Optimizing G.722.2 Speech Codec using MIPS® DSP Application Specific Extensions

This paper describes 3GPP AMR-WB speech encoder optimization using the MIPS32® DSP ASE (Application Specific Extension). After initial analysis of the code, a number of code optimization techniques are discussed, including implementing intrinsics, re-writing DPF operations, using SIMD instructions to perform DSP operations in parallel, efficient loading of 16-bit data, loop unrolling with instruction reordering, loop merging, loop splitting, and split summation. These optimizations lead to an overall performance gain of 5.8x for the encoder mode 6 (19.85 kbps), reducing peak processor load from 572 MCPS to 99 MCPS.

Document Number: MD00551
Revision 1.02
August 22, 2011

Contents

1 Preview

This application note describes the process of optimizing the C source code of a 3GPP Adaptive Multirate Wide Band (AMR-WB) speech encoder using the MIPS32® DSP ASE (Application Specific extension) while ensuring the bit-exactness of the modified code with the original C source code provided by 3GPP. The application note begins with a brief overview of speech coding and continues with specific details about AMR-WB speech codec, optimization of AMR-WB speech codec specifically for the MIPS32® 24KE™ core, and finally a conclusion.

2 Overview

With the rapid evolution of digital communication systems like wireless systems, VoIP, and video conferencing, the bandwidth requirements are more stringent as the bandwidth needs to be shared between numbers of users. Speech compression reduces the data redundancy by transmitting only the required parameters and thus eases bandwidth requirements. Algorithms that compress the digitized speech signals are computationally intensive as they involve extensive math operations like multiplication and accumulation, shifting, extended precision arithmetic. Traditionally, these algorithms, which are basically composed of an encoder (to compress the speech) and a decoder (to synthesize the compressed speech), have been implemented on DSP processors. But as general-purpose embedded processors get faster and smarter, they are ready to challenge the DSP processors by efficiently performing the math operations that are core of DSP applications. Hence, it has become possible to migrate speech processing to the main processor, enabling SoC (System on Chip) solutions to eliminate the DSP processor entirely in wireless solutions (cell phones) or in VoIP solutions. This lowers the die size of the chip and power consumption hence ultimately overall cost of the product, which is a critical factor for consumer electronics manufacturers.

However, the speech codecs most widely used today to transmit the speech over wireless and IP data networks do not reproduce speech faithfully for a variety of dialects as they are based on telephone bandwidth nominally limited to about 200-3400 Hz and sampled at a rate of 8 kHz. The complete frequency spectrum of the human voice cannot be captured with the above sampling rate, which inhibits the full analysis on the encoder side and the faithful reproduction of speech in the decoder. The AMR-WB is a better choice as it samples the signal at 16 KHz and scans bandwidth of 50-7000 Hz, resulting in a more natural sounding reproduced speech with increased intelligibility.

The AMR-WB speech codec was adopted by both 3GPP and ITU-T(G.722.2) standards bodies. This eliminates the need for transcoding and facilitates the implementation of wideband voice applications and services across a broader communication systems and platforms.

Wideband speech coding can be found in 3GPP communication systems, high fidelity telephony over broadband packet networks and ISDN, audio and video conferencing, internet applications, and digital radio broadcasting.

Considering the importance of AMR-WB for wideband applications in both wired and wireless communications, we at MIPS Technologies went forward with benchmarking the AMR-WB codec on a MIPS core. The DSP extensions of the MIPS32® 24KE™ core family are ideally suited to accelerating the performance of wireless and VoIP applications such as 3GPP AMR-WB, facilitating the migration of speech processing to the main processor. The reference C source code for 3GPP AMR-WB is not optimized to a specific target, and therefore a basic compilation of the code does not take full advantage of the DSP extension or the compiler schedules for the 24KE core. We will identify two levels of code optimization techniques that result in more than 5.8x overall performance gain for AMR-WB encoding, reducing the average 24KE™ processor load to approximately 99MCPS (million cycles per second), compared to 572 MCPS for an un-optimized encoder.

3 Technical Overview of AMR-WB

The AMR-WB speech codec is based on the ACELP (Algebraic Code Excitation Linear Prediction) algorithm which is also employed in the AMR-NB (Adaptive Multirate Narrow Band) and EFR (Enchance Full Rate) speech codecs as well as ITU-T G.729 and G.723.1 at 5.3 kbit/s. The AMR-WB speech codec supports nine modes with bit rates of 23.85, 23.05, 19.85, 18.25, 15.85, 14.25, 12.65, 8.85 and 6.6 kbit/s. The codec also includes a background noise mode designed to be used in the discontinuous transmission (DTX) operation of GSM and as a low bit rate source dependent mode for coding background noise in other systems. In GSM the bit rate of this mode is 1.75 kbit/s. The AMR-WB codec is capable of switching its bit-rate every 20 ms frame command from the scheduler (for ex: GSM/UMTS).

The 12.65 kbit/s mode and the modes above it offer high quality wideband speech. The two lowest modes at 8.85 and 6.6 kbit/s are intended to be used only temporarily during severe radio channel conditions or during network congestion.

The AMR-WB codec operates at a 16 kHz sampling rate. Coding is performed in blocks of 20 ms. Two frequency bands, 50-6400 Hz and 6400-7000 Hz are coded separately for decreasing complexity and focusing the bit allocation into the subjectively most important frequency range. Note that already the lower frequency band goes far above narrowband telephony.The lower frequency band is coded using an ACELP algorithm. Several features have been added to obtain a high subjective quality at low bit-rates on wideband signals. Linear prediction (LP) analysis is performed once per 20 ms frame. Fixed and adaptive excitation codebooks are searched every 5 ms for optimal codec parameter values. The processing is carried out at a 12.8 kHz sampling rate.

The higher frequency band is reconstructed in the decoder using the parameters of the lower band and a random excitation. The gain of the higher band is adjusted relative to the lower band based on voicing information. The spectrum of the higher band is reconstructed by using an LP filter generated from the lower band LP filter.

