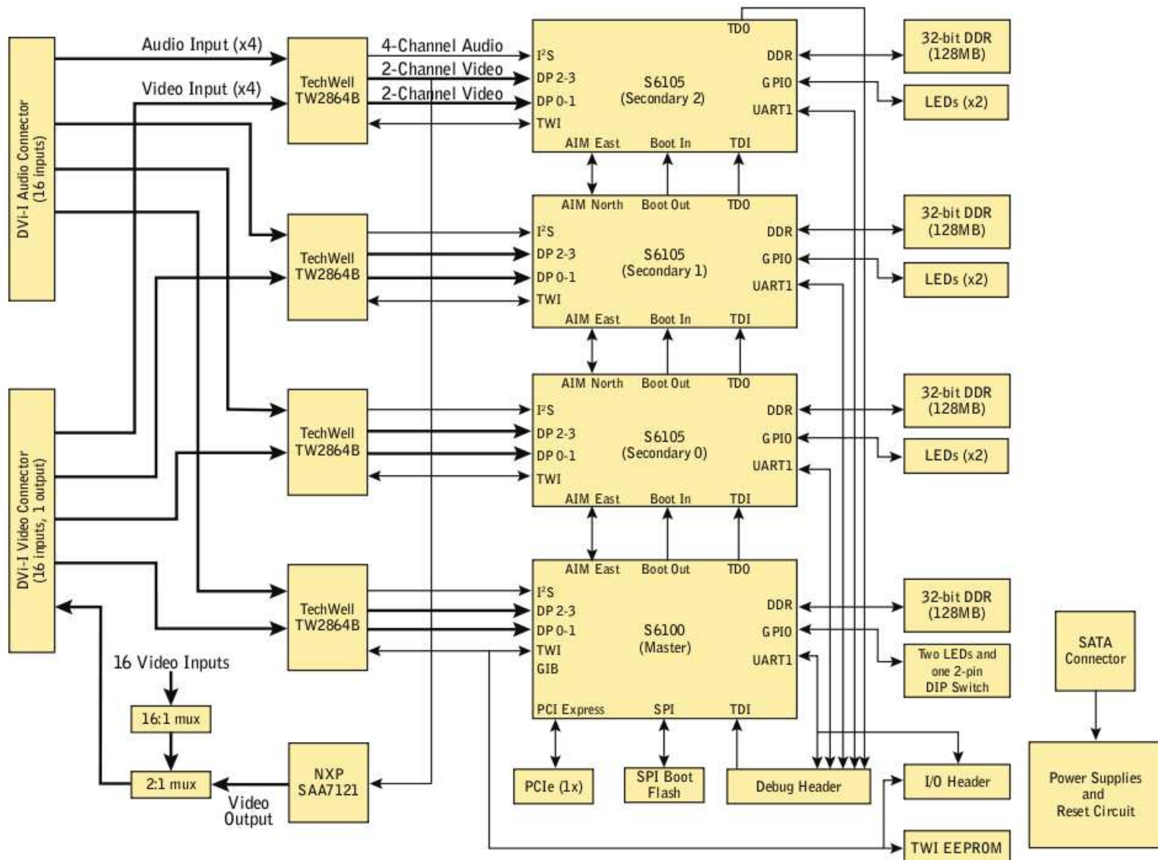


Figure B.3: VRC6016 board block diagram

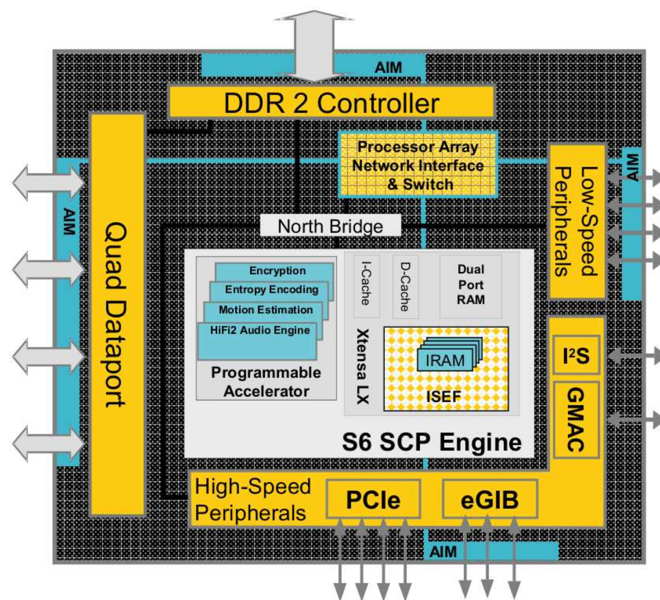


