Optimal speech codec implementation on ARM9E (V5E architecture) RISC processor for next-generation mobile multimedia.

Ajay Kumar Bangla*, Vinay M.K, Suresh Babu P.V
eMuzed, #02-03, Jeevan Bhima Nagar Main Road, Bangalore, India – 560075.

ABSTRACT

The mobile phone is undergoing a rapid evolution from a voice and limited text-messaging device to a complete multimedia client. RISC processors are predominantly used in these devices due to low cost, time to market and power consumption. The growing demand for signal processing performance on these platforms has triggered a convergence of RISC, CISC and DSP technologies[3,4,5,6] on to a single core/system. This convergence leads to a multitude of challenges for optimal usage of available processing power. Voice codecs, which have been traditionally implemented on DSP platforms, have been adapted to sole RISC platforms[8,9] as well. In this paper, the issues involved in optimizing a standard vocoder to RISC-DSP convergence platform (DSP enhanced RISC platforms) are addressed. Our optimization techniques are based on identification of algorithms, which could exploit either the DSP features or the RISC features or both. A few algorithmic modifications have also been suggested. By a systematic application of these optimization techniques for a GSM-AMR (NB) codec[1] on ARM9E core[2], we could achieve more than 77% improvement over the baseline codec and almost 33% (worst-case) over that optimized for a RISC platform (ARM9T) alone in terms of processing cycle requirements. The optimization techniques outlined are generic in nature and are applicable to other vocoders on similar 'application-platform' combinations.

Keywords: ARM9T, ARM9E , RISC, DSP, CELP, GSM-AMR(NB), Vocoder

1. INTRODUCTION

The next generation mobile phones will allow users to surf the internet, rapidly download e-mails, music, video and high quality pictures, run java applications and even hold videoconferences on the move. Mobile multimedia will deliver voice, news, video sequences, etc, in new ways that closely match users' needs – the essence of true multimedia applications. Together with the "always-on" and "anytime/anywhere", the mobile multimedia handsets and devices will become an essential part of everyday life. These next generation mobile phones have put further demands on the processing platforms to deliver efficient control capability along with high signal processing performance. Efficient control capability is required to support RTOS, and applications like browers, word processors and protocol stacks. High signal processing ability is required to enable speech, audio, video and other multimedia applications. RISC cores have dominated the mobile platforms, due to low power consumption, small silicon area and low time to market. They have been traditionally known for efficient control capability and for non-DSP applications. There are two ways of achieving the desired control and signal processing performance: 1) integrating RISC and DSP cores into a single SoC[1]; or 2) Enhancing the RISC architecture with DSP functionality[2,5]. This convergence leads to a multitude of challenges for optimal usage of the available processing power.

ARM is the market leader in RISC technology and has become a de facto standard in the mobile phone market. Older ARM processors (architecture version 5T and below) with primitive 8bit multiplier will not be the ideal platforms for these next generation DSP applications. Thus ARM has adopted the second method of enhancing existing RISC cores with DSP functionality by introducing architecture V5E. This broaden it's suitability to applications that require intensive signal processing, whilst retaining the power and efficiency of a high-performance RISC making it a competitive choice for next generation mobile phones. These RISC DSP convergence platforms throw open a completely new paradigm of optimizing scenario. The challenge is in the optimal usage of the DSP features while exploiting the traditional RISC features and working around the constraints. The DSP enhancements are particularly suitable for speech

* ajay@emuzed.com; phone: 91-80-5252224; www.emuzed.com

codecs. Even on the multimedia phone, speech codec will be the most frequently used DSP application, as it will be used for Push-to-Talk, voicemail, MMS and videoconferencing, apart from conventional voice communication. Optimal implementation of speech codecs will reduce the power consumption, which means longer battery life. In applications such as MMS and videoconferencing, where speech and video codecs operate simultaneously, optimal speech codecs makes high-quality video shots a reality. The GSM-AMR(NB) (Adaptive Multi Rate) speech codec standard[1] has been selected by the Third Generation Partnership Projects (3GPP) for evolved GSM, UMTS and WCDMA networks due to its near-wireline quality and efficient spectrum usage.

