Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze

.

Hardware/Software Codesign in speech compression applications

February 9, 2000
Term Thesis WS 99/00 SA-2000.14

Christian Plessl Simon Maurer

Advisors:
Jonas Greutert, Michael Eisenring

Responsible: Prof. Dr. L. Thiele

Abstract

What began with the introduction of ISDN will be a increasing trend in the future: the *integration of speech and data services*. With the omnipresence of IP based networks it's evident to use these networks for voice communication too.

While the idea of using such *Voice over IP* solutions is evident, the actual implementation is more difficult. Firstly, IP based networks were not designed for the use in speech communication applications e.g. it is hard to ensure a certain quality of service in terms of delay and bandwith. Secondly, speechsignals do require a high data-rate (128kbit/s when using 16bit linear PCM coding at 8kHz sampling rate), thus use of compression is advantageous. Over the last 20 years, considerable improvements in speech compression algorithms have been acheived, starting from simple nonlinear qunatization (A-law, μ-law) to very sophisticated vocoders as G.723.1 which are capable to reduce the datarate needed for transmission to about 4-16 kbit/s while still providing good speech quality [1]. The price for this very efficient compression is a significantly enlarged computation effort. Using such algorithms in real-time applications was only possible with the progress of digital signal processors.

In order to keep the actual telephone devices simple, it is desireable to use ordinary phones and do all compression at the switchboard which acts a as ISDN resp. POTS to IP Gateway. When designing a VoIP Gateway which shall be able to provide about 100 simultaneous connections significant computing power is required.

Starting from the widely used speech compression algorithm G.723.1 our term thesis focussed on the following questions:

1. What are the concepts of G.723.1?
2. What is the computing power required for full-duplex G.723.1 connections?
3. What are the most timeconsuming parts of the algorithm?
4. Could custom hardware (FPGA, ASIC) be useful?
5. Will a system using a combination of DSPs, FPGAs, CPUs be a better solution than a pure DSP system?

In our thesis we try to point out our approach to this problem in the first part. In the second part, we present the results of our work and the resulting conclusions.

Contents

1 Speechcoding - an overview **4**
1.1 Overview of speech coding concepts 4
1.2 Classification of speech coding algorithms 4
1.3 Overview of important coding methods 5
1.3.1 Waveform coding 5
1.3.2 Source coding 6
1.3.3 Hybrid coding 10

2 Description of the ITU-T G.723.1 Recommendation **11**
2.1 About ITU-T 11
2.2 The ITU-T G.723.1 Recommendation 11
2.2.1 Overview 11
2.2.2 Fixed-point implementation 12
2.2.3 Floating-point implementation 12
2.2.4 Copyright and Patents 12
2.2.5 Available implementations 13
2.3 Speechcoding concepts of G.723.1 13

3 Analysis of the Compression Algorithm **17**
3.1 Introduction 17
3.2 Profile results 17
3.3 encoding speed 18
3.4 Function DotProd 19
3.5 Function Find_Acbk 19
3.5.1 General 19
3.5.2 Analysis Results 20
3.5.3 ACBK Gain Table 21
3.5.4 Why this complicated table? 21
3.6 Function Find_Fcbk, Find_Best 23
3.6.1 General 23
3.6.2 Analysis Results 25
3.7 Ideas for acceleration 25
3.7.1 Acceleration of Find_Acbk 25
3.7.2 Acceleration of Find_Best 25
3.7.3 Using parallelity 26
3.7.4 Using own hardware FPGA, ASIC 27
3.8 Conclusions 27

4 Summary and Outlook **30**

A Code of Find_Acbk **32**

B Code of Find_Best **37**

List of Tables

1 list of companies that supplies implementations of the G.723.1 recommendation for TI DSPs 16
2 Profile results on different targets 18
3 Duration of encoding and decoding a sound file on different targets 19
4 C code of DotProd 20
5 Computing complexity of G.723.1 for popular DSPs 26

1 Speechcoding - an overview

Speechcoding and speechcompression is a very broad field, which surely cannot be treated in a term thesis like this in its full scope. We will present here some of the basic ideas of speechcompression and show the different approaches to this problem. In particular we want to point out the ideas of the voice compression standard we did concentrate on CCITT G.723.1.

A lot of books and papers have been written on the subject of speechprocessing including some very interessting, easy-to-read tutorials. The book "Sprachverarbeitung" [2] by Eppinger is a general introduction to speechprocessing in german. It covers not only speechcoding but also speech analysis, speech recognition and speech synthesis. The paper "Speechcoding: a tutorial review" Spanias et al. [3] gives an overview of various methods for speechcoding, including an extensive bibliography on the coding methods discussed in the paper.

The book "Digital Speech" [4] of Kondoz has a different objective. It treates speechcoding for low bit rate communication systems and concentrates on the mathematical concepts and methods which are described in great detail.

1.1 Overview of speech coding concepts

In the following we will restrict our scope to speechcoding in telephony applications since most of the currently used algorithms are designed for this purpose. In this applications, speech is sampled with 8 or 16 bit at 8 KHz and bandlimited to 4 kHz or 3.2 kHz. These samples are coded, transmitted and the original signal is reconstructed at the receiver. If these samples are transmitted by simple linear pulse code modulation (PCM) the data rate is: 8 bit/sample $*$ 8000 samples/sec = 64 kbit/s.

Sampling with 8 bit has to be considered at a lower bound because for achieving good speech quality quantization with 12 or 13 bit is necessary, which leads to datarates of about 100 kbit/s. Therefore it is necessary using speech compression when transmitting highquality speech over lowbitrate channels.

1.2 Classification of speech coding algorithms

Speech coding algorithms can be classified in 3 large classes:

1. Waveform coders

   This class of algorithms doesn't make explicit use of the specific properties of speech. They makes only use of the waveform caracteristics as bandlimitation and try to match the waveform of the signal as close as possible. These methods lead typically to bitrates of 9.6 kbit/s - 300 kbit/s.

2. Source coding, speech specific

   These algorithms make use of models, how speech is actually produced. They are known as "vocoders" (voice coders) and are designed to produce intelligible speech without necessarily matching the original waveform. Usually the speech is analyzed first to determine for instance the spectral charakteristics and the fundamental frequency. In the following steps all the processing is made on these parameters and *not* on the original speech signal. This short set of parameters is transmitted to the receiver and used to synthesize the speech. The achievable datarates are 5 kbit/s and below.

   Our language consists of about 60 phonems which can be coded in 6 bit. Since we can pronounce up to 10 phonems per second the theoretical minimum bitrate is approximately 60 bit/s. This implies that the phonems are recognized, transmitted and synthesized at the receiver. As consequence the characteristic of the voice of the speaker is completely lost.

   The computational effort for source coders is high if compared to waveform coders. So the application of source coders was only possible after the technical progress of digital signal processors in the past couple of years.

3. Hybrid coders

   The name "hybrid coder" leads back to the fact, that this class of coders on the one hand tries to match the waveform of the original signal as good as possible, on the other hand they are making use of the perceptual properties of the ear. Hybrid vocoders are able to achieve bitrates of about 4-16kbit/s. The goal of hybrid coders is to combine the efficiency of source vocoders with the highquality properties of waveform coders.

1.3 Overview of important coding methods

1.3.1 Waveform coding

The most important form of waveform coding is pulse code modulation (PCM). A pulsecode modulated signal is simply the lowpass filtered original analog signal, converted e.g. quantized by a analog/digital converter. Since the human ear perceives loudness in a logarithmic way, a linear quantizer is not optimal. When using a logarithmic quantizer, which provides lower quantization steps for small input signals, an 8 bit logarithmic quantizer can provide the same quality than a 11 bit linear quantizer. Logarithmic quantizers as the μ-law or A-law quantizers as defined in CCITT standard G.711 [5] are very common in telecom applications.

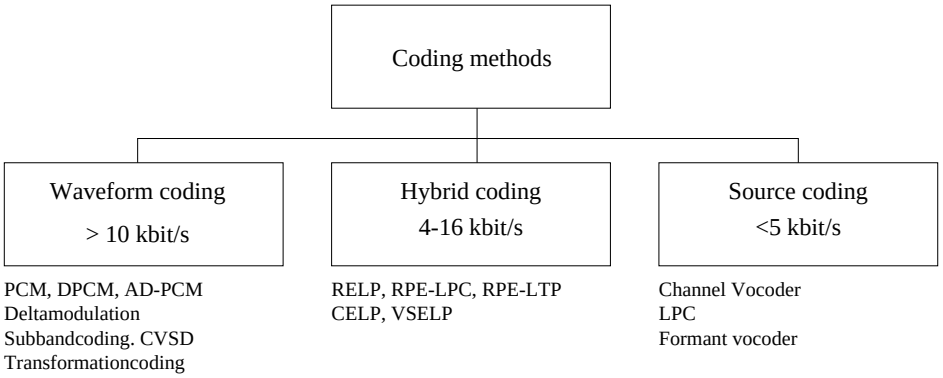

Figure 1: Classification of speechcoding algorithms (Source: [2])

Since a speech signal is inherently bandlimited, the difference between two following speech samples can not be arbitrarily large. Due to this correlation the range of values of this difference is smaller than the range of values of the speechsignal. Therefore it is advantageous to transmit only the differences between the individual samples which can either be coded in fewer bits to reduce the data rate or when using the same bitrate to achiev a better speech quality. This method is know as Differential Pulsecode Modulation (DPCM). When the quantization of the differatial signal is not fixed but adaptive, this method is known as ADPCM (adaptive differential pulscode modulation). ADPCM is frequently used and the base for CCITT standard G.726 [6].

1.3.2 Source coding

Source coding algorithms try to build a model of the generation of speech and process the parameters of this model rather than the real waveform. The parameters of the model are calculated by analyzing the speech input samples in the encoder. At the decoder, speech is reconstructed by synthesis from the parameters of the model as determined by the encoder. A typical vocoder system performs a spectral and "excitation" analysis of the input signal. These parameters are multiplexed and transmitted to the receiver, where the appropriate excitation is used to reconstruct the original using a synthesis filter with the frequency characteristics of the original signal.