Figure B.4: Architecture of the Xtensa processor

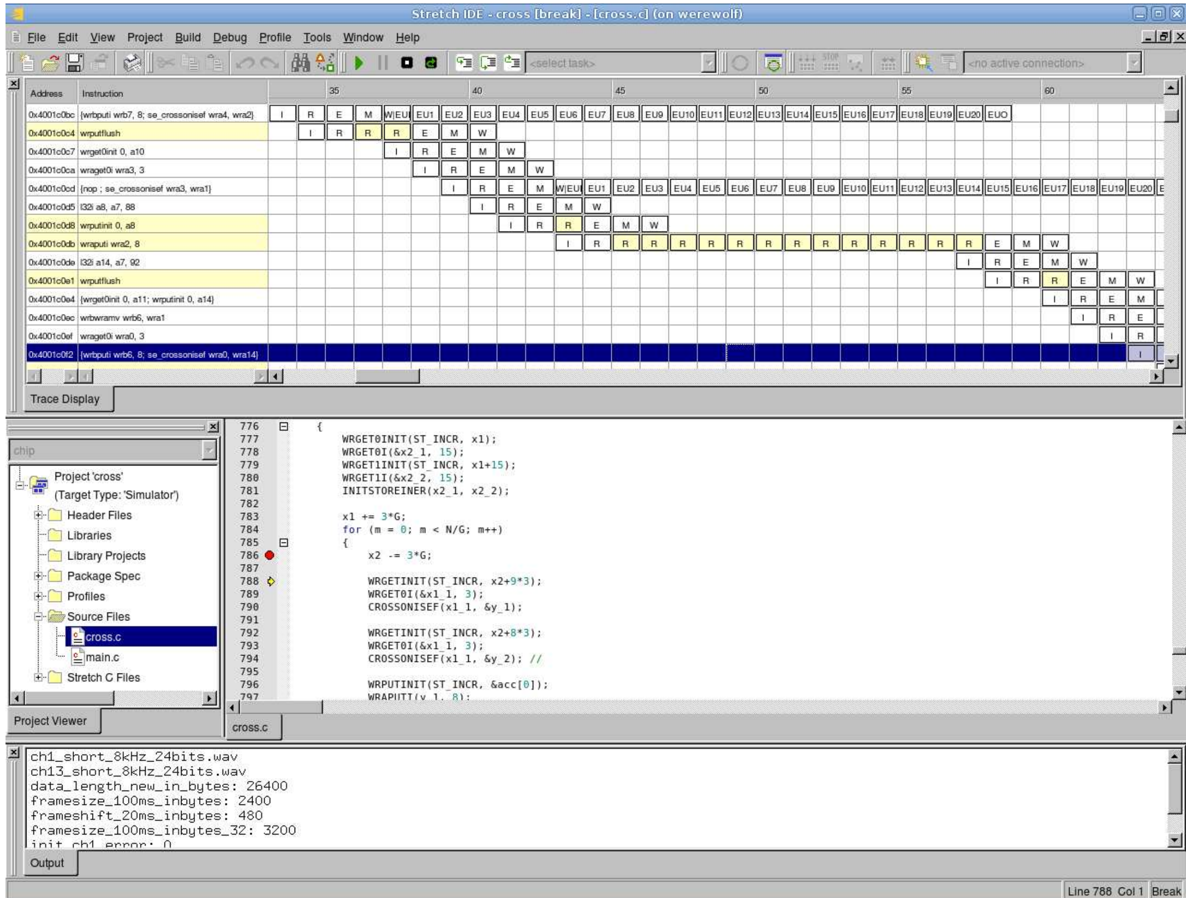
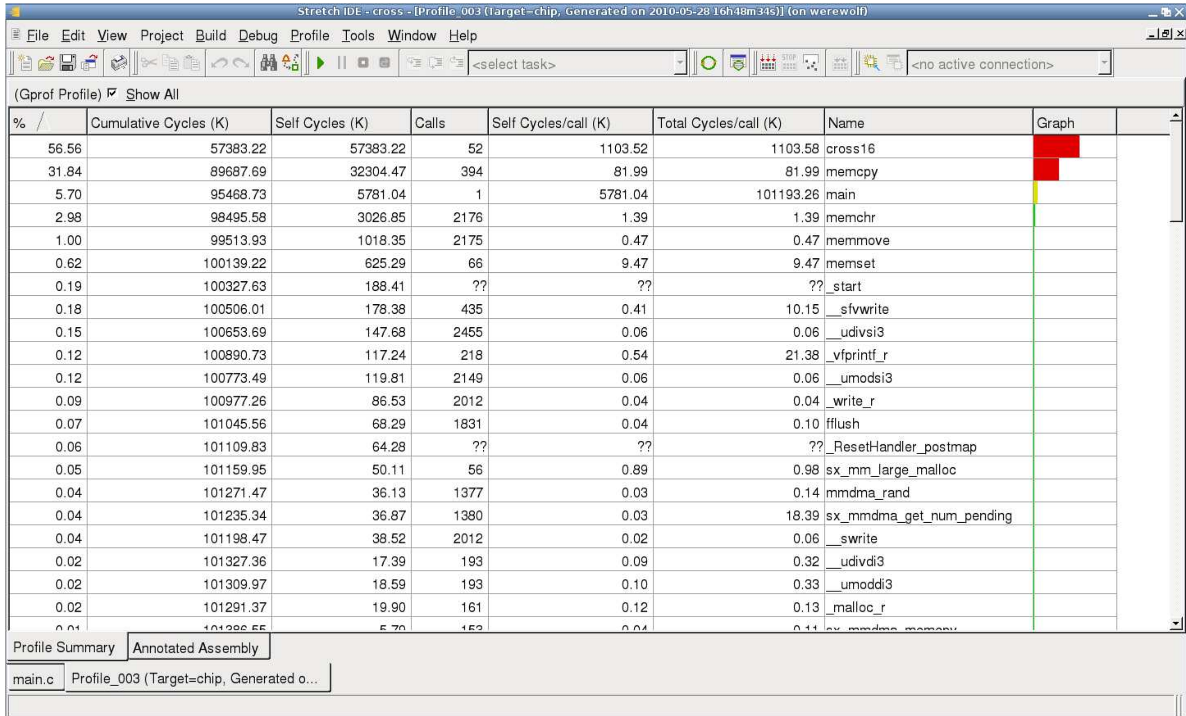


