ASSEMBLY CODE OPTIMIZATION TECHNIQUES FOR REAL TIME DSP IMPLEMENTATION OF SPEECH CODECS

Manish Arora[1], Nitin Changdeo Lahane and Pradeep Srinivasan

Sasken Communication Technologies Limited
Survey #139/25, Amar Jyoti Layout, Ring Road, Domlur Post Office
Bangalore-560071, India. Phone +91-80-5578300.
nitin, pradeeps@sasken.com

ABSTRACT

A lot of effort has been spent over the last many years in the development of digital speech coding methods and their subsequent standardization. Algorithms have evolved which provide good quality speech at sub 8 kbps bit rates although at a much higher computational expense. DSP processors have also improved with time, have been well molded by algorithms, providing specific signal processing functionalities aiding in easier codec implementations along with lower power consumption at higher clock speeds. Software development tools and compilers have also improved although they do not work well in high volume, low cost systems. The performance of the firmware critically determines the cost and performance of the overall systems. This paper describes techniques and approaches commonly used to realize systems where hand assembly is necessary to obtain the needed performance. The specific codec implemented was ITU-T G.729 Annex B. The techniques described in this paper are applicable to any speech codec and DSP processor platform.

1. INTRODUCTION

The need for interoperability between different telecommunication networks has prompted a lot of standardization activity. Organizations such as the International Telecommunication Union (ITU), the European Telecommunication Standards Institute (ETSI) and the Telecommunication Industry Association (TIA) act as standard bodies, defining and publishing telecommunication standards. The standard bodies provide documentation describing the algorithms and references to the related technical publications, ANSI C code corresponding to a fixed-point fractional arithmetic implementation of the codec and a set of test vectors, which comprise input and output signals that can be used to verify code being developed.

Programmable DSP processing hardware has undergone tremendous growth in the last 15 years. Typical of the growth are the wireless DSP processors where the hardware has been well molded by applications. These enhanced conventional DSP processors are characterized with irregular data paths, small register files, specialized and non-orthogonal instruction sets. Consequently they make very poor compiler targets [1] and providing high-level language support is a difficult task. The need for hand assembly becomes very significant since the firmware determines the target applications performance. As an example the quality of the coreware in a mobile handset may determine its battery life significantly.

In this paper we describe techniques to obtain optimal performance while reducing development time and effort while working with such systems. The organization of the paper is as follows. In section 2, we describe common speech coding techniques. We describe typical DSP processor features in section 3. Section 4 suggests optimization techniques applicable to the reference code available from the standard body with respect to the optimal use of hardware features described in section 3. The results and conclusions are suggested in section 5.

---

[1] The author is presently working with Emuzed India Pvt. Limited, 839, 2 Cross, 7 Main, HAL II stage, Bangalore-560008, India. Email: arora@emuzed.com. Phone number: +91-80-5267038

2. SPEECH CODING TECHNIQUES

The field of speech coding has received a great amount of attention since the last many years. Algorithms have evolved to provide good quality speech at sub 8 kbps bit rates. Linear Prediction Coding with adaptive prediction and quantization allows the bit rate to be reduced to about 32 kbps while maintaining good speech quality. Further reductions are possible through the use of Code-Excited Linear Prediction (CELP). In CELP the predictor's excitation is derived from a codebook of stored vectors rather than from the LPC residual signal. The LPC residual is not transmitted and bit rates down to about 16 kbps can be easily achieved using CELP. Figure 1 shows the analysis-by-synthesis CELP configuration.

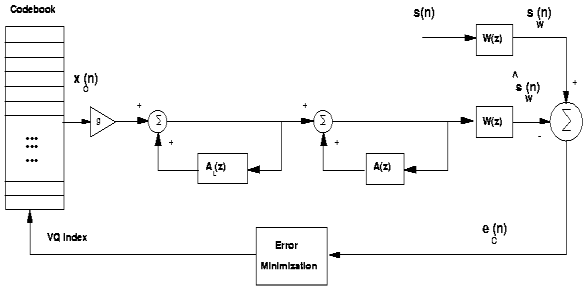

Figure 1. Analysis by Synthesis CELP Configuration.