Since speech signals are not stationary the analysis of the speech must be repeated in regular intervalls. Therfore the speechsignal is split into segments of 10 to 30 ms and the calculation of the speech characteristic is performed for every segment. The parameters calculated are typically: kind of excitation (voiced/unvoiced), fundamental frequency (pitch) of excitation, amplitude of the excitation and parameters of filter applied to the excitation.

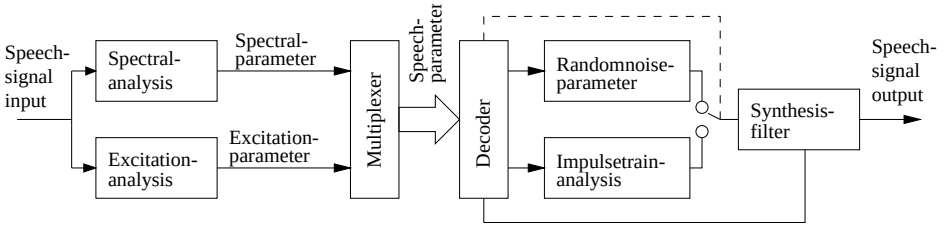

Figure 2: Block diagram of vocoder system (Source: [2])

The parameters are typically calculated as follows:

- Fundamental frequency (pitch) of excitation signal: is determined via the autocorrelation function. The distance between 2 maxima gives the fundamental period. The pitch can also be determined from the spectrum directly.
- Kind of excitation: voiced excitation (periodic) or unvoiced excitation (noisy) can be distinguished by looking at the number of zero-crossings in a given intervall, since noisy signals do have more zero-crossings for a given time intervall than periodic signals. Additionally the kind of excitation can be determined from the distinctness of the fundamental frequency. It is hard to clearly determine the fundamental frequency the excitation is voiceless since the spectrum is noise-like (flat), whilst a voiced excitation typically leads to clear peaks in the spectrum.
- Amplitude of excitation: Can be calculated from the energie of the speech signal.
- Filterparameters: This is dependant on the type of vocoder. Some concepts of calculating this parameters are given in the following paragraphs.

Eppinger [2] mentions 3 different main types of vocoders: Channel vocoders, which splits up the signal in different frequency bands and use the energie of the signal in these respective bands as parameters, formant vocoders which try to find the position an bandwidth of the speech signal and LPC vocoders. LPC vocoders perform a LPC (linear predictive coding) analysis and do represent the filter using LPC-filters. As our coding algorithm under investigation, G.723.1, makes use of the LPC method, we will explain this method in the following paragraph in more detail.

LPC Coding LPC (linear predictive coding) is one of the most powerfull speech analysis and source coding methods. The main idea is to predict the

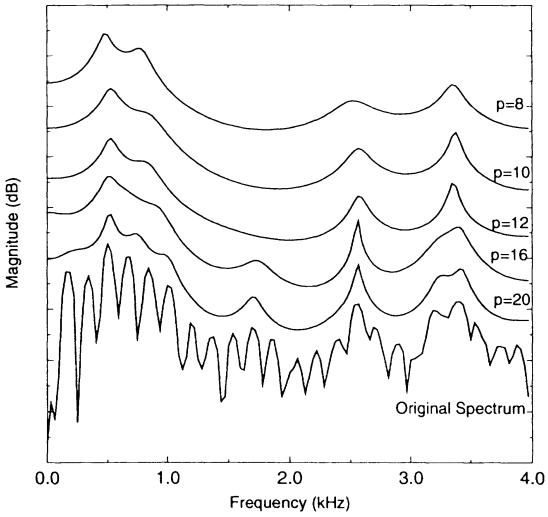

Figure 3: Example of speech spectrum with its LPC representations (Source: [4])

future speech samples from the past speech samples. This would be easy, if the signal were periodic. Of course, speech signals are not periodic, but since speech signals do not change their charactistics in a short time intervall, one considers the signal in a specific segment quasi-stationary. The coming sample $\tilde{s}(n)$ shall be predicted by calculating a linear combination of the past K values $s(n-k)$ with weighting coefficients a_k and the weighted input signal at time n.

$$s(n) = \sum_{j=1}^{K} a_j \cdot s(n-j) + G \cdot x(n) \tag{1}$$

The goal is to find optimal coefficients a_j which predict the future speech samples as good as possible. These coefficients determine the characteristics of a periodic signal. This serves to describe the speech signal in a typical intervall with 12-18 coefficients.

Figure 3 shows the effect of LPC analysis. The higher the order of the LPC filter the better the approximation of the spectral characteristics of the original signal is.

These coefficients can be used for speech coding, when using the following common model of speech generation called the *source filter model of speech production*, see Figure 4.

In this model, speech is modelled as answer of linear system to an excitation by either an impulse train sequence with a given fundamental frequency or to random noise. The transfer function of this very general model is:

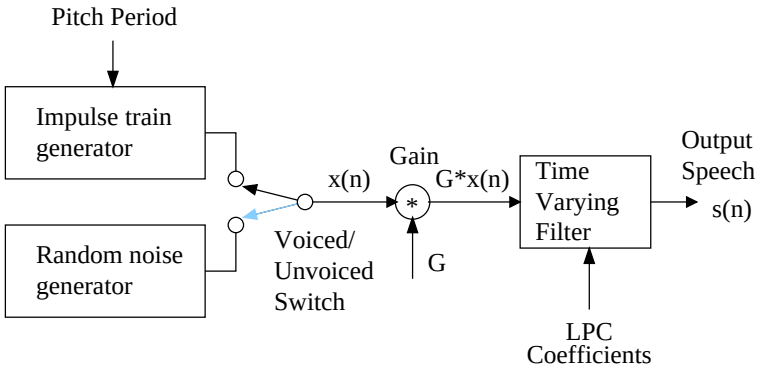

Figure 4: Source filter model of speech production (Source: [4])

$$H(z) = \frac{S(z)}{X(z)} = \frac{G(1 - \sum_{i=1}^{M} b_i z^{-i})}{1 - \sum_{j=1}^{N} a_j z^{-j}} \tag{2}$$

In reality the transfer function has poles and zeros, but if the order of the denominator is high enough, $H(z)$ can be modelled as all-pole filter, with transfer function:

$$H(z) = \frac{G}{1 - \sum_{j=1}^{K} a_j z^{-j}} = \frac{G}{A(z)} \tag{3}$$

If we transform this equation back to the time domain, we obtain the equation

$$s(n) = Gx(n) + \sum_{j=1}^{K} a_j s(n-j) \tag{4}$$

which is analog to the LCP equation we have seen above. Therefore we can formulate the LPC problem as follows: Given the signal $s(n)$, determine the parameters of the transfer function $H(z)$ (LPC coefficients) a_j, $j = 1 \ldots K$.

We try to minimize the mean squared error between the estimated signal and the original signal $s(n)$. The error of the residual is given by

$$e(n) = s(n) - \sum_{j=1}^{K} a_j s(n-j) \tag{5}$$

The expected mean squared error is

$$E = E\{e^2(n)\} = E\left\{ \left[s(n) - \sum_{j=1}^{K} a_j s(n-j) \right]^2 \right\} \tag{6}$$

In order to minimize the mean quared error, we set the partial derivatives $\partial E/\partial a_j$ to zero for $j = 1 \dots K$ an get the following equation system:

$$E\left\{\left[s(n) - \sum_{j=1}^{K} a_j s(n-j)\right] \cdot s(n-j)\right\} \stackrel{!}{=} 0, \qquad \forall j = 1, \dots, K \tag{7}$$

Equation 7 says, that $e(n)$ is orthogonal to $s(n-i)$. We can rewrite equation 7 as

$$\sum_{j=1}^{K} a_j \phi_n(i,j) = \phi_n(i,0), \qquad \forall i = 1, \dots, K \tag{8}$$

where ϕ is

$$\phi(i,j) = E\{s(n-i) \cdot s(n-j)\} \tag{9}$$

There are 3 methods to solve this equation systems, the auto correlation method, the covariance method und the lattice method. For a detailed description of this methods see the corresponding section in [4]. The compression standard G.723.1 uses LPC with 10 prediction coefficients. The equation system 7 is solved via the autocorrelation method by using *Durbin's algorithm*, which is also explained in the book of Kondoz [4].

1.3.3 Hybrid coding

Both, waveform coders and source coders have their respective advantages. Waveform coders provide excellent quality of the reconstructed speech signal but datarate is high. Source coding can provide a noticeable reduction of the datarate but as consequence of the large compression the naturalness of speech gets lost and the speaker is harder to identify.

Hybrid coders try to eliminate this drawback by not only using source coding but also coding the error signal between the original speech and the reconstructed signal of the source coding stage.

Most hybrid systems use linear prediction analysis of the speech signal and differ in the way of coding of the error signal.

2 Description of the ITU-T G.723.1 Recommendation

2.1 About ITU-T

The Internaltional Telecomunications Union (ITU) is a body within the United Nations Economic, Scientific and Cultural Organisation (UNESCO).

The ITU consists of two main bodies: the ITU - Telecomunications Standardization Sector (ITU-T) and the ITU - Radio Standardization Sector (ITU-R). Ironically the ITU does not make standards. These standardization sectors produce documents that are formally known as Recommendations. Within each of these large bodies are smaller groups that focus on specific topics. For example there is a group (Study Group 14 (SG14)), that makes recommendations for modems, such as V.34 and V.32. Study Group 12 (SG12) is charged with studying matters related to network preformance, such as speech quality. The Speech Quality Experts Group (SQEG) within SG12 designs and conducts the subjective testing experiments used to determine the performance of proposed ITU speech coder recommendations. Study Group 15 (SG15) is charged with making recommendations related to speech and video processing, such as speech coding or videotelephony.

2.2 The ITU-T G.723.1 Recommendation

2.2.1 Overview

G.723.1 is a standard which describes a method and implementation of a codec used for transmitting speech and audio signals in multimedia applications at very low bitrates of 5.3 and 6.3 kbit/s. It was designed as part of the very low bitrate visual telephony standard family H.324.

The definition of the ITU-T G.723.1 standard is *not* – as one would naively expect – in the form of a algorithmic mathematical definition or in the form of a signal-flow graph but in the form of C source code. While the document mentions the methods by name they are neither discussed in detail nor is explained what they are good for excactly. The writers obviously assume, that the reader is very familiar with the details of speech compression algorithms and methods.