AMR-WB encoder and decoder algorithms are briefly described below, for more details please refer to the 3GPP specifications (series 26) for AMR-WB speech coding.

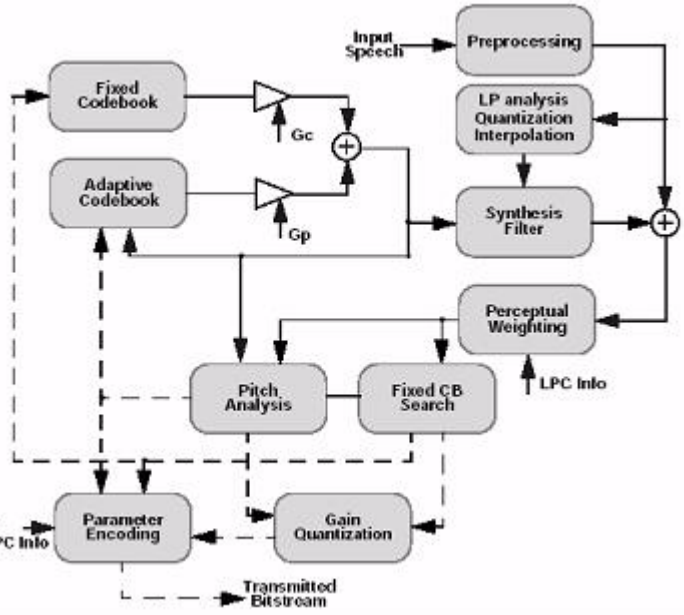
3.1 AMR-WB Encoder

The AMR-WB encoder is based on a algebraic code-excited linear-prediction model. It uses analysis by synthesis technique wherein the encoder has a local decoder and the decoded signal is compared with the original signal. The filter parameters are then selected to minimize the mean-square weighted error between the original and reconstructed signal. The encoder operation is depicted in Figure 1.

After decimation, high-pass and pre-emphasis filtering is performed. A sixteenth order LP analysis is performed once per frame. The set of LP parameters is converted to intermittent spectral pairs (ISP) and vector quantized using split-multistage vector quantization (S-MSVQ). The quantized and unquantized LP parameters or their interpolated versions are used depending on the subframe. The speech frame is divided into 4 subframes of 5 ms each (64 samples at 12.8-kHz sampling rate) to optimize tracking of the pitch and gain parameters and reduce the complexity of the codebook searches. The excitation in each subframe is represented by both an adaptive-codebook contribution, which simulates the pitch structure of the voiced sounds, and a fixed codebook contribution, which simulates unvoiced sounds. The adaptive and fixed codebook parameters are transmitted every subframe. An open-loop pitch lag is estimated in every other subframe or once per frame based on the perceptually weighted speech signal.

Figure 1 Encoding Principle

The adaptive codebook component represents the periodicity in the excitation signal using a fractional pitch lag with 1/4th or 1/2nd sample resolution (depending on the mode and sample resolution). The adaptive codebook is searched using a two-step procedure. An open-loop pitch lag is estimated per frame based on a perceptually weighted speech signal. The adaptive codebook index and gain are found by a closed-loop search around the open-loop pitch lag. The signal to be matched, referred to as the target signal, is computed by filtering the LP residual through the weighted synthesis filter. The target signal is updated by removing the adaptive codebook contribution, and this new target is used in the fixed codebook search, the fixed codebook is an algebraic codebook. The gains of the adaptive and fixed codebooks are vector-quantized with 6 or 7 bits with moving average prediction applied to the fixed codebook gain.



In each 20 ms speech frame 132, 177, 253, 285, 317, 365, 397, 461, 477 bits are produced, corresponding to a bit rate of 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05, 23.85 kbit/s.

3.2 AMR-WB Decoder

In the decoder, the transmitted indices are extracted from the received bitstream. The indices are decoded to obtain the coder parameters at each transmission frame. These parameters are the ISP vector, the 4 fractional pitch lags, the 4 LTP filtering parameters, the 4 innovative code vectors, and the 4 sets of vector quantized pitch and innovative gains. For 23.85-kbit/s bit-rate, high-band gain index is also decoded. The ISP vector is converted to the LP filter coefficients and interpolated to obtain LP filters at each subframe. Then, at each 64-sample subframe:

– the excitation is constructed by adding the adaptive and innovative code vectors scaled by their respective gains;

– the 12.8-kHz speech is reconstructed by filtering the excitation through the LP synthesis filter;

– the reconstructed speech is de-emphasized.

Finally, the reconstructed speech is upsampled to 16 kHz, and high-band speech signal is added to the frequency band from 6 kHz to 7 kHz.

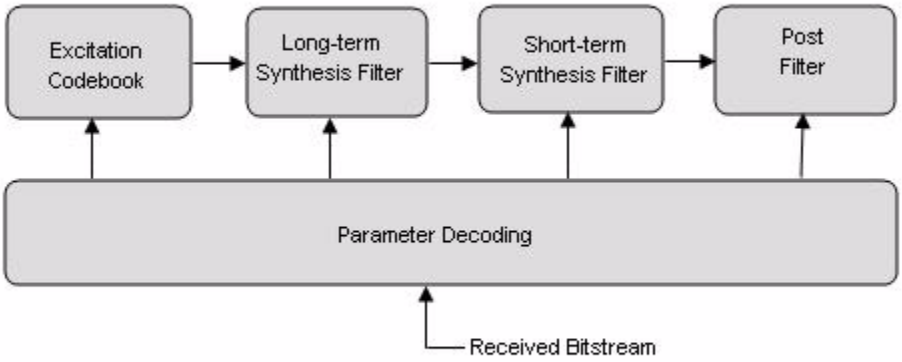


Figure 2 AMR-WB Decoder Speech Synthesis Procedure

The rest of this paper is organized as follows. In Section 4 we present details about software tool chain and a brief analysis of the run-time characteristics of the AMR-WB encoder as motivation for the optimization techniques that follow. In Section 5 we identify first-order optimization techniques and discuss their impact, and in Section 6 we identify second-order optimization techniques and discuss their impact. Finally, in Section 7 we summarize our optimization approach. Section 8 provides a brief explanation of our method for measuring the performance of AMR-WB in terms of its load on the processor.