This paper describes the optimal implementation of GSM-AMR(NB) codec on ARM9E core which is based on the V5E architecture. Section 2 briefly describes GSM-AMR(NB) speech coding process. Section 3 gives an overview of the ARM V5E architecture. Section 4 describes the optimization techniques in detail. Results are presented in Section 5. Conclusions are given in Section 6. References are listed in Section 7.

2. SPEECH CODING FOR MOBILE COMMUNICATION

In communications, speech coding enables efficient usage of bandwidth by representing speech with a minimum number of bits while maintaining its perceptual quality. Most notable and popular technique for speech coding is code-excited linear predictions (CELP), which is attributed to Schroeder and Atal[10]. CELP essentially broke the 9.6 kbps barrier, which was considered for years as the lower boundary for communications quality speech. Different variants of CELP have found their way into different national and international standards.

GSM-AMR (NB) is the voice codec that has been mandated for use in the third generation wireless networks. The AMR system adapts speech and channel coding rates according to the quality of the radio or network channel. It uses eight source codecs with bit-rates ranging form 12.2 to 4.75 kbit/s. The coder operates on speech frames of 20ms corresponding to 160 samples at the sampling frequency of 8000 samples/s. It performs the mapping from the input blocks to 160 speech samples in 13-bit uniform PCM format to encoded blocks, and from encoded blocks to output blocks of 160 reconstructed speech samples. The coding scheme for the multi-rate coding modes is the Algebraic Code Excited Liner Prediction Coder (ACELP). At each of the 160 speech samples, the speech signal is analyzed to extract the parameters of the CELP model (LP filter coefficients, adaptive and fixed codebooks' indices and gains). The parameters are encoded and transmitted. At the decoder, these parameters are decoded and speech is synthesized by filtering the reconstructed excitation signal through the LP synthesis filter. The encoder block diagram is shown in Fig. 1.

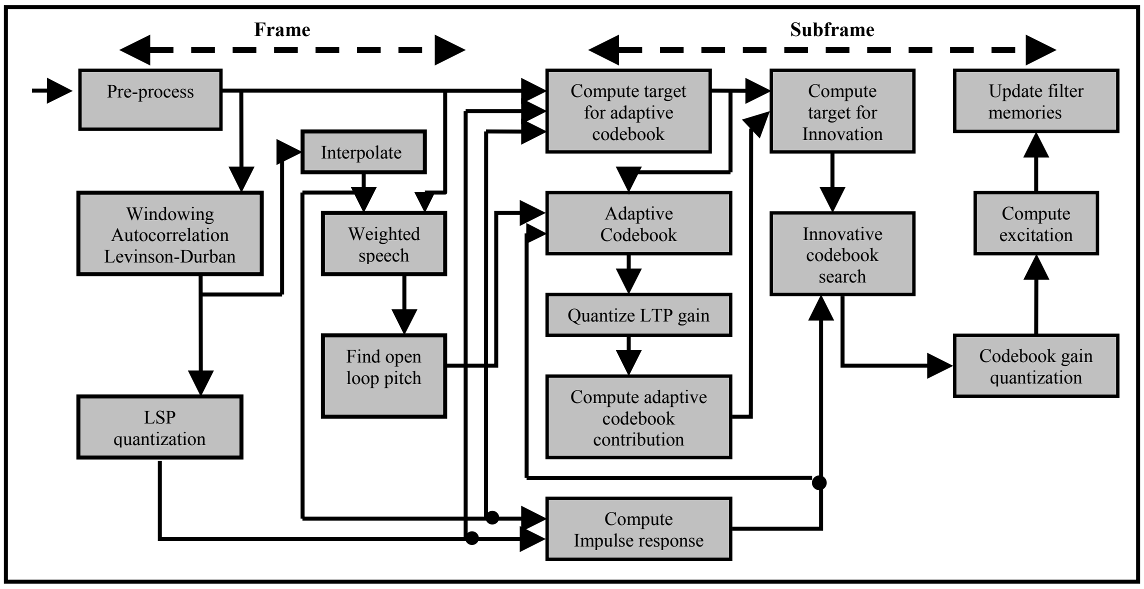
Frame
Subframe
Pre-process
Interpolate
Windowing Autocorrelation Levinson-Durban
Weighted speech
Find open loop pitch
LSP quantization
Compute target for adaptive codebook
Adaptive Codebook
Quantize LTP gain
Compute adaptive codebook contribution
Compute Impulse response
Compute target for Innovation
Innovative codebook search
Update filter memories
Compute excitation
Codebook gain quantization

Fig. 1. Block diagram of the GSM-AMR (NB) encoder.