The description of the speech coding algorithm of this recommendation [7] is made in terms of bit-exact, fixed-point mathematical operations. As part of this recommendation a reference implementation of a an encoder and decoder as well as a test suite is given as ANSI C code, that reflects this bit-exact and fixed-point description approach.

The mathematical descriptions of the encoder and decoder could be implemented in several other fashions, possibly leading to a codec implementation not complying with this recommendation. Therefore, the fixed-point C code of the reference implementation shall take precedence over the mathematical descriptions whenever discrepancies are found.

This kind of description of the algorithm complicates the understanding of every block of the codec and necessitates a close examination of the source code and the available literature on speech coding for low bitrate systems.

G.723.1 operates at 2 different bitrates, 5.3 and 6.3 kbit/s. The high-bitrate and low-bitrate version differ in the way the excitation is encoded. The high-bitrate uses multi-pule maximum likelihood quantization (MP-MLQ) whereas the low bitrate version uses an algebraic-code-excitation (ACELP). The 6.3 kbit/s version provides noticeably better speech quality with reasonable computational complexity, *so we focused on the 6.3 kbit/s bitrate for all our considerations.*

2.2.2 Fixed-point implementation

In the fixed-point implementation a set of so called ETSI-intrinsics are used. These instrisics are function-calls that can easyly be mapped to DSP instructions and a binary code optimized for DSP can be generated.

The fixed point arithmetic version of the reference implementation allows a bitexact simulation of this 16bit fixed point code on any CPU. Therefore the ETSI-instrinsic operations are given as C-functions called *baseops* which are emulating the arithmetic operations with saturation arithmetic, mac instructions, shifts, ... as provided by any common DSP.

For DSP code generation, some C compiler for DSPs include macro definitions, as the Texas Instrument Compilers for TMS320C54x, which translate these ETSI-intrinsics directly into the corresponding native assembler instructions. Consequently it should be possible to generate a quite efficient implementation, without explicit use of inline assembler.

2.2.3 Floating-point implementation

In addition there is a floating point implementation in ANSI C of the code. This implementation is primarily intended to be used with normal microprocessors with a floating-point unit. It's also easier to understand the behaviour of the reference implementation instead of studying the fixed-point implementation, because the reader can concentrate on the algorithm itself and is not disturbed by the specific problems of fixed point computing.

2.2.4 Copyright and Patents

Some contributors and authors have intellectual property rights (copyrights, patents) related to the algorithm used in the G.723.1 speech coder. The intellectual property holders have declared to the ITU-T that they are willing to grant a license to an unlimited number of applications throughout the world under reasonable terms and conditions. If someone manufactures or sells products based on the implementation of this recommendation, he should contact the intellectual property rights (IPR) holders for licence.

2.2.5 Available implementations

Windows 95/98/NT: On Windows the G.723.1 algorithm is included in the basic installation of the OS, as well as other copression algorithms.

TI DSPs: There are several implementations of different companies for the TI TMS320 family. Table 1 lists these implementations with their features.

2.3 Speechcoding concepts of G.723.1

Almost all of the recent coding standards including G.723.1 belong to the class of linear prediction analysis by synthesis coders, for example: G.723.1, G.728, G.729, GSM (full-rate, half-rate, enhanced full-rate) as well as the north american and the japan mobile telephony standards.

In the following paragraph we will give an short overview on how the G.723.1 algorithm is actually working.

The coder operates on 16 linear PCM input samples at a sampling rate of 8000 Hz. The encoding method bases on linear prediction analysis-by-synthesis coding. The input samples are grouped into block of 240 samples, what is equal to 30 ms. These frames are further split up into *subframes* whith a length of 60 samples. For every subframe a Linear Predictive Coding analysis is performed and a 10th order LPC filter is computed. For a brief description of LPC, see section 1.3.2 on page 7. The LPC coefficient are determined via the autocorrelation method using Durbin‘s algorithm as described in [4].

During further processing, the algorithm tries to minimize the error energy between the input signal and the reconstructed signal. This leads to having approximately the same error energy in all frequency bands. Since the human ear is less sensitive to quantisation noise in the frequency bands with the high energy, the perceived noise can be reduced when weighting the frequencies with low energy higher than the bands with higher energy. This means, although noise shaping increases the overall squared error between original and reconstructed signal, it results in a reduction of the perceived error.

A suitable weighting filter can be derived directly from the LPC filter, since the LPC filter describes the spectral enveloppe of the speech signal. A typical weighting filter is shown in figure 5.

This weighting filter is used to filter the complete frame to obtain the perceptually weighted speech signal.

From this point on, the speech is processed on a 60 samples subframe basis. Using the pitch perdiod previously calculated, a harmonic noise shaping filter is constructed. The combination of the LPC synthesis filter, the formant perceptual weighting filter and the harmonic noise shaping filter is

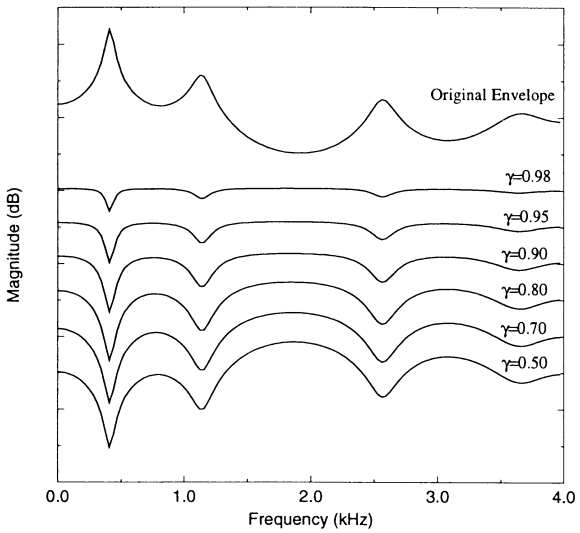

Figure 5: Typical plots of weighting filter spectra compared with the original speech envelope (Source: [4])

used to compute the impulse response of the combined filter. This impulse response is used for further computations.

In the following steps a pitch period estimator is computed, which is used for modelling the periodic excitation. As last step, the effect of the periodic excitation is substracted from the initial signal and the nonperioidic part of the excitation is approximated.

The block diagram, figure 6, taken from the ITU standard shows the coder and the decoder. The numbers in the respective boxes are references to the corresponding sections of the G.723.1 document.

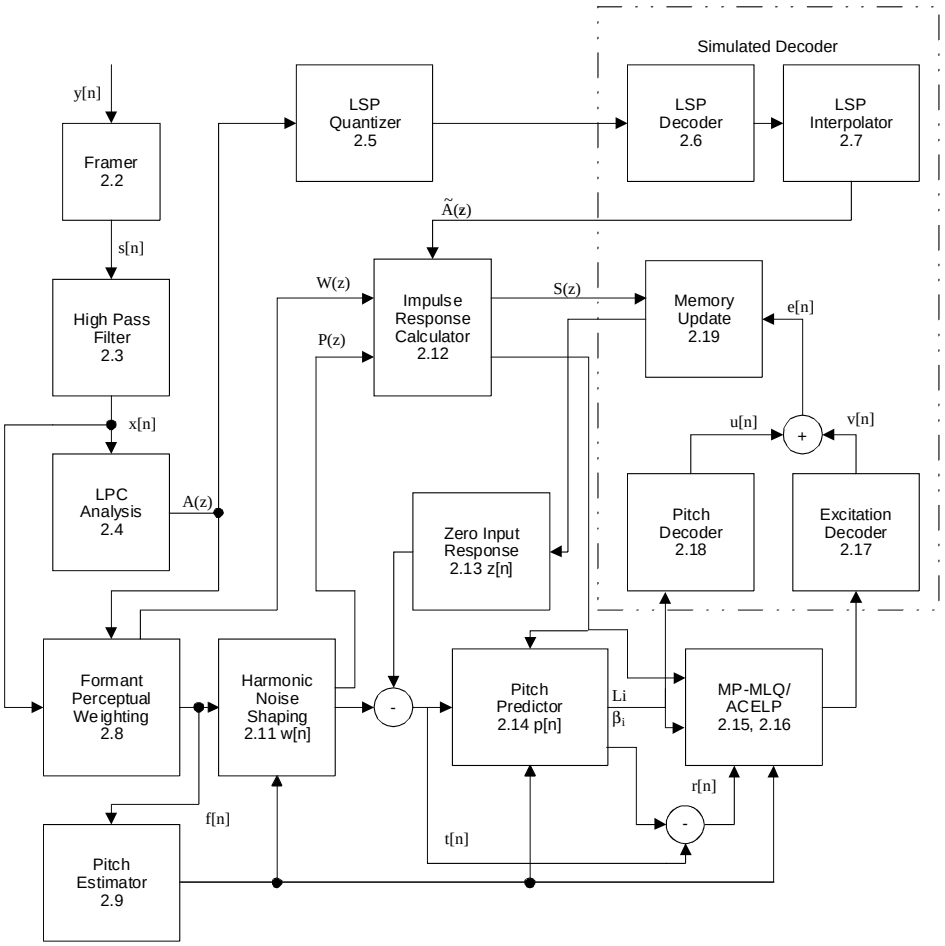

Figure 6: Blockdiagram of G.723.1 (Source: [7])

Supplier	Platform	Channels 6.3 kBit/s	Channels 5.3 kBit/s	MIPS 6.3 kBit/s	MIPS 5.3 kBit/s	Status	Remarks
D2 Technologies, Inc.	TI TMS320C54x	< 18		17.4	16	Production	DPS: 30-532 MIPS
DSP SW Engineering	TI TMS320C62x	20		8		In Development	
RadiSys Corporation	TI TMS320C62x	N/A	N/A	N/A	N/A	Production	from 64 kBit/s
HotHaus Technologies	TI TMS320C3x	< 8				Production	
Signals and Software Limited	TI TMS320C62x	N/A	N/A	N/A	N/A	Production	from 64 kBit/s
FRANCE TELECOM - CNET	TI TMS320C54x	N/A	N/A	N/A	N/A	Production	
MVP Development Group, Inc	TI TMS320C6x	N/A	N/A	N/A	N/A	Production	
DSP Group	Motorola 56156			22	22	Production	

