Evaluation, Implementation and Optimization of FIR Filter Algorithms for TMS320VC5510 DSP Processor

Anita Szabó, Norbert Sram

Polytechnical Engineering College Subotica
M. Oreškovića 16, 24000 Subotica, Serbia
saboanita@vts.su.ac.yu
norbert.schramm@gmail.com

Abstract: Real-time digital signal processing applications are everywhere. The most essential part of these applications are FIR filters. For these systems to achieve the real-time demands it is essential that filtering executes as fast as possible. For systems with sever constraints - limited resources, low power consumption – stronger hardware is not a solution. The solution is to evaluate FIR filter algorithms in the context of hardware capabilities. Based on this we create optimized implementations for the FIR filter algorithms, both in high-level programming languages and in assembly. Based on the profiling results from these implementations we have been able to deduce possible optimization methods. With the usage of these methods we have been able to produce a highly optimized FIR filter implementation for the TMS320VC5510 DSP processor. To verify the results we conduct a benchmark which contains all the implementations and an optimized FIR filter implementation created by Texas Instruments.

1 Introduction

FIR [1,5,6,7] (Finite Impulse Response) filters are finite impulse response digital filters. FIR filters could be written in the following form, where p is the filter order, x(n) is the input signal, y(n) is the filtered signal, biare the filter coefficients.

$$y(n) = \sum_{i=0}^{p} b_i x(n-i)$$

FIR [1,5,6,7] filters are the vital components of the digital signal processing applications [2]. They represent a critical point in real-time digital signal processing applications, therefore their as far as possible optimal implementation means a significant advantage in the runtime performance of these applications.

There are several algorithms for FIR [1,5,6,7] filters, each have their advantages and disadvantages. To use the appropriate one we must take into consideration the possibilities provided by the hardware. It's not enough to choose an algorithm based upon their complexity. To achieve maximum runtime performance, context dependent optimizations are required. All the evaluated filter algorithms have an O(n) complexity, where n is the filter order. To optimize the implementations we need to evaluate the algorithms and their properties. We will present an evaluation of algorithms in the following sections. All implementations can be found in Appendix A.

2 Algorithms

2.1 FIR Buffer Shifter

The buffer shifter algorithm is the simplest to implement. The buffer symbolizes the filter window, which size is equal to filter's order, that is, higher filter order means bigger buffer size. This is the drawback of the algorithm. It is required to shift the filter window for every filtered sample and to insert the new sample into buffer. In case of processors, which support the MACD (Multiply Accumulate With Data Move) instruction, this algorithm is optimal. Multiplication, accumulation and buffer shifting could be executed in one operation, leading to a very efficient implementation.

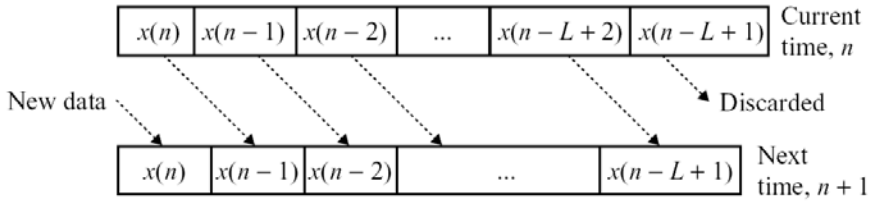

Figure 1
Symbolizes the actions required in case of an incoming sample for the buffer shifter algorithm. It is necessary to move all accumulated samples in order to insert a new one. x is the filter window, L is the order of the filter.

2.2 FIR Double Buffer

In case of the double buffer algorithm, the buffer that forms the filter window has a size which is twice the size of the filter order. In this case there is no need to shift the samples in the buffer, because the incoming samples are written into two positions in the buffer, also a state parameter is modified at the end of filtering. This state parameter determines the buffer slot for the input sample and the beginning of the buffer for the input sample filtering. In case of simple hardware this is the most optimal implementation, also for processors which support vectorized operations it can be used extremely efficiently as well. This implementation is redundant; it consumes twice as much memory as the other algorithms. This could be a drawback in case of systems with limited resources or filters with high order.