3. ARM V5E ARCHITECTURE

The ARM9 core[8] based on V4 architecture has reasonable support for DSP – Harvard memory architecture, a multicycle 32x32 bit MAC instruction and conditional execution to minimize branch penalties. ARM V5E builds upon this architecture to further enhance the signal processing performance. This architecture has been implemented in ARM9E core. The new DSP instructions introduced in ARM9E are listed in Table 1. They include single-cycle 16x16 and 32x16 MAC instructions which allow independent access to the top half, bottom half or the full word of the source registers; zero overhead saturation to existing arithmetic instructions and CLZ, an instruction that supports faster normalization by counting leading zeros. Apart from new instructions, there is a Q flag in status register, which indicates saturation or overflow during calculations based on only the new arithmetic instructions.

Instruction	Operation	Purpose
SMLAxy{cond}	16 x 16 + 32 → 32	Signed MAC
SMLAWy{cond}	32 x 16 + 32 → 32	Signed MAC wide
SMLALxy{cond}	16 x 16 + 64 → 64	Signed MAC long
SMULxy{cond}	16 x 16 → 32	Signed multiply
SMULWy{cond}	32 x 16 → 32	Signed multiply long
QADD Rd, Rm, Rs	SAT(Rm + Rs)	Saturating add
QDADD Rd, Rm, Rs	SAT(Rm + SAT(Rs x 2))	Saturating add double
QADD Rd, Rm, Rs	SAT(Rm + Rs)	Saturating add
QDADD Rd, Rm, Rs	SAT(Rm + SAT(Rs x 2))	Saturating add double
CLZ{cond} Rd, Rm	COUNTZ{Rm}	Count leading zeros

Table 1. ARM9E DSP-enhanced extensions

The hardware architecture to support the DSP-enhanced extensions is based on the existing ARM9T RISC core. There are no register or state additions, and no restrictions on register usage.

4. OPTIMIZATION TECHNIQUES

Along with the documentation describing the GSM-AMR (NB) standard, ETSI provides an ANSI C code corresponding to a fixed-point fractional arithmetic implementation of the codec. In this code, all the basic mathematical operations are fixed-point saturated arithmetic operations hence they are simulated with small functions taking care of overflow and underflow conditions. As the reference code is replete with these basic functions, direct cross compilation would result in a highly un-optimized version, because of huge function call over-head with redundant saturation checks. This initial version of the standard code was adapted, by avoiding these function calls. By using additional knowledge about the data values and their ranges, many saturation checks were found to be redundant and hence calls to these basic functions where replaced by regular C operators. Further these arithmetic functions were optimally implemented for ARM9E as described in Section 4.1.1.2. This version was baselined and referenced for all further comparisons.

Profile information of the baselined code provides a good estimate about the relative contribution of different modules to the total computational complexity. Profiling information helped identify critical portions of the code, which were hand assembled to improve the performance. This section describes the techniques used in optimizing (both hand-assembling and change to the C code) various modules. These techniques can be broadly classified into two categories; those based on Processor architecture (ARM9E) and those based on the algorithm of the application (GSM AMR (NB) speech codec). ARM9E being a DSP enhanced RISC processor; we have further classified the techniques into those specific to DSP features and those specific to RISC features of ARM9E.

Table 2 shows the profile of the baseline encoder for the modes 12.2 and 6.70kbps. It also gives the type of computational requirement needed for different modules. Around 65% of the encoder could utilize the signal processing capabilities of ARM9E and the remaining 35% could be benefited by its control capability.

Module	Time Complexity (%)		Computational Requirement
	12.2kbps	6.70kbps	
Fixed Codebook Search	34	32	Control + DSP
Open-loop pitch lag	14	14	DSP
Adaptive Codebook	10	11	DSP
LpcAnalysis	10	6	DSP+Control
LSP Quantization	9	9	DSP+Control
Target and Impulse response	9	9	DSP
Vad	4	4	Control
PitchGain	3	3	DSP
Weighted speech	3	3	DSP
SubFrPost Processing	2	2	DSP
Gain Quantization	1	6	Control
Pre processing	1	1	DSP

Table 2. Profile information of the baseline encoder.