Table 1: list of companies that supplies implementations of the G.723.1 recommendation for TI DSPs

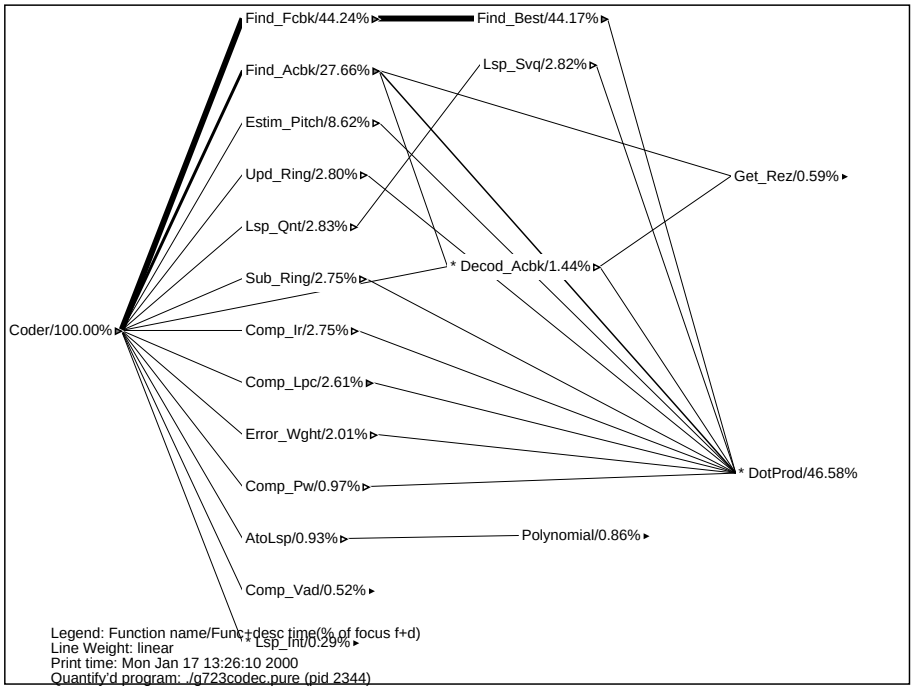

Figure 7: Quantify results of the coder

3 Analysis of the Compression Algorithm

3.1 Introduction

The goal of the analysis was to find thoes code segments, where the most time is spent during execution. Then these segments have been analyzed for the purpose of finding solutions to enhance their preformance. For this analysis the floating-point reference implementation [7] has been choosen (for the analysis on the DSP the fixed-point reference implementation [7] has been taken).

3.2 Profile results

The code has been profiled with MS Visual C++ V5.0 on a Pentium III 550 MHz processor and with Quantify on a SUN Ultra 1 with two 167 MHz UltraSparc RISC processors. The results are shown in table 2. In figure 7 the profile results of the SUN version of the code are illustrated in a graph. The time indications include the time spent in called subfunction. A third processor was a Texas Instrument TMS320C6701 DSP with 166 MHz. Unfortunately there was no satifying profiling tool for the DSP. Therefore only a reduced analysis could be done.

Table 2 shows some interessting differences beween the same program

running on a Pentium III and a UltraSparc. The origin of this differences cannot be explained easily. Especially the differences for Find_Fcbk, Find_Acbk and Lsp_Qnt are noticeable. On one hand these differences are surely an effect of the different architectures of the UltraSparc (RISC) and Pentium III (CISC) CPUs, on the other hand the effect of the compiler used, MS Visual C++ resp. gcc and their capability of code optimization, has an non negligible influence.

Im both cases, it is obviously, that the calculation of the dot product (function `DotProd`) takes almost 50% of the running time of the encoder. Since we are looking for a fast, efficient implementation, we focussed on this function and its callers. The `DotProd` function is described in section 3.4.

The functions that are calling the `DotProd` fuction the most frequently are `Find_Acbk` and `Find_Fcbk` (respectively `Find_Best` in `Find_Fcbk`). Those functions are described in section 3.5 and section 3.6.

Time in ...	Pentium	RISC
Find_Fcbk	17.7%	43.8%
Find_Acbk	31.8%	29.2%
Lps_Qnt	13.4%	2.8%
Sub_Ring	6.4%	2.7%
Comp_Ir	7.5%	2.7%
Comp_Lpc	1.1%	2.7%
Upd_Ring	6.6%	2.6%
Error_Wght	4.8%	1.8%
Comp_Pw	0.8%	1.0%
AtoLsp	2.2%	0.7%
Lps_Int	0.3%	0.3%
DotProd is called by those function above		
DotProd	**44.2%**	**49.2%**

Table 2: Profile results on different targets

3.3 encoding speed

In table 3 the duration of encoding and decoding a file with sampled speech are compared.

It is quiet interesting, that the DSP is not able to encode a speach signal in real time, even though the fixed point reference implementation has been choosen, that is optimised for DSPs as described in subsection 2.2.2. However there are implementations on DSPs, that are able to encode speech signals in real-time (see subsection 2.2.5). That there are significant differences

CPU	CPU-Clock	Compiler	Duration	Duration 1)
Normal Duration			27.7 sec	
Pentium III	550 MHz	MS	3.4 sec	3.7 sec
UltraSparc	440 MHz	sun	5.8 sec	5.1 sec
UltraSparc	440 MHz	gcc	13.5 sec	11.9 sec
UltraSparc	167 MHz	sun	14.8 sec	9.7 sec
UltraSparc	167 MHz	gcc	29.0 sec	4.9 sec
TI DSP 'C6701	166 MHz	TI	42.4 sec	14.1 sec

1) scaled to 500 MHz

Table 3: Duration of encoding and decoding a sound file on different targets

between different compilers can also be seen in table 3. The sun-compiler is able to compile the same code with a much better optimization than the gcc-compiler.

3.4 Function DotProd

As it can be seen in the floating-point code of the function `DotProd` in table 4, the function `DotProd` is simply the dot product of two vectors, e.g. a repetition of multiply-and-add operations. DSPs have hardware support, for this kind of instructions and can handle it in one single clock-cycle (MAC-instruction), thus DSPs are very suitable for this task.

There is no obvious possiblity to speed up the computation of the dot-product, since it consists only of these two main operations, which cannot be accelerated by a clever algorithm. Maybe some improvement could be reached by coding this routine by hand in assembler, since it is executed this often.

So there are two more ideas for a really fast dotproduct routine: Either – this is trivial – using a faster CPU or DSP for computing or – this is more promising – carefully examine which parts of the source code make intensive use of the `DotProd` routine and try to reach the aim due to `DotProd` was called in some other way.

3.5 Function Find_Acbk

3.5.1 General

The goal of this function block is to describe the excitation for current subframe as linear combination of 5 other excitations, which are the time-shifted excitation of the previous frame. The error of this approximation is calculated using the Acbk-Gaintable as explained in the following. The idea behind this procedure is, that two successive subframes typically look

```
FLOAT DotProd(FLOAT *in1, FLOAT *in2, int len)
{
    int   i;
    FLOAT sum;

    sum = (FLOAT)0.0;
    for (i=0; i<len; i++)
        sum += in1[i]*in2[i];

    return(sum);
}
```

Table 4: C code of DotProd

very similar and can be approximated by a time-shifted suframe as a first approach.

Not all possible linear combinations of the time-shifted excitations are allowed. Only the 170 combinations, so called *codebook*, listed in the `AcbkGainTable170` are valid. This reduces the set of parameters that have to be transmitted to the decoder (a kind of quantization). This way only the index of the gain combinations has to be transmitted since the decoder uses the same table.

3.5.2 Analysis Results

In `Find_Acbk` (see appendix A) are several loops, where many multiplications or dot products are done. In two of those loops most time is spent:

At line 107 a convolution of two vectors with length 60 is calculated with regular multiplications. Both vectors are variable. 24.28%[1] of the time in `Find_Acbk` are spent at this position.

At line 179 the dot product of a variable vector with length 20 and a fixed codebook entry is done. Depending on the transmitting rate and other parameters, this is done with 85 or 170 different codebook entries. The codebook is explained in detail in section 3.5.3. 36.45%[2] of the time in `Find_Acbk` are spent at this position. The aim of the this dot product ist to find the minimal square distance of two vectors for quantization.

To accelerate this operations with the codebook, other methods than using the dotproduct should be found, since the whole calculation has already been optimized, as described in 3.5.4. A description of the codebook is in subsection 3.5.3.

[1] taken profiling results for the RISC processor
[2] taken profiling results for the RISC processor

3.5.3 ACBK Gain Table

In this section, the ACBK[3] gain table is explained.

Purpose

This table is used to calculate the error expression in pitch prediction optimisation and is also expressed as an adaptive codebook approach.

Table Structure

As mentioned in section 3.5.2 there are two codebooks with different lenth. The table is structured as 170 (or 85) 20-element vextors. These vectors are structured as precalculated values in the error expression for the pitch predictor. G_i is the gain value multiplying the signal delayed by a pitch period (+/- offset).