4 Optimization Process For a MIPS32® 24KE™ Core

This section elaborates the optimization process for 3GPP AMR-WB reference code on the MIPS32 24KE™ core. We classify the optimization into two main levels, first order and second order optimizations. Each level in turn is composed of sub-levels to clearly describe the complete steps followed.

The first order optimizations are compiler-specific changes to fully exploit the DSP instructions, core pipeline characteristics, and implementing basic arithmetic operations using MIPS instructions.

Second order optimizations are more code specific changes, computationally intensive functions are identified and various optimizations techniques are applied to improve speed. The optimization techniques include using SIMD instructions, loop unrolling, alternate memory access instructions to access 16-bit data, instruction interleaving to reduce pipeline stalls, manual function inlining, loop merging, loop splitting, and split summation.

We observe the basic 80-20 rule of thumb, where 80% of the functions consume 20% of the time and the remaining 20% of the functions consume 80% of the time. Among these 20% functions we identify the most critical functions and apply the second order optimizations. The programmer needs to have an in depth knowledge of the algorithm to apply second order optimizations efficiently, he should be able to identify the time-consuming functions which the compiler does not optimize optimally.

The emphasis here is mainly on speed improvement, due to time constraints we have not investigated much on the code size optimizations. We provide approximate estimate of the code memory and data memory requirements before and after all the optimizations are implemented.

As our main intention is to demonstrate the capability of MIPS32® 24KE core to speech processing, we have limited ourselves to C-based optimization. Using only C optimizations we are able to show that a complex speech coder like AMR-WB can run 5.8x faster compared to the reference code on MIPS32® 24KE core.

4.1 Initial Compilation with gcc

The original AMR-WB C-reference code from 3GPP was compiled using gcc and tested against the reference test vectors provided by 3GPP for all possible modes/bit-rates of AMR-WB to ensure bit-exactness. For more details about the C-reference code and test vectors please refer to www.3gpp.org (series 26).

4.2 Development Tool Chain

The development tool chain is composed of a standard gcc compiler, assembler for MIPS core, MIPSsim simulator and debugger. Make files were written with all the required build options to generate the executable for native platform and MIPS platform. We performed all tests using the Linux operating system.

The reference test vectors provided by 3GPP were used for testing the optimized code. As mentioned earlier the optimized code produced bit-exact results when compared with the reference test vectors.

4.3 Functional Profiling

Performance analysis was done to identify the most critical functions that could be the potential targets for optimization. We did functional profiling with -03 optimization level enabled in the compiler. Later, we will show how the various optimizations affect these critical functions, as well as overall performance. A flat function-based profile of a default compilation of the 3GPP reference C code encoding the first 25 frames of 3GPP test stream `T00.inp for mode 6 (19.85 kbps)` is presented in Table 1. Only the top 20 functions are shown for simplicity. The profile is flat because the cycle counts for each function exclude those spent in called functions.

Table 1 AMR-WB Encoder Function Profile[a]

Function	Percentage of time	Avg. Cycles / Frame	Avg. Calls / Frame	Avg. Cycles / Call
L_mac	38.79	4444744	233782	19
ACELP_4to64_fx	14.84	1700242	4	425060
L_shl	5.17	592966	13430	44
mult	4.60	527465	23489	22
L_msu	4.30	492152	27273	18
add	3.69	422924	32528	13
Convolve	2.47	282794	12	23566
Norm_corr	2.14	210056	4	52514
Residu	2.01	212496	20	10624
sub	1.76	181940	16540	11
L_mult	1.74	127545	15943	8
Syn_filt	1.40	136272	12	11356
Pitch_med_ol	1.28	138381	2	69190
L_shr	1.08	123361	7119	17

Table 1 AMR-WB Encoder Function Profile[a] (Continued)

Function	Percentage of time	Avg. Cycles / Frame	Avg. Calls / Frame	Avg. Cycles / Call
round	1.01	116070	12136	10
cor_h_x	0.94	108166	4	27041
Sub_VQ	0.94	107995	20	5399
VQ_stage1	0.88	100679	2	50339
L_Extract	0.86	99075	2818	35
coder	0.85	97341	1	97341

a. Test Conditions: Run under cycle-driven MIPSsim™ Simulator 4.07.00, configured with a 24KEc core, 64KB 4-way associative I and D caches, zero wait-state memory.

Apart from the code book search which is second in the list, most of the top functions are all basic DSP arithmetic operations and the dominant function is `L_mac` at 38.79%, which performs a multiply-and-accumulate operation. All of the DSP arithmetic operations in the table (`L_mac`, `L_msu`, `L_shl`, `sub`, `mult`, `L_mult`, `L_shr`, `Mpy_32_16`, and `L_Extract`) are called many times per frame but take relatively few cycles per call. This makes them ideal candidates for inlining, which avoids the overhead of a function call. They can also be optimized to use the DSP ASE, as discussed in Section 5.3, "Implementing Basic Operations using MIPS DSP ASE Instructions".

Of course, when these functions are inlined, the cycles they consume will be accounted for in the main 3GPP AMR-WB routines, which will significantly change the profile. These routines, such as `ACELP_4to64_fx`, `Convolve`, `Norm_corr`,`Residu`, `Cor_h_x`, Syn_filt and `Pitch_med_ol` in the table above, will then become the dominant functions. We can look at the code of these dominant routines to see which of them are suitable for optimization. On this basis, we chose some routines for optimization: `Convolve`, `Autocorr`, `Residu`, `Syn_filt`, `Pitch_med_ol`, `Cor_h_x`, `Pred_lt4`, `az_isp`, `isp_az`. Due to time constraints we did not optimize ACELP codebook search routine. Based on performance analysis we found that after first and second order optimizations the codebook routine accounts for 20% of the overall encoder cycle count. So this can be a potential candidate for further optimization.