2.3 FIR Circular Buffer

In case of circular buffer algorithm the size of the buffer is equal to the order of the filter. The buffer is represented as a circle. We start from a defined position and we go around. The algorithm has a state which represents the starting point in the circle. This implementation is extremely efficient is case of processors which support circular addressing mode. This can be found in most of the DSP (Digital Signal Processing) processors. On the other hand, without hardware support the implementation performs poorly, because of the manual bound checking for the end of the circle.

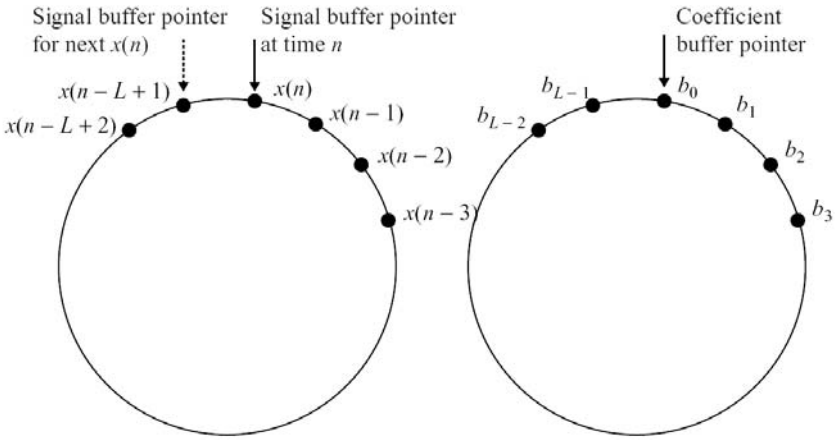

Figure 2

The image on the left shows that for every incoming sample we rotate the circle buffer counterclockwise. As it shows on the left image the coefficient buffer doesn't get rotated - it always starts from the same point.

3 Optimization Methods

To achieve the desired performance in our application, often the right algorithm for the hardware is not enough. For example, a simple C implementations use floating point calculations. Some processors do not support floating point operations, only integer arithmetic. For these processors the provided compilers produce software emulation for the floating point calculations. Often the processor's word length is shorter than the floating point number's word length. This means that the execution of the generated code will be slow and it will occupy more space. The solution is to represent the decimal numbers in fixed point format.

3.1 Fixed Point Formats

The most common fixed point format is Q0.15. It can be represented as the following:

$$(x)_{10} = b_1 2^{-1} + b_2 2^{-2} + \ldots + b_M 2^{-M} = \sum_{m=1}^{M} b_m 2^{-m}$$

The word length B= (M + 1) bit, where M is the number of magnitude bits plus one sign bit. The MSB (most significant bit) bit is the sign bit, which represents the sign of the number. The value of the sign bit is 0 if the number is positive, 1 if the number is negative.

The fixed-point DSP processors usually use the second complement format, in this way the addition and the subtraction can be realized with the same hardware. General formula to calculate the decimal value in case of B binary decimal number:

$$(x)_{10} = -b_0 + \sum_{m=1}^{M} b_m 2^{-m}$$

The Q0.15 fixed point format can be used to represent decimal values from -0.999 to 0.999. This is adequate enough for FIR filter calculations.

3.1.1 Fixed Point Formats in ANSI C

The C programming language doesn't contain fixed point types, so we need to implement our own fixed point library. The Q0.15 format is isomorphic with a 16 bit signed integer. Simply a type synonym is created for that built-in variable type which suits our needs; in this case this is a 16 bit integer. In case of a 16 bit processor int is 16 bit long for the C compiler. There are situations when more the one variable type is 16 bit long. This does not mean that it's unimportant which

variable type is used; because the compiler generates different code for different types, apart from the fact that their length is identical. This is the case with the C compiler provided with Texas Instruments Code Composer Studio, which is used for C55xx processors. The difference between the speeds of the generated codes can be drastic. In case of C55xx the most effective code is achieved when short int is used as 16 bit integer, which in our case is marked with i16_t synonym. In this case the compiler takes full advantage of every available register. The highest priority for the fixed point library is execution performance. We have to keep in mind that we want to create simple operations like addition, subtraction, multiplication, and division. In these cases the preparation for the function call makes up most of the execution time. This can be prevented if we use macros provided by the C language. In this case calls to our routines will be replaced with the appropriate operations at compile time, thus eliminating the function call. The same effect can be achieved with the usage of the inline keyword. It's a more elegant and type safe solution. It's advised to use this if our compiler supports it. Also it's standard C99.We can implement our algorithms to use fixed point calculations using the fixed-point C library.