One of the most popular techniques in current practical use is Algebraic Code-Excited Linear Prediction. Conventional CELP requires high computational effort for codebook search and a large memory for storing the codebook. ACELP does not require the storage of the codebook as it is generated on the fly. The encoder builds the codebook entries according to a set of rules governing the pulse positions. It selects the optimum vectors by the analysis-by-synthesis procedure and transmits the pulse positions and amplitudes to the decoder. The decoder can reconstruct the code vector with this information. Since each vector has a limited number of pulses efficient search procedure for the optimum pulse positions and amplitudes are well evolved.

The ACELP principle has been used in many speech-coding standards including GSM EFR [2] and G.729 [3]. The number of pulses and restrictions on pulse positions vary from standard to standard.

3. DSP PROCESSOR FEATURES

Programmable DSP processors have become a key component in many consumer, communication, medical and industrial products. They are cost-effective for low volume applications and have the advantages of potentially being reprogrammed in the field, allowing product fixes and upgrades. From the outset, algorithms have influenced DSP processor architectures. For every feature in a DSP processor there exists a DSP algorithm whose computation is optimized by addition of the feature.

DSP algorithms such as filtering, convolution are all multiplication intensive. Originally microprocessors implemented multiplications by a series of shift and add operations taking multiple cycles. Today all commercial DSP processors incorporate specialized hardware to compute one or more multiplication in a single cycle coupled most with a add operation to provide multiply-accumulate capability. Even if a single cycle multiplier is present there is a memory bottleneck to load the operands. To address the need for increased memory bandwidth DSP processors use two or more memory banks each with its independent address and data bus (Harvard Architecture). The program memory access is usually made efficient with the use of a program cache.

Instead of using the processor logic and arithmetic units for address generation there are hardware units available. Since the memory access patterns in DSP algorithms are predictable, sequential access and circular addressing, specialized addressing modes are implemented via dedicated hardware. The most common of these modes is Register Indirect addressing with Post Increment, which automatically increments the address pointer where repetitive computations are to be performed on memory aligned set of data.

DSP instruction sets try to make efficient use of the underlying hardware and at the same time reduce program memory size. This is because of the cost-sensitiveness of DSP applications. To

achieve these two goals the instruction set architecture compromises on orthogonality and hence compiler performance. Instructions allow several parallel operations in a single instruction such as multiple memory loads subsequent pointer updates. To reduce the instruction size minimal numbers of registers are used with instructions and instead mode control bits are used to toggle hardware features such as saturation, rounding. The overall result of this approach is complicated, irregular instruction sets and very poor high-level language performance.

Since DSP applications have very high computational requirements, several independent execution units capable of operating in parallel and pipelined in operation are provided. As an example there may be multiple arithmetic logic units and adjusting shifters present. The internal data path of the DSP may provide guard bits and be wider than needed to aid higher precision arithmetic and maintain numeric fidelity. Processors also provide extensive support for Zero-Overhead looping since DSP algorithms are looping intensive.

4. CODE OPTIMIZATION METHODS

Along with the documentation describing the algorithms and references, the standard bodies provides ANSI C code corresponding to a fixed-point fractional arithmetic implementation of the codec and set of test vectors for bit exact verification. The C code is written in a manner to aid porting to different platforms. In order to achieve this the basic mathematical operations of addition, subtraction, multiplication etc. are implemented as separate functions. This is done so that these operations can be rewritten in the most optimized manner for the specific DSP architecture. Also since the C language does not support fixed-point fractional arithmetic all the mathematical operations are simulated and functionalized.

Before the start of the porting, it is significant to have an idea of the routines that are the best candidates of optimization. A C code profiler can be used to profile both the encoder and decoder standard reference code. Profilers are usually bundled along with processor development tools. Functions critical from the profile results are possible choices for hand assembly implementation. The optimization methods can now be applied to the functions identified as candidates from the profile results.