5 First-Order Optimizations

This section explains the optimization techniques that require relatively little effort, yet have a large impact on the overall efficiency of the code. These include:

1. Invoking appropriate compiler options,
2. function inlining, and
3. implementing basic arithmetic operations using MIPS32® DSP ASE instructions.

5.1 Getting The Most Out Of The Compiler

It is critical to use the most up-to-date version of the `sde-gcc` compiler. All the optimizations discussed in this paper were tested with version 6.05.00 of `sde-gcc`. The following compiler options impact the efficiency of compiled code, and should be used:

```
-O3              (optimization level 3)
-funroll-loops   (unroll loops)
-mtune=24ke      (tune for 24KE pipeline)
-mips32r2        (MIPS32 revision 2 architecture)
-mdsp            (DSP ASE)
```

Note: For the CodeSourcery SG++ compiler for MIPS (Spring 2008 release and newer), the `mtune` switch would be: `-mtune=24kec`.

It is worth bearing in mind that compiling all C sources with a single invocation of `sde-gcc` can improve performance, because it enables inter-module optimizations. We did not enable inter-module optimization. Because of the loop unrolling feature in -03 optimizations, there will be a definite increase in the code size, but as our main focus of performance improvement was speed, we used -03 optimization for all our tests.

Initial performance after compiling with the above mentioned compiler options is as below, the memory figures include sizes for both the encoder and decoder.

Table 2 Initial Performance of AMR-WB Codec

Speed	Program Memory	Tables/Constants	Static Data
572 MCPS (Encoder)	100 kbytes	26.2 kbytes	~3.8 kbytes

5.2 Function Inlining

Inlining is a process of replacing a function call by the body of the function. Inlining basically eliminates the function call overhead, thereby improving speed at the expense of an increase in code size. As the code size/memory is also an important factor in the optimization process, there must be a fine balance of the two.

Inlining can be done automatically or manually. In the case of automatic inlining (the optimization level must be set to -04), the compiler chooses the functions to be inlined. This can speed up some programs but can also increase code size and in some cases reduce I-cache efficiency. Instead, we used gcc's ***inline static*** keyword on critical functions to be compiled inline, thus maintaining the fine balance mentioned above.

Later in the paper we will discuss manual inlining, wherein we inlined the functions inside some big functions and did some instruction re-ordering to further improve the performance.

The Speech coder performance details after using ***inline static*** keyword on critical functions that needed to be inlined are shown in Table 5.1

5.3 Implementing Basic Operations using MIPS DSP ASE Instructions

The reference C-code uses a set of functions, declared in `basic_ops.h`, that perform many DSP arithmetic operations ranging from addition to multiply and accumulate operations. These small functions are used throughout the code, so optimizing them provides an easy way to immediately improve codec performance across the board. The functions emulate DSP operations in C, which typically translate to functionally-correct but inefficient instruction sequences that lead to significant cycle growth. These functions were rewritten with DSP ASE instructions to dramatically improve performance. In addition, the reference code does not explicitly inline these functions, so forcing function inlining can reduce the overhead of function calls. We also inline the functions in `oper_32b.h`, which automatically take advantage of the DSP ASE since they are composed of the basic operations already optimized. In

addition the Word16, Word32, and Flag data types used in AMR-WB code were redefined in typedef.h to comply with MIPS32® architecture.

As mentioned earlier, specific DSP ASE instructions can be invoked at the C level by use of compiler intrinsics. Intrinsics, as discussed in more detail in the *MIPS® SDE 6.x Programmer's Guide* (MD00428), have the appearance of function calls. For example, note the instruction-intrinsic pair below:

```
instruction:  SHLL_S.PH rd, rt, sa
intrinsic:    v2q15 __builtin_mips_shll_s_ph( v2q15, i32 )
```

For the CodeSourcery SG++ compiler for MIPS (Spring 2008 release), please refer to *MIPS® SDE Library* (MD00623).

The compiler has information about expected instruction latencies in the 24KE pipeline and can therefore schedule the DSP instructions efficiently. The same information is not available to the assembler for asm statements in C, which assumes one-cycle latencies. Therefore, the use of inline `asm` operands can lead to pipeline stalls and poor execution performance. For this reason, we strongly recommend that compiler intrinsics be used to invoke DSP instructions at the C level.

As an example, we show below how the `L_mult()` operation (fractional multiplication) was re-written to take advantage of an equivalent DSP ASE instruction:

5.3.1 Original 3GPP Implementation for L_mult:

```
Word32 L_mult (Word16 var1, Word16 var2)
{
    Word32 L_var_out;
    L_var_out = (Word32) var1 *(Word32) var2;
    if (L_var_out != (Word32) 0x40000000L)
    {
        L_var_out *= 2;
    }
    else
    {
        Overflow = 1;
        L_var_out = MAX_32;
    }
    return (L_var_out);
}
```

5.3.2 DSP ASE-Optimized Implementation for L_mult:

```
Word32 L_mult(Word16 var1, Word16 var2)
{
   Word32 L_var_out;

   L_var_out = (uint32_t)__builtin_mips_muleq_s_w_phr((v2q15)((uint32_t)var1),
               (v2q15)((uint32_t)var2));
   return(L_var_out);
}
```

Other basic operations were optimized in a similar manner.

5.4 Performance Impact of First-Order Optimizations

The performance impact of the various optimizations discussed above is shown in Table 3. Note that all speedups are relative to an `sde-gcc` compilation of the reference 3GPP C code. The compiler techniques discussed in Section 5.1, "Getting The Most Out Of The Compiler" were used throughout, even for the reference compilation. For a brief discussion of how to measure performance in terms of MCPS, see Section 8.