4 Fixed Point FIR Implementations

In case of the buffer shifter implementation, considering the C55xx processor's hardware this is not the most ideal algorithm. Even with an assembly implementation, we wouldn't achieve significant speed-ups in our application, because we wouldn't eliminate buffer shifting. The double buffer C implementation is extremly efficienct, we would not gain a speed up by doing assembly based optimization. The C55xx processor has circular addressing capabilites. Unfortunately the C complier was not able to generate code which takes advantage of this feature. To take full advantage of this algorithm and the hardware we have to implement it in assembly. The provided implementation uses circular adressing instead of simple pointer arithmetic.

5 Summary

Efficiency comparison of the implementations was done on TMS320VC5510 DSP [3, 4] processor with the help of Code Composer 3.1. The implementations for C55xx processor and the ANSI C [8] implementations are compared to each other and to the FIR filter implementation provided by Texas Instruments in their library. Results are shown on Figure 3.

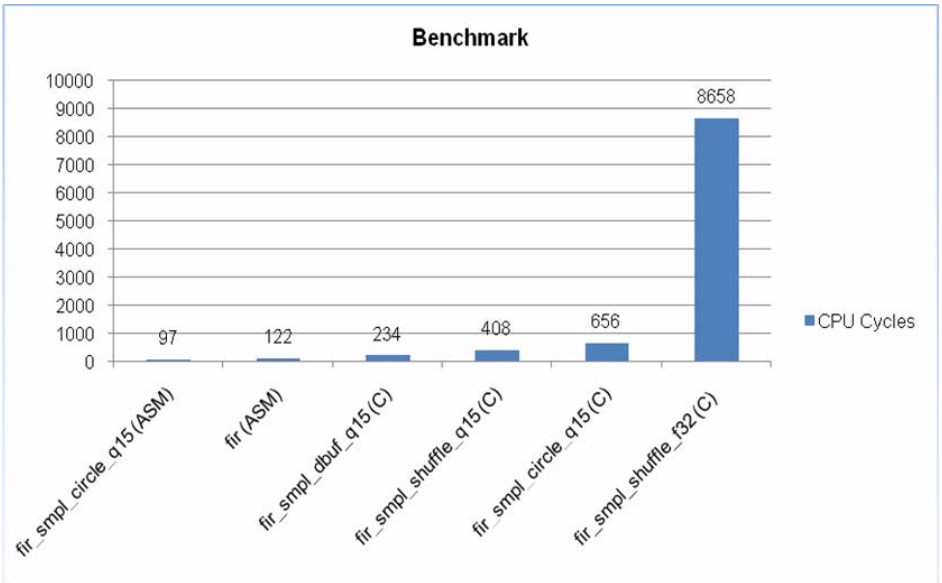

Figure 3
The comparison of implementations based on the number of cycles it takes to execute on the TMS320VC5510 DSP processor

All implementations are C callable and are tested in a benchmark program written in C. The fastest implementation is the assembly implementation of the circular algorithm. This implementation is 20% faster than the optimized filter provided by Texas Instruments. The double buffer ANSI C [8] implementation was 1.9 times slower than the Texas Instruments filter, but its platform independent and it is an implementation made in a high level programming language. This is the most ideal implementation if we use a high level programming language. With compiler friendly, clean and simple code we can achieve significant execution performance as shown by the fir_smpl_dbuf_q15 implementation. Based on the fact that the ANSI C [8] version of circular FIR implementation is 6.7 times slower than the assembly version we can see that if we want to achieve maximum performance we cannot rely on the abilities of compilers even if it is produced by the hardware vendor itself. Although some C compilers support circular buffer optimizations, we should always check the generated code. The biggest speed-up is achieved by using fixed-point calculation instead of floating point calculations. This is a huge leap in performance for every processor which only supports integer based arithmetic. By switching the calculation operations in the buffer shifter implementation 21, 2 times speed-up was achieved. Another advantage of this method is that it is simple to implement in a high level programming language.