Figure B.5: Pipeline view of the Stretch IDE



%	Cumulative Cycles (K)	Self Cycles (K)	Calls	Self Cycles/call (K)	Total Cycles/call (K)	Name	Graph
56.56	57383.22	57383.22	52	1103.52	1103.58	cross16	
31.84	89687.69	32304.47	394	81.99	81.99	memcpy	
5.70	95468.73	5781.04	1	5781.04	101193.26	main	
2.98	98495.58	3026.85	2176	1.39	1.39	memchr	
1.00	99513.93	1018.35	2175	0.47	0.47	memmove	
0.62	100139.22	625.29	66	9.47	9.47	memset	
0.19	100327.63	188.41	??	??	??	_start	
0.18	100506.01	178.38	435	0.41	10.15	__sfvwrite	
0.15	100653.69	147.68	2455	0.06	0.06	__udivsi3	
0.12	100890.73	117.24	218	0.54	21.38	_vfprintf_r	
0.12	100773.49	119.81	2149	0.06	0.06	__umodsi3	
0.09	100977.26	86.53	2012	0.04	0.04	_write_r	
0.07	101045.56	68.29	1831	0.04	0.10	fflush	
0.06	101109.83	64.28	??	??	??	_ResetHandler_postmap	
0.05	101159.95	50.11	56	0.89	0.98	sx_mm_large_malloc	
0.04	101271.47	36.13	1377	0.03	0.14	mmdma_rand	
0.04	101235.34	36.87	1380	0.03	18.39	sx_mmdma_get_num_pending	
0.04	101198.47	38.52	2012	0.02	0.06	__swrite	
0.02	101327.36	17.39	193	0.09	0.32	__udivdi3	
0.02	101309.97	18.59	193	0.10	0.33	__umoddi3	
0.02	101291.37	19.90	161	0.12	0.13	_malloc_r	