1st 5 elements:	G_1	G_2	G_3	G_4	G_5
2nd 5 elements:	$-G_1^2$	$-G_2^2$	$-G_3^2$	$-G_4^2$	$-G_5^2$
Next 10 elements:	These are the off-diagonal elements				
	$-G_1 \cdot G_2$	$-G_1 \cdot G_3$	$-G_2 \cdot G_3$	$-G_1 \cdot G_4$	$-G_2 \cdot G_4$
	$-G_3 \cdot G_4$	$-G_1 \cdot G_5$	$-G_2 \cdot G_5$	$-G_3 \cdot G_5$	$-G_4 \cdot G_5$

3.5.4 Why this complicated table?

The idea of this table is to find as fast as possible the minimal mean squared error between the gains of the five signals and the codebook entries with its precalculated values.
As well known the minimal square distance of two vectors is

$$d_{min} = \min\left((\vec{a} - \vec{b}_i)^2 \; ; \quad i = 1 \ldots 170\right). \tag{10}$$

Therefore, the minimal square distance for the vector combination looks like this:

$$d = \left((G_1\vec{F}_1 + G_2\vec{F}_2 + G_3\vec{F}_3 + G_4\vec{F}_4 + G_5\vec{F}_5) - \vec{I}\right)^2 \tag{11}$$

$$\begin{aligned} d \;\; = \;\; & (G_1\vec{F}_1 + G_2\vec{F}_2 + \cdots + G_5\vec{F}_5)^2 \\ & -2 \cdot \vec{I}(G_1\vec{F}_1 + G_2\vec{F}_2 + \cdots + G_5\vec{F}_5) \\ & +\vec{I}^2 \end{aligned} \tag{12}$$

[3] ACBK: Adative Codebook

$$\begin{aligned} d &= G_1^2\vec{F}_1^2 + G_2^2\vec{F}_2^2 + \cdots + G_1^2\vec{F}_1^2 \\ &\quad +2\cdot G_1G_2\vec{F}_1\vec{F}_2 + 2\cdot G_1G_3\vec{F}_1\vec{F}_3 + 2\cdot G_1G_4\vec{F}_1\vec{F}_4 + \cdots \\ &\quad -2\cdot G_1\vec{F}_1\vec{I} - 2\cdot G_2\vec{F}_2\vec{I} - \cdots - 2\cdot G_5\vec{F}_5\vec{I} \\ &\quad +\vec{I}^2 \end{aligned} \tag{13}$$

In equations 11 to 13 $\vec{F}_i$ are impulse resopnses of filters and $\vec{I}$ is the input signal of the funtion `Find_Acbk` (that isn't the original signal anymore).
Because of the recurrence of several terms (like G_1^2 and $-2\cdot G_1G_2$), they have already been included in the ACBK gain table as described in subsection 3.5.3. So the don't need to be recalcutated each time.
Furthermore all combinations of the dot products of the filters (like $\vec{F}_1^2$ or $\vec{F}_1\vec{F}_2$) are the same for all calculations with different gain combinations of the table. Therefore it doesn't make sense to calculate these products each time. So this is done once at the begining of the function `Find_Acbk` at line 121 to 136. The same applies to the $\vec{I}\vec{F}_i$ terms.
And because we are only interested in the index of the codebook entry and not in the error value itself, it is not necessary to calcultate the $\vec{I}^2$ term.
By subtracting $\vec{I}^2$ from equation 13 and multiplying it with $-\frac{1}{2}$ we get following equation (the factor $\frac{1}{2}$ is to reduce the number of multipications one more time in algorithm):

$$\begin{aligned} max &= G_1\vec{F}_1\vec{I} + G_2\vec{F}_2\vec{I} - \cdots + G_5\vec{F}_5\vec{I} \\ &\quad -\frac{1}{2}G_1^2\vec{F}_1^2 - \frac{1}{2}G_2^2\vec{F}_2^2 - \cdots - \frac{1}{2}G_5^2\vec{F}_5^2 \\ &\quad -G_1G_2\vec{F}_1\vec{F}_2 - G_1G_3\vec{F}_1\vec{F}_3 - G_1G_4\vec{F}_1\vec{F}_4 - \cdots \end{aligned} \tag{14}$$

With this transformation, we have to find the maximal value of "max" instead of a minimal value to find the minimal mean square error.
In the algorithm there are two vectors $\vec{a}$ and $\vec{b}$ with this precalculated terms ($\vec{b}$ is a entry of the code book as describe above). The vector $\vec{a}$ has following structure:

1st 5 elements:	$\vec{F}_1\vec{I}$	$\vec{F}_2\vec{I}$	$\vec{F}_3\vec{I}$	$\vec{F}_4\vec{I}$	$\vec{F}_5\vec{I}$
2nd 5 elements:	$\frac{1}{2}\vec{F}_1^2$	$\frac{1}{2}\vec{F}_2^2$	$\frac{1}{2}\vec{F}_3^2$	$\frac{1}{2}\vec{F}_4^2$	$\frac{1}{2}\vec{F}_5^2$
Next 10 elements:	These are the off-diagonal elements				
	$\vec{F}_1\vec{F}_2$	$\vec{F}_1\vec{F}_3$	$\vec{F}_2\vec{F}_3$	$\vec{F}_1\vec{F}_4$	$\vec{F}_2\vec{F}_4$
	$\vec{F}_3\vec{F}_4$	$\vec{F}_1\vec{F}_5$	$\vec{F}_2\vec{F}_5$	$\vec{F}_3\vec{F}_5$	$\vec{F}_4\vec{F}_5$

The result of the multiplication of $\vec{a}$ and $\vec{b}$ is the same as max of equation 14. Therefore the minimal square distance can be found by finding the maximum of the dot product of $\vec{a}\cdot\vec{b}$.

$$max = \max\left(\vec{a}\cdot\vec{b}_i\,;\quad i = 1\ldots 170\right) \tag{15}$$

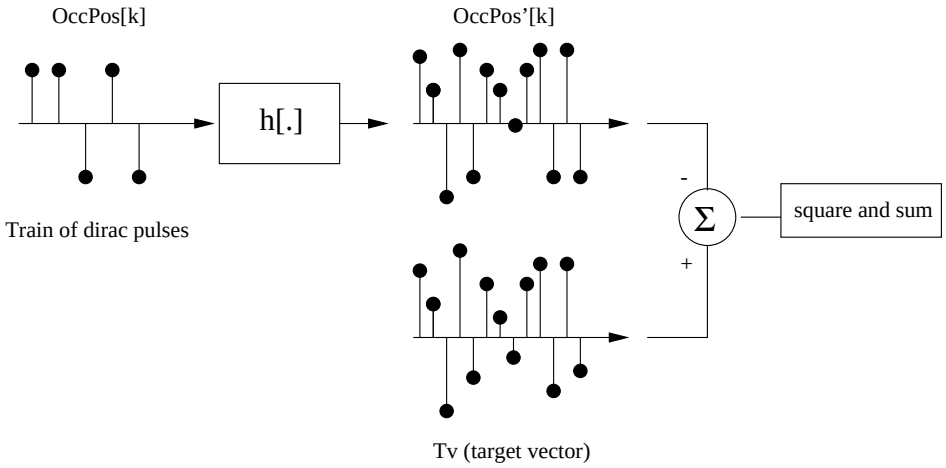

Figure 8: Illustration of the Multipulse Excitation Encoding

This is done at line 177 to 188 of funtion `Find_Acbk`.
In equation 16 all multiplications, that have to be done by the algorithm, are summarized in a matrix-multiplication.

$$\begin{pmatrix} a_1 & a_2 & \cdots & a_{20} \end{pmatrix} \cdot \underbrace{\begin{pmatrix} b_1^{(1)} & b_1^{(2)} & \cdots & b_1^{(170)} \\ b_2^{(1)} & b_2^{(2)} & & b_2^{(170)} \\ \vdots & & \ddots & \vdots \\ b_{20}^{(1)} & b_{20}^{(2)} & \cdots & b_{20}^{(170)} \end{pmatrix}}_{\begin{pmatrix} \vec{b}^{(1)} & \vec{b}^{(2)} & \cdots & \vec{b}^{(170)} \end{pmatrix}} = \begin{pmatrix} \vec{a} \cdot \vec{b}^{(1)} \\ \vec{a} \cdot \vec{b}^{(2)} \\ \vdots \\ \vec{a} \cdot \vec{b}^{(170)} \end{pmatrix}^T \tag{16}$$

3.6 Function Find_Fcbk, Find_Best

3.6.1 General

As we have seen from table 2 `Find_Fcbk` is a very time consuming function. Since `Find_Fcbk` spends almost all of the time in `Find_Best` we focus on `Find_Best` for the following considerations. The source code of `Find_Best` is listed in appendix B.

The goal of the function `Find_Fcbk` is to determine the so called "fixed codebook contribution". The basic idea is to approximate a given target signal by the response of a linear system with given impulse response to an excitation with a fixed number of discrete dirac pulses.

Figure 8 shall illustrate this. The length of the targetvector (Tv) which is the signal which shall be approximated is 60. The problem, how to approximate a given sigal by the response of an linear system excited by dirac

impulses can be solved analytically in the frequency domain. But if there are no restrictions on the number, position and amplitude of the pulses, there is such a high degree of freedom that this set of parameters is large. Since we want to acheive data compression and transmit only as little information as necessary, there have to be introduced significant restrictions on the number, position and amplitude of the pulses to keep the set of parameters that has to be transmitted from the coder to the decoder reasonably compact.

In this application the excitation vector of length 60 has the following restrictions: The number of pulses used for excitation (Np) is limited to 5 or 6 for odd resp. even subframes. The pulses have to occur all either on even or all on odd positions. The pulses must have all the same amplitude with either positive or negative signs. The amplitude is quantized using one of 24 possible levels.

Let $Tv[k]$ be the target vector, $OccPos[k]$ the excitation to be optimized and $OccPos[k]'$ the response, e.g. the first 60 samples, of the system to this excitation. The impulse response of the system is given by $h[k]$. $h[k]$ is the impulse response of the *combined filter*, which is the LPC filter, the weighting filter and the harmonic noise shaping filter as described in chapter 2.3

$$OccPos'[k] = OccPos[k] * h[k] \tag{17}$$

The measure of quality for the approximation is the mean squared error between the approximated and the target signal. The energy of the quadratic error is:

$$\begin{aligned} Error &= \sum_k (Tv - OccPos')^2 && (18)\\ &= (Tv \cdot Tv) - 2(Tv \cdot OccPos') + (OccPos' \cdot OccPos') && (19) \end{aligned}$$

$$Error = \underbrace{(Tv \cdot Tv)}_{\text{fix}, >0} - \underbrace{[2(Tv \cdot OccPos') - (OccPos' \cdot OccPos')]}_{\text{Error is minimimal, if this term is maximal}} \tag{20}$$

This shows that the algorithm has to maximize the term above.

The reference source code shows, how this search for the optimal excitation is performed. Since there are too many possible distribution of 6 pulses on 30 even resp. odd positions for an exhausive search, e.g. testing all possible distributions, an iterative algorithm is used. Only one pulse at once is placed. The pulse is placed at the position, where it is most advantagious. After this, the effect of this pulse, e.g. the response of the system to this single pulse, is substracted from the original target signal and this procedure is repeated for all remaining pulses.