4.1. Architectural optimizations

These optimizations are based on the architectural features of ARM9E. Optimal implementation is possible if both the signal processing and traditional RISC capabilities are appropriately utilized according to the need of the algorithms. The techniques in Section 4.1.1 are possible due to the DSP enhancements in ARM9E and hence, are applicable to ARM9E based systems only. The techniques in Section 4.1.2 are based on the common features of the ARM RISC processor family.

4.1.1. ARM9E specific optimizations

The new instructions introduced in the ARM9E bring it close to a DSP processor. These instructions were used to optimize those sections of the code which require typical DSP type of functionality such as filtering, energy computation etc. The relevant new instructions and their implications are discussed in this section

4.1.1.1. New multiplication instructions

The set of new MAC instructions have a very significant implication on the overall implementation of the speech codec. Typically speech codecs are designed for 16-bit DSP processors. GSM-AMR (NB) is no exception and deals with only 16-bit data. Apart from being single cycle, the new MAC instructions can select the upper or lower half words of the source registers without any need for masking or shifting. These features give rise to the following possibilities:

1. The capability to select the upper or lower half words allows two 16-bit operands for MAC to be in the same register. This can be exploited in two ways:

    a. Efficient usage of the 32-bit bandwidth to load 16-bit packed data: If the operands are in consecutive memory locations, they can be loaded in one register in one memory cycle using LDR instruction. Fig. 2 illustrate the application of this method to energy computation of a N-point sequence. Modules accessing contiguous memory locations for processing are most suitable for this technique. The usage of this technique in such modules reduces the number of data memory accesses and load instructions by almost 50%.

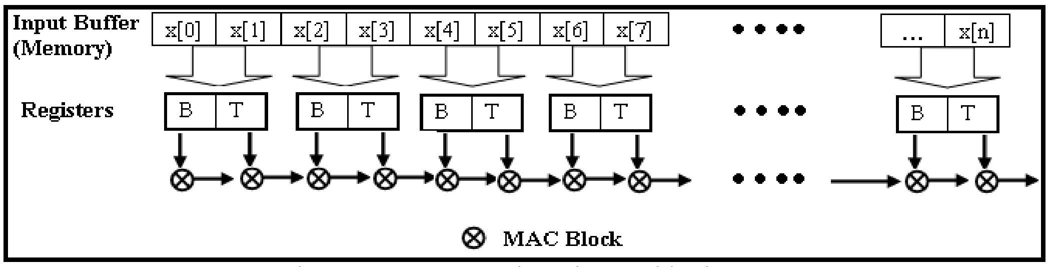

Fig. 2. Energy computation using word-loads

This technique when extensively applied to operations such as filtering, correlation, convolution and energy computation resulted in a reduction of over 70000 memory reads per frame. Apart from reducing the number of memory access and instructions, word load instruction does not induce pipeline stalls associated with half word loads. In case of ARM processors, half-word loads induce pipeline stalls if the loaded registers are used immediately. Further multiple load and store instructions, which reduce the code size, operate only on word data and not on half-word data.

b. Efficient usage of registers to hold 16-bit data: The number of registers required to hold 16-bit data/constants/coefficients is halved as two 16-bit data values can be packed into a single register. Thus freed registers can be used in different ways to further enhance the performance.

   i. In any filtering operation, of the total memory reads, 50% are for reading coefficients and 50% for reading from the input buffer. For filters of small order, the coefficients can be held in the register before starting the filtering operation. This would result in a 50% reduction in the total memory reads.

   ii. Block processing: Filtering and convolution operations can be implemented to compute two or more outputs per iteration. Fig. 3 depicts the computation grouping for a four-tap FIR filter. Outputs y(k) and y(k–1) are computed in parallel i.e., in the same iteration. For the first term in each of these two rows, the partial products a_0x(k) and a_0x(k–1) are computed. Proceeding to the second term in each row, a_1x(k–1) and a_1x(k–2) are computed similarly, and so on with the remaining terms. If the filter coefficients are held in two registers, then in a single sample FIR implementation, a total of 8 loads (half-word) will be required while in a block implementation only 5 half-word loads are necessary to compute two outputs.