Figure B.6: The profiling functionality

C Extension Instructions of the CC

C.1 Square Decomposition

```
1  static se_sint<32> res[15];
2
3  SE_FUNC void CROSSCORR(WRA wx1, WRB wx2, WRA *Y_1, WRB *Y_2)
4  {...}
5
6  SE_FUNC void GETFROMER(WRA *Y_1, WRB *Y_2)
7  {...}
8
9  /**********************************************************************\
10 *                    Final resource usage report                       *
11 *----------------------------------------------------------------------*
12 * Configuration crosscorr:
13 * Total AUs                 =  1728 out of 4096
14 * Total MUs                 =  8192 out of 8192
15 * Total SHIFTs              =     0
16 * Total IRAMs               =     0 out of   32
17 * Total PRIENC bits         =     0 out of  256
18 * Target issue rate         =     1
19 * Target chip frequency     = 250.0 Mhz
20 * Target ISEF frequency     = 250.0 Mhz
21 * Achieved ISEF frequency   = 250.3 Mhz
22 * Maximum output write cycle =    18
23 * ISEF can run at the required frequency.
24 * Compile time              =  1821 seconds
25 \**********************************************************************/
```

Two EIs are defined for the 8 bit implementation of the square decomposition. In the EI CROSSCORR (defined in line 3) a square of the CC is computed in parallel (cf. section 5.2.1). With the EI GETFROMER (defined in line 6) the second part of the CC results is transferred from the ISEF to the final CC result vector. Input values are transferred to the ISEF over WRA (wx1) and WRB (wx2) defined in line 3. Intermediate results from the computation of the CC are stored in the ER res defined in line 1. The results and the intermediate results have a higher bit width (32 bit) than the input values (8 bit). Therefore, the data transfer from the ISEF to the CC result vector has to be split and is

performed in two EIs (line 2 and line 6) over WRA (*Y_1) and WRB (*Y_2). In lines 9 to 25 the final resource usage report of the ISEF for the square decomposition is presented.

C.2 Linewise Decomposition

```
1  static se_sint<24> A[10];
2  static se_sint<64> B[10];
3
4  SE_FUNC void INITSTOREINER(WRA X2_1, WRB X2_2)
5  {...}
6
7  SE_FUNC void CROSSONISEF(WR X1, WR *Y)
8  {...}
9
10 SE_FUNC void GETFROMER1(WRA *Y_1, WRB *Y_2)
11 {...}
12
13 SE_FUNC void GETFROMER2(WRA *Y_1, WRB *Y_2)
14 {...}
15
16 SE_FUNC void GETFROMER3(WRA *Y_1)
17 {...}
18
19 /*************************************************************************\
20 *                     Final resource usage report                         *
21 *-------------------------------------------------------------------------*
22 * Configuration cross24:
23 * Total AUs                  =  2144 out of 4096
24 * Total MUs                  =  6400 out of 8192
25 * Total SHIFTs               =     0
26 * Total IRAMs                =     0 out of   32
27 * Total PRIENC bits          =     0 out of  256
28 * Target issue rate          =     1
29 * Target chip frequency      = 180.0 Mhz
30 * Target ISEF frequency      = 180.0 Mhz
31 * Achieved ISEF frequency    = 180.1 Mhz
32 * Maximum output write cycle =    12
33 * ISEF can run at the required frequency.
34 * Compile time               =  2222 seconds
35 \*************************************************************************/
```