Table 3 Performance Impact of First-Order Optimizations on AMR-WB Encoder Processor Load

Parameter	(Reference code) -03 Optimization	Inline Functions in basic_ops.h and oper_32b.h.	Implementing Basic Operations with DSP ASE Instructions
Speed	572 MCPS	356 MCPS	130.82 MCPS
Speedup	-	1.61x	4.37x

Simply by inlining the functions in `basic_ops.h` and `oper_32b.h`, we see a 1.61x speed up on average. By implementing the functions in `basic_ops.h` to efficiently utilize DSP ASE instructions, overall performance improved by a factor of 4.37x, decreasing the average processor load per frame to 130.82 MCPS, from a whopping 572 MCPS.

6 Second-Order Optimizations

The optimization techniques discussed in the previous section are generic and can be applied to optimization of code on any processor, provided that the intrinsic operations noted above are used. Second-order optimization techniques, as described below, are processor-dependant optimization techniques in which the code is modified to fully exploit features of the processor.

The following optimizations were performed:

(1) Optimize math operations on 32-bit DPF (Double Precision Format) data

(2) Use Single-Instruction, Multiple-Data (SIMD) instructions for 16-bit data operations

(3) Use LW and LWL/LWR instructions for accessing 16-bit data

(4) Use loop unrolling with instruction reordering to avoid pipeline stalls

(5) Split Summation

(6) Loop Merging and Loop Splitting

6.1 Optimizing Math Operations on 32-bit DPF Data

Most of the standard speech coders/vocoders use a non-standard representation of 32-bit double precision numbers, known as double precision format (DPF), defined in the following equation:

L_32 = hi_val<<16 + lo_val<<1

where *L_32* is a 32-bit signed integer, and *hi_val* and *lo_val* are 16-bit signed integers.

The DPF format and operations based on it are in `oper_32b.c` file. The operations include `Mpy_32( )`, `Mpy_32_16()` and `Div_32()`. The DPF format was designed for 16-bit processors that could not handle 32-bit operations. Although MIPS processors are 32-bit, in order to maintain bit-exactness with 3GPP code, the operations had to be implemented in DPF format. The operations were optimized by combining two 16-bit numbers into one 32-bit number and then performing multiplication.

As an example we show how an optimized version of `Mpy_32_16` was re-written using the MIPS instruction set.

6.1.1 Original 3GPP Implementation for Mpy_32_16:

In the original 3GPP implementation, the two 16-bit DPF values had to be extracted from 32-bit numbers before proceeding with the multiplication.

```
void L_Extract (Word32 L_32, Word16 *hi, Word16 *lo)
{
    *hi = extract_h (L_32);
    *lo = extract_l (L_msu (L_shr (L_32, 1), *hi, 16384));
    return (L_32);
}
```

Then `Mpy_32_16` was required:

```
Word32 Mpy_32_16 (Word16 hi, Word16 lo, Word16 n)
{
     Word32 L_32;

     L_32 = L_mult (hi, n);
     L_32 = L_mac (L_32, mult (lo, n), 1);
     return (L_32);
}
```

6.1.2 MIPS Implementation for Mpy_32_16:

As 24KE core is 32-bit, there is no need to extract the 16-bit values, so calls to `L_Extract` function are no longer required—only `Mpy_32_16` (multiplication of 32-bit by 16-bit) had to be implemented.

```
Word32 Mpy_32_16(Word32 L_32,Word16 x)
{
     long long r64;

    r64 = __builtin_mips_dpaq_sa_l_w(r64, (q31)t0, (q31)x);
    t0 = __builtin_mips_extr_w(r64, 16 );
    t0 = t0 & 0xFFFFFFFE;;
}
```

With this implementation, the call to the `L_Extract()` function was completely eliminated in functions like `Chebps`, `Get_lsp_pol`, `Get_lsp_poly16Khz`, `isp_az`, and the original `Mpy_32_16()` was replaced by an optimized `Mpy_32_16()` routine. This saved more than 35000 cycles/1.75 MCPS per frame.

6.2 Using SIMD Instructions

Digitized speech data can be represented using 16-bit (half-word) samples, so much of the 3GPP AMR-WB code is based on 16-bit data operations. In some cases these operations can be performed in parallel. We can exploit this data-level parallelism by using paired-half-word SIMD instructions to perform two 16-bit operations at the same time.

Care must be exercised when using SIMD instructions, however, because naïve implementations can sometimes be less efficient than their single-data-operation counterparts. Although paired-half-word SIMD instructions replace two instructions with one, multiple data must be loaded and stored, and inefficient scheduling can result in pipeline stalls and extra cycles. SIMD operations are often most efficient when used in conjunction with two other techniques: parallel data loads and loop unrolling, as discussed below.

6.3 Efficient Loading of 16-Bit Data

To complement paired-half-word SIMD operations, when two 16-bit half-words are adjacent in memory and aligned (that is, both half-words are in the same addressable word), the LW instruction can be used to load both at the same time into a GPR. Subsequently, the data can be separated and sign-extended into two GPRs or, ideally, it can be used immediately by a paired-half-word SIMD instruction.

When two 16-bit half-words cross a word-addressable boundary in memory the LWL/LWR instruction pair may be used to load both half-words into a single GPR. While not as efficient as a LW instruction, this is the most efficient way to load unaligned 16-bit data into a paired-half-word GPR for a SIMD operation.

6.4 Using Loop Unrolling and Instruction Reordering

Loop unrolling is a common method for improving performance at the expense of code size. At the C level, loop unrolling improves performance by reducing the overhead due to each loop iteration. At both the assembly and C levels, loop unrolling provides more opportunity for instruction reordering to reduce hardware pipeline stalls. For DSP ASE-optimized code in particular, loop unrolling with instruction reordering allows the two techniques already discussed, SIMD instructions and efficient 16-bit data loads, to be used more effectively.

6.5 Second-Order Optimization Example: Residu()

The function `Residu()`, which computes residual by filtering the input speech through A(z), was re-written using all of the second-order optimizations discussed in 6.2, 6.3 and 6.4. We will illustrate with a specific loop from the routine. Similar optimizations are used in other functions.