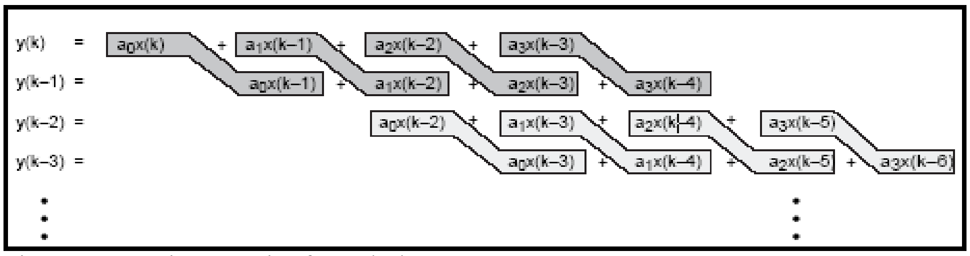

Fig. 3. Computation Grouping for a Block FIR

Using word loads, the total memory reads in block implementation can be reduced to three. Block operations also allow for more efficient instruction sequencing to reduce the pipeline stalls due to data dependencies. By interleaving the instructions corresponding to different outputs, the stalls can be minimized. Further, the loop overhead is reduced by N counts, where N is the number of outputs per iteration. These techniques have been applied to other modules such as convolution and correlation computations.

2. In architectures prior to V5E, due to the lack of single cycle MAC (31x16, 16x16), reducing MAC operations was one of the optimization techniques employed. On these architectures, it would be more efficient to implement (a*b)+(a*c) as a*(b+c). With single cycle MAC on ARM9E, cycles required for both of these implementations will be same. With added advantage of using 16-bit data values in a 32-bit register, it could be more advantageous to use the former implementation. For example in quantization of LSP residues (as given in equation 1), by packing two coefficients and data values in a single register, we can perform two-quantization, with one set of load. The LSF residual vector is quantified using split vector quantization. The weighted LSP distortion measure used in the quantization process is

$$E_{LSP} = \sum_{i=m}^{n} \left[f_i w_i - \hat{f}_i^{k} w_i \right]^2 \qquad (1)$$

On ARM9T and previous architectures, the partial products were calculated as $w_i(f_i - \hat{f}_i^{k})$ to reduce one multiplication. On ARM9E, the most optimal way to compute the partial product would be $f_i w_i + ({`}\hat{f}_i^{k})w_i$

where $`\hat{f}_i^k = -\hat{f}_i^k$. This would allow efficient usage of registers to pre-store the coefficients similar to as explained for filtering. Also, word-loads can be used to read the codebook entries.

4.1.1.2. Saturated Signed Arithmetic

In speech codecs, saturated arithmetic is very frequently used operation. In the earlier architectures, saturated arithmetic had to be simulated in software. The ARM9E has two hardware saturation blocks. One saturation block performs a double and saturate, required for fractional MAC (Q15 x Q15 + Q31→Q31); the other performs a straight saturation of the accumulated value. These instructions are used to optimize the arithmetic functions. Table 3 shows the reduction in the instructions required for implementation as compared to that required for ARM9T. The reduction in basic arithmetic functions' sizes makes inlining a favorable option as the increase in the code size is negligible.

4.1.1.3. Count Leading Zeros

CLZ is a single cycle instruction to count the leading number of zeros in the source register. This instruction will aid the normalization operation, as it reduces the code size and improves speed drastically. The reduction in the worst-case complexity for norm_l (32-bit) and norm_s (16-bit) is illustrated in the Table 3. On ARM9T, this instruction has to be simulated by bitwise check resulting in sizable code. Further, for positive values such as energy, the normalization is possible in just two cycles.

Arithmetic Functions	ARM Optimized	
	9T	9E
add(), sub()	5	4
L_add(), L_sub()	3	1
L_mult(),	4	2
L_mac(), L_msu()	7	2
round()	4	2
norm_s()	62	7
norm_l()	139	7

Table 3. Instructions/Cycles(worstcase) required for arithmetic functions.