Five EIs are defined for the 24 bit implementation of the linewise decomposition. In the EI INITSTOREINER (defined in line 4) the first input vector is transferred to the ER A (defined in line 1) via WRA (X2_1) and WRB (X2_2). In the EI CROSSONISEF (defined in line 7) a line of the square of the CC is computed in parallel (cf. section 5.2.2). One input value of input vector two is transferred to the ISEF over the WR (X1), and one result value is transferred from the ISEF to the CC result vector via the WR (*Y). After the computation of the square is finished, the second part of CC results is stored in the

ER B (defined in line 2) and has to be transferred to the CC result vector by the EIs GETFROMER1, GETFROMER2, and GETFROMER3 (defined in lines 10, 13, and 16) over the WRs (*Y_1 and *Y_2). In lines 19 to 35 the final resource usage report of the ISEF for the linewise decomposition is depicted.

C.3 Linewise decomposition with IRAM

```
1  //   Computational Resources
2  //     Arithmetic bits.................640
3  //     Logic bits......................336
4  //     Mux bits.......................1160
5  //     Register bits.....................0
6  //     Pipeline bits..................1092
7  //   AU bits total...................3228 out of 4096
8  //   MU bits total...................2560 out of 8192
9  //   SHIFT bits total.....................0 out of 4096
10 //   IRAM total.........................32 out of 32
11 //   PRIENC bits total..................0 out of 256
12 //   Extension registers................0 out of 4096
13
14 static se_sint<16> A[2][8];
15 SE_MEM(A);
16 static se_sint<64> B[4][2];
17 SE_MEM(B);
18 static se_sint<64> C[2][2];
19 SE_MEM(C);
20 static se_sint<64> D[8][2];
21 SE_MEM(D);
22
23 SE_FUNC void CROSSONISEF(WR X1, WRA *Y)
24 {...}
25
26 SE_FUNC void GETFROMIRAM()
27 {...}
28
29 /***************************************************************************\
30 *                       Final resource usage report                          *
31 *---------------------------------------------------------------------------*
32 * Configuration crossisef:
33 * Resource usage is not available - please complete compilation to bitstream.
34 There is no logic in ISEF. no frequency data is available
35 * Compile time              =      4 seconds
36 \***************************************************************************/
```

The 16 bit implementation of the linewise IRAM approach uses a different data flow than the linewise WR approach described in section C.2. The first input vector and intermediate results from the computation are not stored in the ER but in the IRAM (defined in lines 14 to 21). For efficient use of the IRAM variables A, B, C, and D are

distributed over all 32 banks of the IRAM (cf. section 4.4.3). The single load and store instructions of the results which are computed during the linewise computation of the squares (cf. section 5.2.2) are transferred via the WRs `X1` and `*Y` (line 23). After the linewise computation of the square is finished, the second part of CC results is sorted and stored in the IRAM (EI `GETFROMIRAM` line 26) and has to be transferred to the CC result vector by DMA. The compilation with bitfile generation failed due to an internal error of the Stretch compiler, therefore no final resource usage report is available (lines 29 to 36). An estimated resource usage report is available in lines 1 to 12.