This method leads to a solution which is not necessarily optimal, because only one pulse after the other is set and not all pulses at once. For a exact discussion of the algorithm and its optimality see the corresponding chapter on *Multi Pulse LPC (MPLPC)* in the book of Kondoz [4].

3.6.2 Analysis Results

The hotspot of the excecution of the function `Find_Best` is without doubt the calculation of the convolution of the $OccPos[]$ vector containing the excitation with the impulse response of the combined filter. As we can see from the profiling results for the RISC processor *more than 50%* of the time is used for this convolution, see lines 190-196 in B.

The only computations of further relevance with regard to computational effort are the calculation of the autocorrelation function of the impulse response of the combined filter and the crosscorrelation function of the impulse response and the target vector, lines 90-99 (each of these computations takes about 9% of the excecution time of `Find_Best`). Furthermore, the loop which determines the optimal position and amplitude for the approximation of the excitation (lines 157-172) takes about 16% of the execution time.

3.7 Ideas for acceleration

3.7.1 Acceleration of Find_Acbk

As explained in subsection 3.5 most time is used to calculate some dot products and convolutions. Both hotspots are based on multiplication followed by an addition (MAC-operations). If the reference implementation is used as basis for an implementation, a DSP would be the most efficient solution.

3.7.2 Acceleration of Find_Best

Since the hotspots of `Find_Best` are convolutions, which are classical *MAC operations* in a loop, a processor with a dedicated single-cycle MAC instruction and zero-overhead loops might lead to a faster implementation. The loop overhead of the code compiled for the RISC processor is significant. We determined an loop overhead cause to initialization and loop-counter increment of 16% of the runtime of `Find_Best` just for the convolution of all $OccPos[.]$ vectors with the impulse response $h[.]$.

A further optimizations which depends largely on the CPU resp. DSP used for implementation is the optimization of the convolution. The convolution of $OccPos[.]$ with $h[.]$ is trivial, because $OccPos[.]$ is just the sum of 5 or 6 dirac impulses with positive or negative sing scaled by a constant factor so we can consider $OccPos[.]$ as sum of 5 or 6 time shifted and scaled dirac impulses, e.g.

$$OccPos[.] = a \cdot (\pm\delta[n - k_1] \pm \delta[n - k_2] \pm \ldots \pm \delta[n - k_6]) \qquad (21)$$

As the convolution is a linear function, e.g.

$$(a[k] + b[k]) * h[k] = (a[k] * h[k]) + (b[k] * h[k]) \tag{22}$$

the convolution of a time shifted dirac impulse with the impulse response is simply the time shifted impulse response. Therefore the convolution of *OccPos* with the impulse response is trivial, because it is nothing more than the sum of 5 resp. 6 time shifted and scaled impulse responses.

By rewriting this convolution as a summation over the time shifted impulse responses with scaling one could reach a performance gain.

3.7.3 Using parallelity

When having an algorithm with large computational effort – as G.723.1 surely is – one might think of using parallel computation to speed up the calculation. Parallelisation is particularly interesting, when an algorithm *must be processed in realtime*, but the given hardware (which may be a CPU, DSP, ASIC, etc) *cannot compute the algorithm in realtime* since the computation is too complex. In this case, parallelization might be the only way to speed up processing time besides using faster hardware.

In our specific case the situation is somewhat different. The complexity of G.723.1 is sufficiently low, that any common DSP should be able to process the compression and decompression in realtime. Commercial vendors (as DSP Group) of G.723.1 codecs for various DSPs claim to have highly optimized implementations which require about 20 DSP-MIPS for the encoder and about 2.5 DSP-MIPS for the decoder.

The following table 5 gives an overview over the MIPS requirements for encoding and decoding using G.723.1 at 6.3kbit/s rate for some popular DSPs (Source: DSP Group, http://www.dspg.com).

DSP Family	Coder [MIPS]	Decoder [MIPS]
Motorola 56156	19.4	1.7
TI TMS320C52	21.1	2.6
TI TMS320C50,51,53	19.7	2.2

Table 5: Computing complexity of G.723.1 for popular DSPs

As one sees from Table 5 a typical 16bit fixedpoint DSP as a TMS320C50 at 66 Mhz (33 MIPS) should be able to code and decode one single channel in realtime. More sophisticated highspeed DSPs as the TMS320C62x family with up to 2400 MIPS could be used for multichannel implementations on a single DSP.

In other words, since the realtime constraints are not very demanding, there is no obvious reason to use parallel compution for an *individual* channel. But in our scenario, where dozens of channels shall be compresses

simultaneously, parallel computation on several DSPs surely is one solution to the problem. But even for this scenario the same statement holds true, one channel is preferably computed on a single DSP and several channels can be processed on a single DSP if the computation capacity is high enough.

3.7.4 Using own hardware FPGA, ASIC

The implementation of the complete encoder / decoder on an ASIC or an FPGA imposes some problems. Since the G.723.1 standard is defined as ANSI C Code, it is necessary to port all code from C to VHDL first. This can be quite a tedious and time consuming task, since for a really efficient VHDL implementation one has to do consider, how the heavily used mathematic operations particularly multiplications can be mapped to the available hardware. The efficiency of such an implementation will depend largely on these available building blocks as multipliers, adders and so on.

A major problem when implementing the algorithm on FPGA will be the implementation of 16 bit fixed point arithmetic. Since a lot of multiplications with variable coefficients are required complex multipliers are needed which require many CLBs (about 160 CLBs on a XILINX Virtex FPGA per multiplier). Typically it is difficult to implement fast single-cycle multipliers.

The point is, that the algorithm does *not* use weird operations but rather typical DSP instructions. An optimal implemenation on FPGAs with respect to system cost is hard to imagine.

When considering ASICs the situation look a different as they provide more flexibility, since more chip size is usable for the actual function. An interesting option could be the integrate one or several DSP cores as basic blocks on the ASIC.

3.8 Conclusions

The most promising solution for implementing a voice over IP-Gateway is a DSP based solution. A exemplary implementation could look like Figure 10.

The basic idea is to use a larger quantity of DSPs to compress and decompress the respective channels. Each DSP has statically assigned certain channels to it, which are handled exclusively by this particular DSP. Since efficient, optimized DSP Code requires about 25 MIPS to handle one single channel, even DSPs with moderate performance can be used to process at least one, typically several channels. Thus each DSP processes several channels sequentially and a larger quantity of DSPs are used in parallel to further improve the processing capabilities of the resulting system.

Figure 9 illustrates the proposed method. On the left, the incoming speech samples, aggregated to frames of 240 samples are shown. These sam-

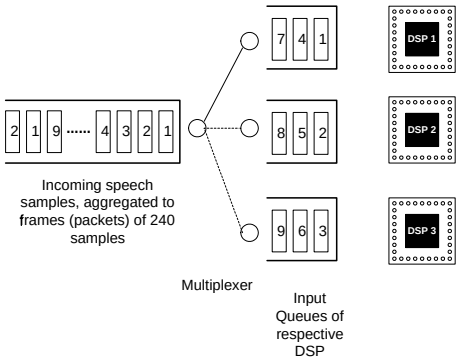

Figure 9: Schematic view of proposed system (9 Channels on 3 DSPs)

ples are distributed by a multiplexer to the appropriate DSP input queues. In this example DSP1 processes channels 1,4,7, DSP2 processes channels 2,5,8 and channels 3,6,9 are processed on DSP3. Naturally the performance of the DSPs has to be chosen such that it is possible to process 3 channels in realtime.

Figure 10 shows a possible implementation of a VoIP-Gateway. The view in the following focusses on the coder but can be easily extended to the decoder too.

The collecting of incoming speech samples from the PCM Highway and the assignment to the respective DSP input queue is done by an FPGA / ASIC or by an appropriate microprocessor which can be directly attached to PCM Highway. As speech data is processed in frames of 240 samples, the incoming samples have to be aggregated. This can be done either by the FPGA, by saving the data to a RAM attached to it and passing all the frame to the DSP at once if it is complete. Alternatively, the aggregation could be done at the DSP by sending each sample to the respective DSP which maintains the appropriate buffers itself. The communication of FPGA and DSP can be implented by using a shared address-/ data-bus and writing the data directly into the memory of the DSP or by using communication ports as synchronous serial interfaces as virtually any DSP provides. The further processing, e.g. compression and decompression is done as explained above. After that, the compressed data packets are passed to a FPGA / ASIC or microprocessor which is reponsible for sending the packets to a local- or wide area network.

Additionally a superior layer for management, accounting and monitoring is needed, denoted by the RISC processor in the figure. This layer is responsible for proper configuration of the DSPs, mapping of the incoming PCM channels to the outgoing network connections, collecting of information for accounting and so on.

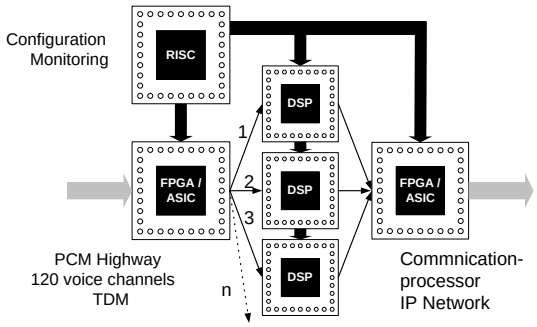

Figure 10: Overview over the proposed System

When implementing the compression algorithm on the DSP one property of G.723.1 can be useful, the runtime behaviour of the algorithm is practically independant of the data. This can be particularily useful when using a architecture for implementation which provides SIMD instructions (single instruction multiple data). Since every channel is processed identically, parallel units of the DSP can be used efficiently and several channels could be processed really in parallel. When building custom hardware (ASIC), for instance using DSP cores, this property can simplify the design for multiple channels.

4 Summary and Outlook

The main goals of our term thesis, analysis of the ITU G.723.1 compression standard, the determination of the computing complexity on different hardware architecture and a proposal for a implementation have been reached.