4.1.1.4. Q Flag

ARM9E incorporates a mechanism to determine whether saturation or overflow has occurred in the course of a calculation. CSPR has a sticky overflow flag called Q flag which is set if saturation results due to any of the new arithmetic instructions. The major limitation in the usage of the Q flag is that it cannot be used for conditional execution of instructions. In spite of the overhead involved in its usage, it was found to be beneficial in a few cases. Before the computation of the autocorrelation for estimating the LP coefficients, the input signal has to be appropriately scaled if the partial sum during energy calculation overflows. Since, speech is downscaled during preprocessing, overflow occurs doesn't occur very frequently. Q flag allows the saturation check to be moved out of the energy computation loop. This reduces two instructions per multiplication within the loop.

4.1.2. Standard RISC Optimizations

Other than the ARM9E specific optimization, optimization techniques applicable to ARM family were also incorporated.

4.1.2.1. Efficient use of extra precision offered by processor

Standard reference GSM-AMR (NB) codec implementation is for a 16-bit DSP, hence all intermediate 32-bit results are stored and maintained in Double Precision Format (DPF), where a 32-bit value is stored as two 16-bit values such that L_32 = hi<<16 + lo<<1. But ARM9E core has 32 bit registers and can performs all operations on 32-bit data operands. Thus it would be more efficient to maintain all intermediate results as they are and avoid the overhead of deliberately converting back and forth from DPF. Except for few additional instructions to maintain bit compliance of the codec, this technique not only reduces conversion overhead but also allows for efficient utilization of the available bandwidth.

4.1.2.2. Loop unrolling

In ARM processors, there is no support for zero overhead looping as in most DSP processors. Looping overhead in software is considerable and is directly proportional to the number of times the loop is executed. Unrolling the inner loops reduces the function looping overhead significantly, as loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears. Unrolling reduces the loads or data movement required. To use the technique in section 4.1.1.1 some amount of unrolling is essential. If word-loads are to be used, then the inner and outer loops must be unrolled at least twice. If the entire filter coefficients are to be held in registers prior to filtering operation, then the inner loop must be fully unrolled.

4.1.2.3. Conditional Execution of Instructions

The ARM state, all the instructions are conditionally executed according to the state of the Current Program Status Register (CPSR) condition flags and the instruction's condition field. This feature combined with the ability of all arithmetic, logical and data move instructions to modify these condition flags can be used to eliminate use of comparison and branching operations. The loop termination condition can cause significant overhead if written without caution. Whenever possible count down to zeros loops were used. In small if/if else statements, branch statements were eliminated using this feature. This feature was fully exploited to implement div_s(), 16-bit fractional integer division, using successive subtractions. Modulo addition was also implemented using this feature.

4.1.2.3. Post Indexed Addressing

In vector operations, at many places the index is recalculated inside the loop which could easily be replaced by post indexed addressing load/store instructions removing the arithmetic instructions for calculating the displacement from the index base by exploiting the order inherent in the vector elements. This technique reduces one ADD or SUB instruction per load/store instruction.

4.2 Algorithmic Optimizations

The requirement of output bit compliance leaves very little scope for any major algorithmic changes. However, a few instances could be identified and were appropriately optimized to either reduce the computation complexity or the memory access.

4.2.1. Fixed Codebook Search Optimization

In the 12.2kbps mode, there are five tracks with 2 pulses per track. First for each of the five tracks the pulse positions with the maximum absolute values of signal b(n) are searched. From these the global maximum value for all the pulse positions is selected. The first pulse i0 is always set into the position corresponding to the global maximum value. Next, four iterations are carried out. During each iteration the position of pulse i1 is set to the local maximum of the track. The rest of the pulses are optimally searched in pairs by sequentially searching each of the pulse pairs {i2,i3}, {i4,i5}, {i6,i7} and {i8,i9} in nested loops. Every pulse has 8 possible positions, i.e., there are four 8x8-loops, resulting in 256 different combinations of pulse positions for each iteration. Before starting the next iteration starting positions of all the 9 pulses are cyclically shifted, so that the pulse pairs are changed and the pulse i1 is assigned to the local maximum of a different track. Fig. 4.a shows the order of search assuming that i_0 is fixed in the 1st track. Only the pulses shaded in grey are searched. The bold boxes enclose the tracks in which pulses are searched in pair. It was found beneficial to perform two iterations in parallel, as it would directly reduce the number of loop overheads by half. The search in pairs being optimal one allows the inner and outer loops to be interchanged i.e., the tracks of $\{i_n, i_{n+1}\}$ can be interchanged.. Fig 4.b shows the track allocation after interchanging the tracks in 1st and 3rd iteration. The bold boxes show the tracks in which pulses are being searched simultaneously. Interchanging allows two iterations to search a common track. This allows some of the values of the correlation matrices **d** and **ǿ** to be reused during the computation of the search criterion, Q_k.