C.4 Diagonal Decomposition

```
1  static se_sint<64> sumver[4];
2  static se_sint<64> sumhor[4];
3  static se_sint<16> olda[3];
4  static se_uint<16> run;
5
6  SE_FUNC void CROSSONISEF(SE_INST CC_MAC, SE_INST CC_INIT_MAC, SE_INST CC_FIN_MAC,
7                           WRA A, WRB B, WRA *Y_1, WRB *Y_2)
8  {...}
9
10 SE_FUNC void GETFROMER(WRA *Y_1, WRB *Y_2)
11 {...}
12
13 /***************************************************************************\
14 *                     Final resource usage report                           *
15 *---------------------------------------------------------------------------*
16 * Configuration cross200:
17 * Total AUs                 =  2304 out of 4096
18 * Total MUs                 =  4096 out of 8192
19 * Total SHIFTs              =     0
20 * Total IRAMs               =     0 out of   32
21 * Total PRIENC bits         =     0 out of  256
22 * Target issue rate         =     1
23 * Target chip frequency     = 230.0 Mhz
24 * Target ISEF frequency     = 230.0 Mhz
25 * Achieved ISEF frequency   = 230.7 Mhz
26 * Maximum output write cycle =    22
27 * ISEF can run at the required frequency.
28 * Compile time              =  1554 seconds
29 \***************************************************************************/
```

Four EIs are defined for the 16 bit implementation of the diagonal decomposition. In the EIs `CC_MAC`, `CC_INIT_MAC`, and `CC_FIN_MAC` (defined in line 6) a rhomboid of the CC is computed in parallel (cf. section 5.2.3). Input values are transferred to the ISEF over WRA (`A`) and WRB (`B`) defined in line 7. Input values which have to be stored between the execution of two rhomboids are stored in the ER `olda`. Intermediate results from the

computation of the CC are stored in the ER `sumhor` defined in line 2. With the EI `GETFROMER` (defined in line 10) the vertical results of the border rhomboids are transferred from the ISEF to the final CC result vector via WRs `*Y_1` and `*Y_2`. It is shown by the internal variable `run` defined in line 4, if the computation of the actual rhomboid is a border case. Horizontal results are transferred to the CC result vector if the computation of a diagonal is finished. The results and the intermediate results have a higher bit width (64 bit) than the input values (16 bit). In lines 13 to 29 the final resource usage report of the ISEF for the diagonal decomposition is presented.

List of Figures

1.1 Development process 11

2.1 Exemplary source localization scenario 13
2.2 Array configuration of the applied microphone array 14
2.3 Simple scenario of source localization 15
2.4 CC between M_1 and M_{13} 16
2.5 Structure of the implementation 19
2.6 Gammatone filters [9] 20
2.7 *Direct Form II* implementation of one Gammatone IIR filter channel. . . 22
2.8 PoPi decomposition 24
2.9 Visualisation of the Position-Pitch matrix. 24
2.10 Projection mismatch 26
2.11 PoPi matrix for one microphone pair 27
2.12 Processing chain of the tracker. 28
2.13 Structure of the time domain implementation 29
2.14 Structure of the frequency domain implementation 30
2.15 The frame mechanism 31

3.1 Time performance measurements 34
3.2 Relative time consumption 35
3.3 Time performance: Variation of frame shift. 37
3.4 Accuracy vs. time performance: Frame shift 38
3.5 Accuracy vs. time performance: Frame size 39
3.6 Accuracy vs. time performance: Sampling frequency 40
3.7 Accuracy and time performance: Filterbank adjustments 42
3.8 Accuracy and time performance: Optimized filterbank adjustments . . . 42
3.9 Accuracy vs. time performance: Input data resolution 43
3.10 Accuracy vs. time performance: Number of microphones 44

4.1 Organization of the Xtensa. 48
4.2 The extended pipeline structure. 49
4.3 Building Process of an ISEF configuration. 55

5.1 The cross correlation algorithm 60
5.2 Principle of square decomposition 62
5.3 Principle of the merging algorithm for square decomposition. 63
5.4 Principle of linewise decomposition 64
5.5 Principle of diagonal decomposition 66
5.6 Data flow in the case of WRs use. 68
5.7 Data flow in the case of IRAM use. 70
5.8 Summary of the Stretch simulations 74

B.1 Picture of the VRC6016 card 81
B.2 VRC6016 board layout 81
B.3 VRC6016 board block diagram 82
B.4 Architecture of the Xtensa processor 82
B.5 Pipeline view of the Stretch IDE 83
B.6 The profiling functionality 84