The code below is a fragment of the code for residual calculation, where `lg` is the size of filtering (64/320) and m is the lpc order, which is 16 here.

```
for (i = 0; i < lg; i++)
{
   s = L_mult(x[i], a[0]);
   for (j = 1; j <= m; j++)
      s = L_mac(s, a[j], x[i - j]);
   s = L_shl(s, 3 + 1);
   y[i] = round(s);

}
```

For the case lg=64, the loop above performs a total of 1024 multiply-accumulate (MAC) operations using the `L_mac()` basic operation.This operation was inlined and optimized to use DSP ASE instructions as part of our first-order optimizations, but we will show how the entire loop can be further optimized with second-order optimizations.

In the optimized implementation, all the techniques mentioned so far can be seen.Two MACs in parallel are implemented with a single `DPAQ_S.W.PH` instruction (dot product with accumulate on vector fractional half-word elements). Data for this instruction is loaded using two `LWL-LWR` instruction pairs. A single `EXTR_RS_H` instruction (extract a value with right shift, rounding, and saturation from an accumulator to a GPR) is needed at the end of the loop to extract the sum from the accumulator. While this is an obvious strategy to improve the performance of the inner loop, it turns out to be inefficient. The overhead of the loop combined with pipeline stalls that cannot be filled mask the benefit of the SIMD operations, resulting in a negative impact on performance gain.

The solution to this problem is to fully unroll the inner loop to eliminate the overhead and allow for more instruction reordering to eliminate pipeline stalls. As before, every two MACs are performed with a single `DPAQ_S.W.PH` instruction, with a single `EXTR_RS_H` instruction to obtain the final sum. In the code, the inner loop is completely unrolled. The coefficients required for multiplication and accumulation `a[]` are loaded at the beginning and completely outside the loop. The software piplelining can also be seen in the code where loading of `x[]` for the first computation is done outside the loop and `x[]` is loaded again at the end of the loop for the next iteration. This function is shown below.

```
void Residu(Word16 a[],Word16 m, Word16 x[], Word16 y[], Word16 lg )
{

    int i;
     a64 acc;
     v2q15_union xx, xx0, xx1;
    v2q15_union aa0, aa1, aa2, aa3, aa4, aa5, aa6, aa7, aa8;

/* Make use of aligned/misaligned location of a.When a is aligned lwl/lwr pair is
replaced with lw (LD32 vs. LD32_A macros)*/

    if (((int)a)%4 == 0)
    {
        aa0.c = LD32_A(&a[0]);
        aa1.c = LD32_A(&a[2]);
        aa2.c = LD32_A(&a[4]);
        aa3.c = LD32_A(&a[6]);
        aa4.c = LD32_A(&a[8]);
        aa5.c = LD32_A(&a[10]);
        aa6.c = LD32_A(&a[12]);
        aa7.c = LD32_A(&a[14]);
    }
    else
    {
        aa0.c = LD32(&a[0]);
        aa1.c = LD32(&a[2]);
        aa2.c = LD32(&a[4]);
        aa3.c = LD32(&a[6]);
        aa4.c = LD32(&a[8]);
        aa5.c = LD32(&a[10]);
        aa6.c = LD32(&a[12]);
        aa7.c = LD32(&a[14]);
    }
    aa8.c = (uint16_t)a[m];
```

```
    /* Moved out of the loop following 3 lines (software pipelining):
       to remove stalls appearing at the end of the loop */
     xx0.c = LD32(&x[0-1]);
     xx0.a = PACKRL_PH(xx0.a, xx0.a);
     xx1.c = LD32(&x[0-3]);

     for (i = 0; i < lg; i++)
     {

         acc = 0;

         xx1.a = PACKRL_PH(xx1.a, xx1.a);

         acc = DPAQ_S_W_PH(acc, aa0.a, xx0.a);
         acc = DPAQ_S_W_PH(acc, aa1.a, xx1.a);

         xx.c = LD32(&x[i-5]);
         xx.a = PACKRL_PH(xx.a, xx.a);
         acc = DPAQ_S_W_PH(acc, aa2.a, xx.a);

         xx.c = LD32(&x[i-7]);
         xx.a = PACKRL_PH(xx.a, xx.a);
         acc = DPAQ_S_W_PH(acc, aa3.a, xx.a);

         xx.c = LD32(&x[i-9]);
         xx.a = PACKRL_PH(xx.a, xx.a);
         acc = DPAQ_S_W_PH(acc, aa4.a, xx.a);

         xx.c = LD32(&x[i-11]);
         xx.a = PACKRL_PH(xx.a, xx.a);
         acc = DPAQ_S_W_PH(acc, aa5.a, xx.a);

         xx.c = LD32(&x[i-13]);
         xx.a = PACKRL_PH(xx.a, xx.a);
         acc = DPAQ_S_W_PH(acc, aa6.a, xx.a);

         xx.c = LD32(&x[i-15]);
         xx.a = PACKRL_PH(xx.a, xx.a);
         acc = DPAQ_S_W_PH(acc, aa7.a, xx.a);

         xx.c = (uint16_t)x[i-m];
         acc = DPAQ_S_W_PH(acc, aa8.a, xx.a);

         y[i] = EXTR_RS_H(acc, 12);

        /* software pipelining */
         xx0.c = LD32(&x[i+1-1]);
         xx0.a = PACKRL_PH(xx0.a, xx0.a);

         xx1.c = LD32(&x[i+1-3]);

     }
 return;
 }
```

Note that the data for the `DPAQ_S.W.PH` instructions are loaded using the `LD32` macro, which expands to two LWL/LWR instruction pairs. For example the first call to this macro,

```
LD32(a1, indata1, 0, a2, indata1, 4);
```

would expand to this instruction sequence:

```
lwl a1, 3+0(indata1)
lwr a1, 0(indata1)
lwl a2, 3+4(indata1)
lwr a2, 4(indata1)
```

Also, because the loop has been unrolled, we can interleave operations from different iterations. The operations are interleaved to increase the distance between instructions with data dependency, eliminating pipeline stalls.