Iterations	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	1	2	3	4	5	1	2	3	4	5
2	1	3	4	5	1	2	3	4	5	2
3	1	4	5	1	2	3	4	5	2	3
4	1	5	1	2	3	4	5	2	3	4

Iterations	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
1	1	2	4	3	1	5	3	2	5	4
2	1	3	4	5	1	2	3	4	5	2
3	1	4	1	5	3	2	5	4	3	2
4	1	5	1	2	3	4	5	2	3	4

(a) (b)

Figure 4. a) The track allocation assuming that i0 is fixed in 1st track. b) The track allocation using the suggested method.

This method can be used for 10.2 kbps mode as well. Modes 7.95, 7.40 and 6.70 kbps use a slightly different non-exhaustive search while the other lower modes use an exhaustive search. This method can be used for other modes as well but since the number of pulses is very less, the codesize vs. performance tradeoff might not be favorable.

4.2.2. Quantization of gains

In mode 12.2kbps, the pitch gain $\hat{g}_p$ and the codebook gain correction factor γ_{gc} are scalar quantized using 4-bit and 5-bit codebooks respectively. Minimizing the error is the search criterion used.

$$E = \left| g_p - \hat{g}_p \right| \quad (2)$$

$$E = \left| g_c - \hat{\gamma}_{gc} g'_c \right| \quad (3)$$

Once the optimum value of $\hat{\gamma}_{gc}$ is chosen, the quantified fixed codebook gain is given by $\hat{g}_c = \hat{\gamma}_{gc} g'_c$

The entries in both the codebooks are stored in increasing order, which results in U shaped error function. Plots of the error functions obtained during pitch gain quantization are shown in Fig. 5. Similar curves can be expected during codebook gain quantization. Hence, there is only one minima, which is the global minima. Hence the search criterion can be modified to detect sign change in the slope of the error function.

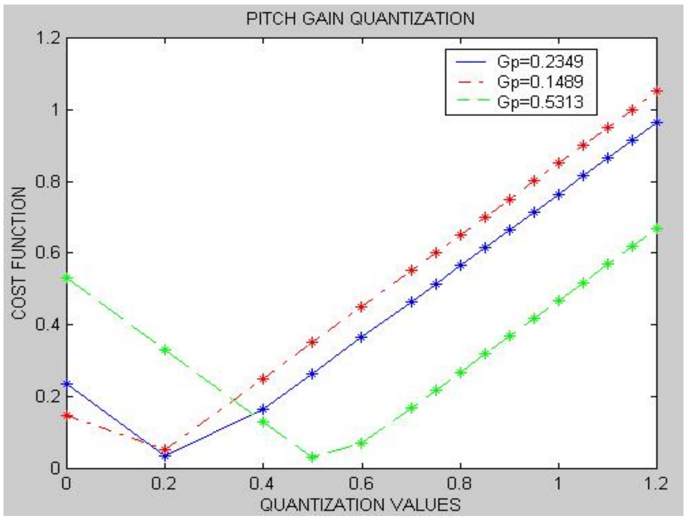

Fig. 5. Error curves during quantization of PitchGain

This technique can also be applied to the 7.95kbps mode as it uses the same scalar codebooks but a different search method. For the other modes, the gains are jointly vector quantized making the error function two-dimensional. The entries are not ordered and hence, there could be several local minima.

4.2.3. Loop fusion

Loop fusion is a process of combining multiple loops, which reuse the same data, in to one loop. The advantage of this is that data can be reused, reducing the memory access, and loop overhead is reduced. For example, scaling and energy computation can be fused in the same loop.