We encountered major problems when we tried to become acquainted with the details of G.723.1. Since the standard is given primarily as ANSI-C reference implementation with only little additional information, it was a major undertaking to reconstruct the ideas from the source code. Since the understanding of current speech compression algorithms in general and G.723.1 in detail needed more than a third of our time, we presented the results in detail.

The second block of our work was the profiling of the reference implementation on a PentiumIII CISC processor, a UltraSparc RISC processor and on a TI-TMS320C6701 DSP. From these results, the computing intensive parts of the algorithm have been detected and analyzed. We showed that the hotspots are typical DSP blocks as calculations of convolutions and dot-products. We analyzed not only the mathematical operations performend at these hotspots but also the goal of this operations.

We stated, that the use of up-to-date, general purpose CPU, as the PentiumIII, for signal processing applications can yield to suprisingly good results. Further we noticed from our measurements on the UltraSparc CPU, that the performance of the code depends drastically on the compiler used.

A rather disappointing point, was the performance of the code on the TI DSP. Although TI claims to provide a highly efficient ANSI-C compiler, the performance of the resulting code was rather poor. A 160Mhz high performance DSP, with several parallel units was not able to process even one single channel in realtime, whereas commercial providers of G.723.1 code claim to process more then 10 channels in realtime on the same processor.

Finally we presented a possible implementation of a multichannel VoIP-Gateway, based on DSPs.

There are several possiblities to carry on this project:

Although the performance of general purpose CPUs in our application was quite good already, it would be worth while examining if the use of "multimedia instructions" which have been added recently to CPUs as Intel MMX and ISSE, AMD 3Dnow or Motorola AliVec could further improve the performance.

Since the understanding of G.723.1 was very time consuming, we had only little time in the end for optimization of the DSP code. It would be interesting optimizing the DSP code and explore in more depth, how a complete system for voice compression can be realized.

References

[1] Cox, Richard V.: *Low Bit-Rate Speech Coders for Multimedia Communication*, IEEE Communications Magazine, December 1996

[2] Eppinger, B. Herter, E.: *Sprachverarbeitung*, Wien: Hanser, 1993 ISBN 3-446-16076-0, ETHBIB: 756 593

[3] Spanias, A.S. : *Speech coding: a tutorial review*, Proceedings of the IEEE Volume: 82 10 , Page(s): 1541 -1582

[4] Kondoz, A.M.: *Digital Speech: Coding for low bit rate communication systems* John Wiley & Sons, ISBN 0-471-623717, ETHBIB: 775 824

[5] CCITT Standard G.711 : *Pulse code modulaton (PCM) of voice frequencies*, ITU-T Recommendation G.711, (Extract from the *Blue Book*), 1988, 1993

[6] CCITT Standard G.726 *40, 32, 24, 16 kbit/s Adaptive differential pulse code modulation*, ITU-T Recommendation G.726, 1990

[7] ITU-T Standard G.723.1 *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s* ITU-T Recommendation G.723.1, 1996

A Code of Find_Acbk

excerpt from G.723.1 floating point reference implementation, annotations by Quantify profiler

```
1
   /*****************************************************
    * Quantify Annotated Source of /home/cplessl/ms_dev/g723/src_float_chris/exc2.c
    * Data from Quantify'd ./g723codec.pure (pid 1611)
5   * Quantify version: 4.2
    * - Annotation: Function+descendant time (% of f+d time)
    *
    * Legend:
    *
10  *  Lines are annotated with a distinguishing character and collected data.
    *
    *  * - A comment line inserted by Quantify.
    *  | - A line containing a single executed block.
    *  + - A line containing multiple basic blocks.
15  *  . - The extent of any basic blocks over several lines.
    *  # - A line containing basic blocks that were not executed.
    *****************************************************/


20
    **
    ** Function:        Find_Acbk()
    **
    ** Description:     Computation of adaptive codebook contribution in
25  **                   closed-loop around open-loop pitch lag (subframes 0 & 2)
    **                   around the previous subframe closed-loop pitch lag
    **                   (subframes 1 & 3).  For subframes 0 & 2, the pitch lag is
    **                   encoded whereas for subframes 1 & 3, only the difference
    **                   with the previous value is encoded (-1, 0, +1 or +2).
30  *                   The pitch predictor gains are quantized using one of two
    **                   codebooks (85 entries or 170 entries) depending on the
    **                   rate and on the pitch lag value.
    **                   Finally, the contribution of the pitch predictor is decoded
    **                   and
35  ** Arguments:
    **
    **  FLOAT  *Tv      Target vector
    **  FLOAT  *ImpResp Impulse response of the combined filter
    **  FLOAT  *PrevExc Previous excitation vector
40  **  LINEDEF *Line   Contains pitch parameters (open/closed loop lag, gain)
    **  int    Sfc      Subframe index
    **
    ** Outputs:
    **
45  **  FLOAT  *Tv     Residual vector
    **  LINEDEF *Line  Contains pitch related parameters (closed loop lag, gain)
    **
    ** Return value:    None
    **
```

```
50
      void  Find_Acbk(FLOAT *Tv, FLOAT *ImpResp, FLOAT *PrevExc,
                           LINEDEF *Line, int Sfc)
      * *********************************************************
55    * Function:                       Find_Acbk
      * Called:                              152 times
      * Function time:                  25229676 cycles (11.53% of .root.)
      * Function+descendants time:      56142192 cycles (25.66% of .root.)
      * Distribution to Callers:
60    *    152 times Coder
      * *********************************************************
     0.00%| {
     0.00%|     int i,j,k,l;
          .
65        .     FLOAT Acc0,Max;
          .
          .     FLOAT RezBuf[SubFrLen+ClPitchOrd-1];
          .     FLOAT FltBuf[ClPitchOrd][SubFrLen];
          .     FLOAT CorVct[4*(2*ClPitchOrd + ClPitchOrd*(ClPitchOrd-1)/2)];
70        .     FLOAT *lPnt;
          .     FLOAT *sPnt;
          .
          .     int   Olp,Lid,Gid,Hb;
          .     int   Bound[2];
75        .     int   Lag1, Lag2;
          .     int   off_filt;
          .
          .     Olp = (*Line).Olp[Sfc>>1];
     0.00%#     Lid = Pstep;
80   0.00%#     Gid = 0;
     0.00%#     Hb  = 3 + (Sfc & 1);
          .
          .     /*  For even frames only */
          .
85   0.00%|     if ((Sfc & 1) == 0)
          .     {
     0.00%|         if (Olp == PitchMin)
          .             Olp++;
     0.00%|         if (Olp > (PitchMax-5))
90        .             Olp = PitchMax-5;
          .     }
          .
          .     lPnt = CorVct;
     0.00%+     for (k=0; k < Hb; k++)
95        .     {
          .
          .         /*  Get residual from the exitation buffer */
          .
     1.14%+         Get_Rez(RezBuf, PrevExc, Olp-Pstep+k);
100       .
          .         /*  Filter the last one (ClPitchOrd-1) using the impulse responce */
          .
     0.17%|         for (i=0; i < SubFrLen; i++)
```

```
        .         {
105   0.00%#            Acc0 = (FLOAT)0.0;
      5.37%+            for (j=0; j <= i; j++)
     24.28%|                Acc0 += RezBuf[ClPitchOrd-1+j]*ImpResp[i-j];
        .
      0.00%#            FltBuf[ClPitchOrd-1][i] = Acc0;
110     .         }
        .
        .         /*  Update the others (ClPitchOrd-2 down to 0) */
        .
      0.01%|      for (i=ClPitchOrd-2; i >= 0; i --)
115     .         {
      0.02%+            FltBuf[i][0] = RezBuf[i];
      0.69%|            for (j = 1; j < SubFrLen; j++)
      4.70%|                FltBuf[i][j] = RezBuf[i]*ImpResp[j] + FltBuf[i+1][j-1];
        .         }
120     .
        .         /*  Compute the cross products with the signal */
        .
      0.01%|      for (i=0; i < ClPitchOrd; i++)
      4.03%|          *lPnt++ = DotProd(Tv, FltBuf[i], SubFrLen);
125     .
        .         /*  Compute the energies */
        .
      0.01%|      for (i=0; i < ClPitchOrd; i++)
      4.05%|          *lPnt++ = ((FLOAT)0.5)*DotProd(FltBuf[i], FltBuf[i], SubFrLen);
130     .
        .         /*  Compute the between crosses */
        .
      0.00%#      for (i=1; i < ClPitchOrd; i++)
      0.04%+          for (j = 0; j < i; j++)
135   8.07%+                *lPnt++ = DotProd(FltBuf[i], FltBuf[j], SubFrLen);
        .     }
        .
        .     /* Test potential error */
        .     Lag1 = Olp - Pstep;
140   0.00%#  Lag2 = Olp - Pstep + Hb - 1;
        .
      0.02%|  off_filt = Test_Err(Lag1, Lag2);
        .
      0.00%|  Bound[0] =  NbFilt085_min + (off_filt << 2);
145   0.00%|  if (Bound[0] > NbFilt085)
      0.00%#      Bound[0] = NbFilt085;
      0.00%|  Bound[1] =  NbFilt170_min + (off_filt << 3);
      0.00%|  if (Bound[1] > NbFilt170)
      0.00%#      Bound[1] = NbFilt170;
150     .
      0.00%#  Max = (FLOAT)0.0;
      0.00%|  for (k=0; k < Hb; k++)
        .     {
        .
155     .         /*  Select Quantization table */
        .         l = 0;
      0.00%+      if (WrkRate == Rate63)
```

```
        .           {
     0.00%|             if ((Sfc & 1) == 0)
160      .             {
     0.00%|                 if (Olp-Pstep+k >= SubFrLen-2)
        .                     l = 1;
     0.00%#             }
        .               else
165      .             {
     0.00%|                 if (Olp >= SubFrLen-2)
        .                     l = 1;
        .               }
        .           }
170      .          else
     0.00%#             l = 1;
        .
        .           /*  Search for maximum */
        .
175  0.00%+          sPnt = AcbkGainTablePtr[l];
        .
     0.39%+         for (i=0; i < Bound[l]; i++)
        .           {
    36.45%+             Acc0 = DotProd(&CorVct[k*20],sPnt,20);
180      .              sPnt += 20;
        .
     0.76%|             if (Acc0 > Max)
        .               {
     0.01%|                 Max = Acc0;
185  0.00%#                 Gid = i;
     0.00%+                 Lid = k;
        .               }
        .           }
        .       }
190      .
        .       /*  Modify Olp for even sub frames */
        .
     0.00%|     if ((Sfc & 1) == 0)
        .       {
195  0.00%|         Olp = Olp - Pstep + Lid;
     0.00%#         Lid = Pstep;
        .       }
        .
        .
200     .       /*  Save Lag, Gain and Olp */
        .
     0.00%+     (*Line).Sfs[Sfc].AcLg = Lid;
     0.00%|     (*Line).Sfs[Sfc].AcGn = Gid;
     0.00%|     (*Line).Olp[Sfc>>1] = Olp;
205     .
        .       /*  Decode the Acbk contribution and subtract it */
        .
     1.68%|     Decod_Acbk(RezBuf, PrevExc, Olp, Lid, Gid);
        .
210  0.05%|     for (i=0; i < SubFrLen; i++)
        .       {
```

```
      0.05%|          Acc0 = Tv[i];
           .
      1.54%+          for (j=0; j <= i; j++)
215   6.44%|              Acc0 -= RezBuf[j]*ImpResp[i-j];
           .
      0.00%#          Tv[i] = Acc0;
           .      }
      0.00%| }
```