4.1. Basic Operation Optimizations

Since the working reference code is based on a 16 bit fractional arithmetic in order to simulate the DSP hardware, there are lots of overheads in basic operation computations. For instance, a fractional multiply-accumulate operation may be implemented using a routine “L_mac” which further calls two other subroutines “L_mult” and “L_add”. A DSP takes 2-3 cycles to jump from a subroutine, 2-3 cycles to return from a subroutine. Hence the overall overhead in the call of the “L_mac” routines would be of the order of 12-18 cycles. This is very significant since most DSP hardware is capable of performing a single cycle MAC operation. Further more the MAC operation is intensively used inside loops, the total overhead encountered increases to a very high value for that subroutine. Weighted Million Operations Per Second (WMOPS) gives a good estimate of the looping overhead associated with specific routines as described in [4]. The subroutine calling overhead can be easily reduced by inlining the basic operation functions in the code. This should be particularly done for all loops since the cumulative looping overhead is very high in loops.

Typical DSP hardware are rich in arithmetic operation functionality. All the basic operations should be optimally implemented on the target platform. Saturation and rounding hardware should be used to implement the “Saturate” and “Round” routines. Some DSP’s provide the hardware to implement “Norm”, Normalization to 16 or 32 bit maximums. This leading edge bit detection hardware should be used to optimally implement “norm_s” and “norm_l” routines.

4.2. Usage of Multiple Memory Access

The Harvard memory architecture of DSP processors provides multiple memory access capability. The multiple memory access are commonly implemented with the use of separate memory banks or a multiport RAM or sometimes a RAM of a much faster access time so that

multiple sequential access are completed in a single cycle [1]. The C code should be studied for the opportunity to place data in different memory spaces so that parallel access can be performed. As an example the filter coefficient and input data can be placed in separate memory space. This can be accomplished with the use of compiler directives facilitated with the tools.

4.3. Optimum Use of Registers

DSP hardware typically provides data path registers, address generation registers and some general purpose registers. It is very important to manage registers while programming in assembly since memory access is costly. A very important optimization method is to dedicate registers for temporary variables instead of memory space. This method should be extensively used to reduce the memory access and improve speed.

The major reason for lack of compiler performance is the lack of general-purpose registers. The programmer can improve performance by writing smaller functions using less of temporary variables inside routines. This would improve compiler performance since the compiler would be able to dedicate registers to local variables. The C language "Register" keyword should be used to provide additional information to the compiler to dedicate important variables to registers.

4.4. Loop Optimizations

Speech codecs and other DSP algorithms have intensive looping. Processors commonly have hardware loops where the loop count modification, end loop condition check and jump to start of loop without delay are performed in hardware. Hardware loops should be used extensively instead of software loops. Post Increment register indirect addressing should be extensively used in loops to reduce the cycles used in address generation.

DSP processors implement a lot of pipelining to speed up their operation. Typical pipeline stages are Program Address generation-Fetch-Decode-Address Generation-Execute-Write to Memory. There may be more pipeline stages for the barrel shifter or multiple execute stages. Because of this pipelined operation there are a lot of pipeline latencies. Either the compiler inserts a "nop" operation or the DSP hardware generates wait states during the execution of the program leading to use of extra cycles. These latencies should be carefully examined and removed. A common method to do this is Instruction Shuffling, where a third instruction of an independent operation is inserted between two instructions having pipeline conflicts. This method is sometimes not applicable in small loops due to lack of independent computation operations. The loop can be then unrolled and instruction shuffling done with the instructions of the originally next loop run.

Consider the pipeline operation when a jump statement is encountered. At the decode stage when a jump is found the next two instructions are in the Program Address Generation and Fetch stages. At this time the pipeline is flushed and the pipeline restarts from the target jump position. The result is a two-cycle latency associated with the jump. Nothing much can be done for conditional jumps. But unconditional jumps can be replaced with delayed branches wherein the hardware finishes two instructions after the branch instruction and then makes the jump. The programmer can modify the code logic and use delayed branches/subroutine calls to save calling overhead.