4.2.4. Compiler Optimizations

The complier performance for RISC processors is substantially better than those for DSP processors. The performance for our working reference code can be still improved by following some simple steps. By the judicious use of local variables to avoid register spilling, compiler performance can be improved considerably. By defining local variables within the scope of their usage, compilers can be aided in efficient register allocation. By converting critical code areas into functions gives the necessary head room for compilers, this is a trade-off between function call overhead and register spillage. Usage of pointers as intermediate address holders for multi-dimensional arrays and structure elements in iterative loops, avoids repeated address computation efforts. Restricting the number of arguments being passed can minimize function call overhead.

5. RESULTS

Optimization techniques suggested in Section 4 were applied to the GSM-AMR (NB) speech codec standard code. Table 4 lists the cycle counts achieved for various modes of operation of the GSM-AMR (NB) codec in our development. The metrics quoted assume zero latency memory access. The codec passed the mandatory bit exactness test as specified by the standard.

Modes	Codec		Encoder		Decoder	
	Baseline	Optimized	Baseline	Optimized	Baseline	Optimized
4.75	142.96	32.06	125.90	27.96	17.06	4.10
5.15	114.05	23.14	96.49	19.06	17.11	4.08
5.90	126.82	27.62	109.90	23.56	16.92	4.06
6.70	150.31	34.93	133.35	30.38	16.96	4.10
7.40	142.66	31.63	125.78	28.01	16.88	3.62
7.95	146.26	33.38	129.28	29.65	16.98	3.73
10.2	149.24	32.19	132.25	28.01	16.99	4.18
12.2	153.78	34.23	136.67	30.34	17.11	3.89

Table 4. Cycle count requirement for various modes of GSM-AMR (NB)

Using just the traditional RISC based optimization techniques [6] GSM-AMR (NB) code on ARM9T core takes about 62.5MHz (peak requirement), while at an additional cost of increased data memory and stack usage, it can be reduced to 52MHz.

6. CONCLUSIONS

In this paper, we presented methodologies for real time Vocoder implementation on new breed of DSP enhanced RISC architectures. The specific codec implemented was GSM-AMR (NB) on the ARM9E processor. By employing the architectural features of the DSP enhance RISC core along with some algorithmic improvements, it was possible to reduce the computational complexity by more than 77% over the baseline codec derived from the ETSI code. When compared to a fully optimized codec for ARM9T with similar code size and stack usage, around 33% reduction was possible. The techniques proposed are equally relevant to other speech codecs such as EVRS, G.723.1, MPEG-4 CELP and GSM-AMR (WB).

REFERENCES

1. “AMR Speech Codec – Transcoding Functions,” 3GPP TS 26.090, Version 4.0.0, Release 4, March 2001.
2. ARM Ltd., *ARM9E-S Technical Reference Manual*, ARM DDI 0240, May 2002.
3. Krishna Yarlagadda, “Arm Refocuses DSP Effort”, Microdesign Resourses, June 21, 1999, Microprocessor report.
4. Texas Instruments Incorporated, “OMAP processors for wireless devices”,SWPT005A, 2003
5. Michael Doell, Manfred Schlett, “A Cost-Effective RISC/DSP Microprocessor for Embedded Systems”, *IEEE Micro*, **Vol. 15**, Oct 1995.
6. Intel Corporation, *Intel XScale Core Developer’s Manual*, 273473-001, December 2000.

7. Ross Bannatyne, “The evolution of the digital signal controller”, http://www.eetasia.com/ART_8800226792_499481,499484.HTM.54535142
8. ARM Ltd, *ARM9TDMI Technical Reference Manual*, ARM DDI 0145A, November 1998.
9. Manish Arora, Suresh Babu P.V, Vinay M.K., “RISC Processor Base Speech Codec Implementation for Emerging Mobile Multimedia Messaging Solutions”, *DSP 2002*.
10. M.R. Schroeder and Atal B.S., “Code-Excited Linear Prediction: High Quality Speech at very low bit rates”, *ICASSP-85*.
11. Hedley Francis, “ARM DSP-Enhanced Extensions”, ARM White Paper, May 2001.
12. ARM Ltd., *ARM Architecture Reference Manual*, ARM DDI 0100E, June 2000.
13. “Writing Efficient C for ARM”, Application Note 34, ARM DAI 0034A, January 1998.
14. P. Frene, J. L. Hurel, “New 3G mobile applications”, *Alcatel Telecommunications Review*, 2nd Quarter 2002.