Overall, we see substantial benefit from this combination of optimizations. On average the original loop, with first-order optimizations, takes 8884 cycles. When this loop is replaced by an unrolled, reordered sequence of SIMD and LWL/LWR instructions the average cycle count drops to 2550, a 3.5x performance improvement.

Obviously, this example is particularly well-suited to benefit from SIMD optimization, given the low level of data dependency between individual dot-product instructions, and it demonstrates how well these techniques can work. But even in less-than-ideal cases, the use of some or all of the techniques can still deliver excellent improvements in performance.

6.6 Split Summation

Split summation involves splitting a sum into partial sums, using n variables and 1/n iterations. A final summation of the partial sums is performed at the end of the process. It removes the accumulation dependency and thus improves the performance. Careful consideration should be given before using this technique as it might ruin the bit-exactness of the code. The gain here might not be really significant as MIPS is a single ALU processor, but still worth trying out for performance improvement. We were successfully able to improve the performance for a couple of functions using this method. Below is a snippet of code before and after split summation:

Original 3GPP code snippet:

```
for (i = 1; i <= m; i++)
    {
        L_sum = 0;
        for (j = 0; j < L_WINDOW - i; j++)
            L_sum = L_mac(L_sum, y[j], y[j + i]);

        L_sum = L_shl(L_sum, norm);
        L_Extract(L_sum, &r_h[i], &r_l[i]);
    }
```

Code after optimizing using Split Summation:

```
    for (i = 1; i <= m; i += 2)
    {
        sum0 = sum1 = sum2 = sum3 = 0;
        t0 = y[i];
            for (j = 0; j < L_WINDOW - i; j += 4)
            {
             t1 = y[j + i + 1];
             t2 = y[j + i + 2];
```

```
            sum0 = L_mac (sum0, y[j + 0], t0);
            sum1 = L_mac (sum1, y[j + 0], t1);
            sum2 = L_mac (sum2, y[j + 1], t1);
            sum3 = L_mac (sum3, y[j + 1], t2);

            t1 = y[j + i + 3];
            t0 = y[j + i + 4];

            sum0 = L_mac (sum0, y[j + 2], t2);
            sum1 = L_mac (sum1, y[j + 2], t1);
            sum2 = L_mac (sum2, y[j + 3], t1);
            sum3 = L_mac (sum3, y[j + 3], t0);
           }

        sumA = L_add(sum0, sum2);
        sumB = L_add(sum1, sum3);
        sumA = L_shl(sumA, norm);
        sumB = L_shl(sumB, norm);
        L_Extract(sumA, &r_h[i], &r_l[i]);
        L_Extract(sumB, &r_h[i+1], &r_l[i+1]);
    }
```

6.7 Loop Merging and Loop Splitting

These two methods are mainly helpful to improve loop efficiency.

6.7.1 Loop Merging

Loops with same loop counts inside a function are ideal candidates for loop merging which reduces loop overhead and hence improves the performance.

For example, windowing and calculation of energy in autocorr routine can be merged into a single loop as shown below:

```
L_sum = L_deposit_h(16);
for (i = 0; i < L_WINDOW; i+=1)
    {
       y[i] = mult_r(x[i], window[i]);
       L_tmp  = L_mult(y[i], y[i]);
       L_tmp = L_tmp >> 8;
       L_sum = L_add(L_sum, L_tmp);
    }
```

6.7.2 Loop Splitting

Loop splitting is breaking a large complex loop into shorter loops. This would enable efficient use of variables and reduce the complexity of the large loops. Code search will be an ideal candidate for loop splitting, where in the code inside the complex loop can be split and optimized.

6.8 Performance Impact of Second-Order Optimizations

The performance impact of the various optimizations discussed above is shown in Table 4 for each function optimized. The speedup, in terms of the average number of cycles per function call, of second-order optimizations ranges from a factor of 1.24 to a factor of 3.98 over the functions with only first-order optimizations.

Table 4 Performance Impact of Second-Order Optimizations on Critical Encoder Functions

Function	**Average Cycles Per Call** (with First-Order Optimizations)	**Average Cycles Per Call** (with Second-Order Optimizations)	Speedup
syn_filt	8859	2222	3.98x
residu	8884	2550	3.48x
chebps	349	138	2.52x
norm_corr	40366	18170	2.22x
Pitch-med_ol	46130	22122	2.08x
cor_h_x	14857	8579	1.73x
pred_lt4	12710	9199	1.38x
isp_az	788	602	1.3x
convolve	12223	9598	1.27x
autocorr	44953	36201	1.24x

The overall impact of both first-order and second-order optimizations on codec performance is shown in Table 5. As shown, the overall performance of the encoder improves by a factor of 5.8, from 572 MCPS to 99 MCPS.

Table 5 Performance Impact of Second-Order Optimizations on Overall AMR-WB Encoder Processor Load

Parameter	-03 Optimization (Reference)	First Order Opt-level1	First Order opt-level2	second order optimizations
Speed	572 MCPS	356 MCPS	130.82MCPS	99 MCPS
Speedup	-	1.61x	4.37x	5.77x

The table below provides the final performance after all the optimizations steps were completed. It provides information about peak load required, code memory required, memory required for constants/tables and memory required for static data. We did not measure the stack size for local variables required.

Table 6 Performance of AMR-WB Coder

Speed	Program memory	Tables/Constants	Static data
99 MCPS (Encoder)	176 kbytes	26.2 kbytes	~3.8 kbytes

7 Summary

This paper presents a number of optimization techniques used to improve the performance of the AMR-WB encoder. while maintaining the bit-exactness of the original code. Note that the effectiveness of the MIPS SDE toolchain allowed us to obtain this level of performance improvement while optimizing entirely at the C-language level. We have successfully reduced average AMR-WB processor load from 572 MCPS to 99 MCPS on a 24KEc core, a 5.8x improvement in performance, without optimizing the ACELP codebook due to time constraints. Considering the optimization of codebook search which accounts for 20% of the overall encoder, the load can be further reduced to around ~90 MCPS. This result largely comes from taking advantage of the DSP ASE architecture and its implementation in the 24KE core.