Some DSP processors provide zero overhead conditional execution of instructions via flag settings at run time. This method can be used instead of conditional branch instructions. Since the programmer has explicit control over the memory map, parallel load-store hardware can be used with aligned data.

5. RESULTS AND CONCLUSIONS

The optimizations suggested above were implemented on the ITU-T G.729 Annex B speech codec reference code. Consider the routine "Residu" from the standard reference code in figure 2. The routine computes the LPC residual by filtering the Signal through A(Z). The loop count "lg" is typically of subframe length (40). The routine consumes approximately 4-7% of the time in the C code for different inputs and is an optimum candidate for optimization. The DSP used in this experiment was an enhanced conventional DSP similar to TMS320C54x. In

the first iteration the code is simply ported to assembly. The cycles used are 10916 per hit of the function. In the second iteration of optimization the prediction coefficient was assigned to a different memory bank. Next all the arithmetic operations were inlined in the code and the calling overhead was removed. The optimization process ended with the use of hardware loops instead of software loops and optimization of the address generation in the inner loop with the use of post decrement addressing.

```
 1: /*-------------------------------------------------------------------------*
 2:  * Procedure Residu:                                                       *
 3:  *                                                                         *
 4:  * Compute the LPC residual  by filtering the input speech through A(z)  *
 5:  *-------------------------------------------------------------------------*/
 6:
 7: void Residu(
 8:   Word16 a[],    /* (i) Q12 : prediction coefficients                     */
 9:   Word16 x[],    /* (i)     : speech (values x[-m..-1] are needed         */
10:   Word16 y[],    /* (o)     : residual signal                             */
11:   Word16 lg      /* (i)     : size of filtering                           */
12: )
13: {
14:   Word16 i, j;
15:   Word32 s;
16:
17:   for (i = 0; i < lg; i++)
18:   {
19:     s = L_mult(x[i], a[0]);
20:     for (j = 1; j <= M; j++)
21:       s = L_mac(s, a[j], x[i-j]);
22:
23:     s = L_shl(s, 3);
24:     y[i] = round(s);
25:   }
26:   return;
27: }
```

Figure 2. *Standard Reference Code for "Residu" Routine.*

Table 1 shows the cycle count the routine after different iterations of coding.

Iteration	Residu Cycle Count	Optimization Performed
1	10916	Original Code Ported To Assembly
2	10016	Prediction Coefficients assigned Independent Memory Access
3	6816	Inlined L_mult, L_mac, round operations
4	1608	Hardware Loop Used
5	688	Optimized x[i-j] Address Generation via Post Decrement Addressing

Table 1. "*Residu" Cycle Count After Various Code Optimizations*.

The overall cycle count required for each run of the routine has reduced from 10916 to 688 cycles after the application of the techniques described in section 4.

This paper has described the techniques involved in optimizing speech-coding standards in real time DSP systems. The approach has been general in nature and can be well applied to any DSP processor or development environment.
The key points of suggested in our work can be summarized as:

- Basic operation inlining to reduce calling overhead.
- Optimum use of available hardware to implement basic arithmetic operations saturation, rounding and norm etc.
- Allocation of variables to independent memory access.
- Optimum use of registers and selective functionalizations.
- Pipeline latency reduction via instruction shuffling.
- Usage of hardware loops, loop unrolling and delayed branch/ function calls/jumps.

6. REFERENCES

[1] Phil Lapsley, Jeff Bier, Amit Shoham and Edward A. Lee, *DSP Processor Fundamentals: Architectures and Features*, *IEEE Press*, 1996.

[2] GSM 06.60 (EN 301 245), "Digital Cellular Telecommunication System: Enhanced Full Rate (EFR) Speech Transcoding".

[3] ITU-T Recommendation G.729, "Coding of Speech at 8kbit/s using Conjugate Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)".

[4] David H. Crawford and Emmanuel Roy, "Techniques for Real-Time DSP Implementation of Speech Coding Algorithms ", Proc. *DSP World, ICSPAT*, pp 2-7, 1-4 November 1999.