List of Tables

2.1 Relationship between the DOA and $f_{\max}$ 18
2.2 Angular accuracy for different sampling frequencies. 26

3.1 Programming methods for Matlab code approaches. 33
3.2 Parameters for time performance simulation. 34
3.3 Frame shift: Simulation parameters. 36
3.4 Frame size: Simulation parameters. 38
3.5 Sampling frequency: Simulation parameters. 40
3.6 Filterbank adjustments: Simulation parameters. 41
3.7 Number of microphone pairs: Simulation parameters. 43
3.8 Simulation with the optimized parameter configuration. 45

4.1 Pipeline stages. 48

5.1 Performance of 8 bit Stretch simulations. 71
5.2 Performance of 16 bit Stretch simulations. 72
5.3 Performance of 24 bit Stretch simulations. 73
5.4 Loop unrolling 75
5.5 Data transfer: With/without DMA 75
5.6 Performance Gammatone filterbank 76

List of References

[1] B. Clénet, "Circular microphone array based beamforming and source localization on reconfigurable hardware," Master's thesis, Graz University of Technology, 2010.

[2] J. C. Chen, K. Yao, and R. E. Hudson, "Source localization and beamforming," *IEEE Signal Processing Magazine*, vol. 19, no. 2, 2002.

[3] E. Lleida, J. Fernández, and E. Masgrau, "Robust continuous speech recognition system based on a microphone array," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1998.

[4] M. Brandstein and D. Ward, *Microphone Arrays: Signal Processing Techniques and Applications*. Springer, 2001.

[5] B. Kwon, Y. Park, and Y. sik Park, "Sound source localization in the non free-field condition; spherical platform," in *15th International Congress on Sound and Vibration*, 2008.

[6] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, 1949.

[7] J. Dmochowski, J. Benesty, and S. Affès, "On spatial aliasing in microphone arrays," *IEEE Transactions on Signal Processing*, vol. 57, no. 4, 2009.

[8] R. A. Kennedy, T. D. Abhayapala, and D. B. Ward, "Broadband nearfield beamforming using a radial beampattern transformation," *IEEE Transactions on Signal Processing*, vol. 46, no. 8, 1998.

[9] D. Wang and G. J. Brown, *Computational Auditory Scene Analysis: Principles, Algorithms, and Applications*. Wiley-IEEE Press, 2006.

[10] M. Cooke, "Modelling auditory processing and organistation," PhD Dissertation, University of Sheffield, 1993.

[11] T. J. Cavicchi, "Impulse invariance and multipe-order poles," *IEEE Transactions on Signal Processing*, vol. 44, no. 9, 1996.

[12] N. Ma, "An efficient implementation of gammatone filters," Website, 2006, Available online at http://www.dcs.shef.ac.uk/~ning/resources/gammatone/; visited on January 27th 2011.

[13] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*. Prentice Hall International, Inc., 1999.

[14] K. Williston, *Digital Signal Processing: World Class Designs*. Newnes, 2009.

[15] M. Azaria and D. Hertz, "Time delay estimation by generalized cross correlation methods," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 32, no. 2, 2003.

[16] C. H. Knapp and G. C. Carter, "The generalized correlation method for estimation of time delay," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 24, no. 4, 2003.

[17] M. S. Brandstein and H. F. Silverman, "A robust method for speech signal time-delay estimation in reverberant rooms," in *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1997.

[18] M. Frigo and S. G. Johnson, "Fastest fourier transform in the west," Information available online at http://www.fftw.org/; visited on January 27th 2011.

[19] R. E. Gonzalez, "A software-configurable processor architecture," *IEEE Micro*, vol. 26, no. 5, 2006.

[20] M. Mücke, T. V. Huynh, and W. N. Gansterer, "Evaluation of a reconfigurable hybrid cpu for streaming mixed-precision floating-point algorithms," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 4, 2012.

[21] Stretch Inc., *Stretch SCP Programmer's Reference - Version 1.0*, Stretch Incorporation, 2007.

[22] IBM Corp. and Microsoft Corp., *Multimedia Programming Interface and Data Specifications 1.0*, Issued as a joint design by IBM Corporation and Microsoft Corporation, 1991.