Although we have focused on the optimization of the AMR-WB encoder, the techniques discussed in this paper are applicable to the optimization of any wireless/wired application, including other speech codecs, echo cancellers, channel coders (convolution coding, viterbi decoding), channel equalization, etc. In fact, these techniques may be applied to any DSP or DSP-like application.

8 Notes on Measuring Processor Load

We measure the run-time performance of the AMR-WB encoder in terms of the load it puts on the processor, in MCPS (Millions of Cycles Per Second). This is the number of cycles the encoder takes to encode one second worth of a speech signal. Note that this is a constant for a given encoder; it doesn't depend on the clock rate of the system it runs on. For instance, an encoder that takes 99 MCPS to encode a speech signal on a 200 MHz 24KEc core still takes 99 MCPS to encode the signal on a 400 MHz 24KEc core (assuming memory latency is the same); however, because the processor runs at twice the clock rate, the encoder will process the input signal twice as quickly.

In this sense, the processor load is also a measure of the minimum clock rate that still lets the encoder run in real time. For example, an encoder that takes 99 MCPS to encode a speech signal requires at minimum a 99 MHz processor to encode in real-time. Of course in practice one would want to provide headroom for variations in the processing time per frame, so this is an unrealistic system design. Looking at this in another way, on a faster processor the encoder wouldn't need all of the processor's time to encode in real time. On a 200 MHz processor, for instance, the encoder could theoretically use only 49% of the processor time, leaving 51% of the processor available for other processing tasks.

8.1 Calculating Processor Load

Processor load is proportional to the number of cycles spent encoding each frame. The relationship is given below:

$$L_{MCPS} = 10^{-6} \cdot cR$$

where

```
L_MCPS = processor load (in MCPS)
C = cycles per frame
R = real-time frame rate (in frames/second)
```

R is a constant that depends on the sample rate of the input stream being encoded and the number of samples encoded in a frame:

$$R = \frac{\left(\frac{samples}{second}\right)}{\left(\frac{samples}{frame}\right)} = \frac{16000}{320} = (50)\frac{frames}{second}$$

Therefore, the processor load for an encoder can be computed with the following formula:

$$L_{MCPS} = 10^{-6} \cdot cR$$

$$L_{MCPS} = 10^{-6} \cdot c \cdot \left((50)\frac{frames}{second}\right)$$

$$L_{MCPS} = 5.(10)^{-5} \cdot c$$

So, for example, if it is experimentally determined that AMR-WB takes, on peak, 1976526 cycles to encode one frame, then the peak processor load is

$$L_{MCPS} = 5.(10)^{-5} \cdot c$$

$$L_{MCPS} = 5.(10)^{-5} \cdot 1976526$$

$$L_{MCPS} = 98.83MCPS$$

8.2 Measuring Cycles per Frame

Given the above formula for calculating L_{MCPS}, we see that the only variable that must be experimentally determined is c, the number of cycles required to encode one frame worth of input data. There are a number of ways to measure this in practice. We will discuss the method we used: reading cycle count from the cycle count register ($9) of CP0 (coprocessor 0).

The cycle count register is incremented on every other cycle of the system clock, and therefore it will account for latencies due to cache misses, pipeline stalls, etc. It will also need to be doubled to determine actual cycle count. To measure the cycles spent encoding a frame we need to read the cycle count register before and after a call to the main frame encoding routine or routines, and compute the difference. For simplicity and to prevent overflow, we can also just reset the counter at the start of encoding and read its value at the end. In the case of AMR-WB, the following function call constitute all of the processing for encoding a frame:

```
coder(&coding_mode, signal, prms, &nb_bits, st, allow_dtx);
```

Therefore, we can wrap them with a call to reset the cycle count register and a call to read the register:

```
asm volatile("mtc0 $0, $9");               // reset cycle counter
coder(&coding_mode, signal, prms, &nb_bits, st, allow_dtx);
asm volatile("mfc0 %0, $9" : "=r" (count)); // read cycle counter
count = count << 1;                        // double cycle count
```

At the end of this code sequence, the variable `count` will have the cycle count for encoding this single frame. Various statistics can be gathered for the encoding of a sequence of a frames, if desired. For example, one can measure average cycles per frame or peak cycles per frame. One can expect the cycles per frame to be higher for the first couple of frames, since the caches have not been warmed.

9 References

Listed below are related documents used as reference and available from MIPS Technologies.

1. MIPS32® Architecture for Programmers Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture
   MIPS Document: MD00374

2. MIPS® SDE 6.x Programmer's Guide
   MIPS Document: MD00428

3. Effective DSP Programming using MIPS® DSP Application Specific Extensions
   MIPS Document: MD00475

4. Efficient DSP ASE Programming in C: Tips and Tricks
   MIPS Document: MD00485

5. JPEG Decoder Optimization using MIPS® DSP Application Specific Extensions
   MIPS Document: MD00483

6. Optimizing G.729AB using MIPS® DSP Application Specific Extensions
   MIPS Document:MD00484

For CodeSourcery users:

7. MIPS® SDE LIBRARY
   MIPS Document: MD00623

8. MIPS® Toolchain Specifics
   MIPS Document: MD00624

Additional references:

9. 3GPP TS 26.190
   AMR Wideband Speech Code; Transcoding functions

10. 3GPP TS 26.173
    A*MR Wideband Speech Code; ANSI-C Code*

11. 3GPP TS 26.190
    AMR Wideband Speech Code; Test sequences

12. 3GPP AMR-WB (series 26) documentation is available free from 3GPP: http://www.3gpp.org

hLJűλY˙λÎλyƎ D¢тнн¢н {LJŜŜŎK /2ŘŜŎ dzăλyƎ aʟt{† 5{t !LJLJfŭŎI·ûλ2y´{LJŜŎλŦλŎ 9ĚűŜyãλ2yãΣ wŜØλăλ2y∀ м¢лн