B Code of Find_Best

excerpt from G.723.1 floating point reference implementation, annotations by Quantify profiler

```
/*************************************************************
 * Quantify Annotated Source of /home/cplessl/ms_dev/g723/src_float_chris/exc2.c
 * Data from Quantify'd ./g723codec.pure (pid 1611)
 * Quantify version: 4.2
 * - Annotation: Function+descendant time (% of f+d time)
 *
 * Legend:
 *
 *  Lines are annotated with a distinguishing character and collected data.
 *
 *  * - A comment line inserted by Quantify.
 *  | - A line containing a single executed block.
 *  + - A line containing multiple basic blocks.
 *  . - The extent of any basic blocks over several lines.
 *  # - A line containing basic blocks that were not executed.
 *************************************************************/


 **
 ** Function:        Find_Best()
 **
 ** Description:     Fixed codebook search for the high rate encoder.
 **                  It performs the quantization of the residual signal.
 **                  The excitation made of Np positive or negative pulses
 **                  multiplied by a gain and whose positions on the grid are
 **                  either all odd or all even, should approximate as best
 **
 ** Links to text:   Section 2.15
 **
 ** Arguments:
 **
 **  BESTDEF *Best   Parameters of the best excitation model
 **
 **  FLOAT  *Tv      Target vector
 **  FLOAT  *ImpResp Impulse response of the combined filter
 **  int    Np       Number of pulses (6 for even subframes, 5 for odd)
 **  int    Olp      Closed-loop pitch lag of subframe 0 (for subframes 0 & 1)
 **                  Closed-loop pitch lag of subframe 2 (for subframes 2 & 3)
 **
 ** Outputs:
 **
 **  BESTDEF *Best
 **
 ** Return value:    None
 **
 /

 void  Find_Best(BESTDEF *Best, FLOAT *Tv, FLOAT *ImpResp,int Np,int Olp)
 ******************************************************
```

```
* Function:                      Find_Best
* Called:                             218 times
* Function time:                 75332013 cycles (34.43% of .root.)
* Function+descendants time:     89646701 cycles (40.98% of .root.)
* Distribution to Callers:
*     218 times Find_Fcbk
*********************************************************
0.00%#        {
      .
0.00%|            int     i,j,k,l;
      .           BESTDEF  Temp;
      .
      .           int     MaxAmpId;
      .           FLOAT   MaxAmp;
      .           FLOAT   Acc0,Acc1,Acc2;
      .
      .           FLOAT   Imr[SubFrLen];
      .           FLOAT   OccPos[SubFrLen];
      .           FLOAT   ImrCorr[SubFrLen];
      .           FLOAT   ErrBlk[SubFrLen];
      .           FLOAT   WrkBlk[SubFrLen];
      .
      .
      .           /* Update Impulse responce */
      .
      .           if (Olp < (SubFrLen-2)) {
0.00%#                Temp.UseTrn = 1;
0.04%|                Gen_Trn(Imr, ImpResp, Olp);
0.00%|            }
      .           else {
0.00%#                Temp.UseTrn = 0;
0.03%|                for (i = 0; i < SubFrLen; i++)
0.03%|                    Imr[i] = ImpResp[i];
      .           }
      .
      .           /* Copy Imr */
      .
0.04%+            for (i=0; i < SubFrLen; i++)
0.04%|                OccPos[i] = Imr[i];
      .
      .     /* Compute Imr AutoCorr function */
      .
0.04%|      for (i=0;i<SubFrLen;i++)
6.38%|          ImrCorr[i] = DotProd(&Imr[i],Imr,SubFrLen-i);
      .
      .     /* Cross correlation with the signal */
      .
0.04%|      for (i=0;i<SubFrLen;i++)
6.38%|          ErrBlk[i] = DotProd(&Tv[i],Imr,SubFrLen-i);
      .
      .     /* Search for the best sequence */
      .
0.00%|      for (k=0; k < Sgrid; k++)
      .     {
```

```
      0.00%#          Temp.GridId = k;
105         .
            .         /*Find maximum amplitude */
            .
      0.00%|          Acc1 = (FLOAT)0.0;
      0.00%+          for (i=k; i < SubFrLen; i +=Sgrid)
110         .         {
      0.09%+              Acc0 = (FLOAT) fabs(ErrBlk[i]);
      0.04%|              if (Acc0 >= Acc1)
            .             {
      0.00%|                  Acc1 = Acc0;
115   0.03%+                  Temp.Ploc[0] = i;
            .             }
            .         }
            .
            .         /* Quantize the maximum amplitude */
120   0.00%|          Acc2 = Acc1;
      0.00%|          Acc1 = (FLOAT)32767.0;
      0.00%#          MaxAmpId = (NumOfGainLev - MlqSteps);
            .
      0.00%|          for (i=MaxAmpId; i >= MlqSteps; i--)
125         .         {
      0.12%+              Acc0 = (FLOAT) fabs(FcbkGainTable[i]*ImrCorr[0] - Acc2);
      0.03%|              if (Acc0 < Acc1)
            .             {
      0.00%|                  Acc1 = Acc0;
130   0.02%+                  MaxAmpId = i;
            .             }
            .         }
      0.00%#          MaxAmpId --;
            .
135   0.00%|          for (i=1; i <=2*MlqSteps; i++)
            .         {
      0.30%+              for (j=k; j < SubFrLen; j +=Sgrid)
            .             {
      0.29%|                  WrkBlk[j] = ErrBlk[j];
140   0.06%|                  OccPos[j] = (FLOAT)0.0;
            .             }
      0.00%|              Temp.MampId = MaxAmpId - MlqSteps + i;
            .
      0.01%|              MaxAmp = FcbkGainTable[Temp.MampId];
145         .
      0.02%|              if (WrkBlk[Temp.Ploc[0]] >= (FLOAT)0.0)
      0.00%|                  Temp.Pamp[0] = MaxAmp;
            .             else
      0.00%#                  Temp.Pamp[0] = -MaxAmp;
150         .
      0.01%|              OccPos[Temp.Ploc[0]] = (FLOAT)1.0;
            .
      0.04%+              for (j=1; j < Np; j++)
            .             {
155   0.01%|                  Acc1 = (FLOAT)-32768.0;
            .
      0.08%+                  for (l=k; l < SubFrLen; l +=Sgrid)
```

```
        .                {
     1.84%+                  if (OccPos[l] != (FLOAT)0.0)
160      .                      continue;
         .
     5.48%+                  Acc0 = WrkBlk[l] - Temp.Pamp[j-1]*
         .                                     ImrCorr[abs(l-Temp.Ploc[j-1])];
     0.71%|                  WrkBlk[l] = Acc0;
165      .
     0.00%#                  Acc0 = (FLOAT) fabs(Acc0);
     0.95%|                  if (Acc0 > Acc1)
         .                   {
     0.03%|                      Acc1 = Acc0;
170  0.53%+                      Temp.Ploc[j] = l;
         .                   }
         .               }
         .
     0.10%|              if (WrkBlk[Temp.Ploc[j]] >= (FLOAT)0.0)
175  0.01%|                  Temp.Pamp[j] = MaxAmp;
         .               else
     0.00%|                  Temp.Pamp[j] = -MaxAmp;
         .
     0.04%|              OccPos[Temp.Ploc[j]] = (FLOAT)1.0;
180      .           }
         .
         .           /* Compute error vector */
         .
     0.35%|          for (j=0; j < SubFrLen; j++)
185  0.35%|              OccPos[j] = (FLOAT)0.0;
         .
     0.06%+          for (j=0; j < Np; j++)
     0.06%|              OccPos[Temp.Ploc[j]] = Temp.Pamp[j];
         .
190  0.23%+          for (l=SubFrLen-1; l >= 0; l--)
         .           {
     0.00%#              Acc0 = (FLOAT)0.0;
    11.15%+              for (j=0; j <= l; j++)
    60.52%|                  Acc0 += OccPos[j]*Imr[l-j];
195  0.00%#              OccPos[l] = Acc0;
         .           }
         .
         .           /* Evaluate error */
         .
200  3.32%|          Acc2 = ((FLOAT)2.0)*DotProd(Tv,OccPos,SubFrLen)
         .                 - DotProd(OccPos,OccPos,SubFrLen);
         .
     0.01%|          if (Acc2 > (*Best).MaxErr)
         .           {
205  0.00%#              (*Best).MaxErr = Acc2;
     0.00%|              (*Best).GridId = Temp.GridId;
     0.00%|              (*Best).MampId = Temp.MampId;
     0.00%|              (*Best).UseTrn = Temp.UseTrn;
     0.01%+              for (j = 0; j < Np; j++)
210      .               {
     0.01%|                  (*Best).Pamp[j] = Temp.Pamp[j];
```

```
0.02%+                      (*Best).Ploc[j] = Temp.Ploc[j];
     .                  }
     .              }
     .          }
     .      }
     .      return;
0.00%| }
```