Conclusions

Performance of algorithm implementations are affected by a lot of factors. Context dependent optimizations are the key to high performance implementations. Assembly level programming is not necessary to obtain notable performance boost, and should only be used if performance is the most critical factor.

References

[1] Sen M. Kou, Bob H. Lee: Real-Time Digital Signal Processing I Implementations and Applications 2nd Edition, 2006

[2] Rulph Chassaing: Digital Signal Processing and Applications with the C6713 and C6416 DSK, 2005

[3] Texas Instruments: TMS320C55x DSP CPU Reference Guide, 2004

[4] Texas Instruments: TMS320C55x DSP Mnemonic Instruction Set Reference Guide, 2002

[5] Texas Instruments: Efficient MSP430 Code Synthesis for an FIR Filter, 2007

[6] Texas Instruments: Digital FIR Filter Design Using the MSP430F16x, 2004

[7] Texas Instruments: Efficient Implementation of Real-Valued FIR Filters on the TMS320C55x DSP, 2000

[8] Brian W. Kernighan, Dennis M. Ritchie: C Programming Language 2nd Edition, 1989

Appendix A

FIR buffer shifter, floating point arithmetic implementation:

```
f32_t fir_smpl_shuffle_f32( u32_t order, f32_t sample, const f32_t* coeffs, f32_t* buffer )
{
 f32_t accu = 0.0f;
 u32_t i = 0;
 --order;
 for( ; i < order; ++i ) {
  buffer[i] = buffer[i+1];
  accu += buffer[i] * coeffs[i];
 }
 buffer[order] = sample;
 accu += buffer[order] * coeffs[order];
 return accu;
}
```

FIR double buffer, floating point arithmetic implementation:

```
f32_t fir_smpl_dbuf_f32( u32_t order, f32_t sample, const f32_t* coeffs, f32_t* buffer, i32_t* state )
{
 f32_t accu = 0.0f;
 f32_t* pBuffer = buffer + *state;
 i32_t i = 0;
 buffer[order + *state] = *pBuffer = sample;
 for ( ; i < order; ++i ) {
  accu += *pBuffer++ * *coeffs++;
 }
 if ( --( *state ) < 0 )
  *state += order;
 return accu;
}
```

FIR circular buffer, floating point arithmetic implementation:

```
f32_t fir_smpl_circle_f32( u32_t order, f32_t sample, const f32_t* coeffs, f32_t* buffer, i32_t* state )
{
 f32_t accu = 0.0f;
 i32_t i = order - 1;
 buffer[*state] = sample;
 if ( ++( *state ) >= order ) {
  *state = 0;
 }
 for ( ; i >= 0; --i ) {
  accu += buffer[*state] * coeffs[i];
  if ( ++( *state ) >= order )
  *state = 0;
 }
 return accu;
}
```

Fixed point C library implementation:

```
#ifndef __FIXED_POINT_H_INCLUDED__
#define __FIXED_POINT_H_INCLUDED__
#include <types/types.h>
typedef i16_t q15_t;
#define Q15_NUMBER_OF_FRACTION_BITS 15
#define Q15_SLOPE ((f32_t)1/32768)
#define Q15_INVERZ_SLOPE 32768
#define i2q15(num)((num)<<Q15_NUMBER_OF_FRACTION_BITS)
```

```
#define f2q15(num)(num*Q15_INVERZ_SLOPE)
#define q152i(num)((num)>>Q15_NUMBER_OF_FRACTION_BITS)
#define q152f(num)((f32_t)num*Q15_SLOPE)
#definemul_q15(rhs,lhs)(\
 ((i32_t)(rhs)*(lhs))>>Q15_NUMBER_OF_FRACTION_BITS)
#define div_q15(rhs,lhs)(\
 ((((i32_t)(rhs))<<Q15_NUMBER_OF_FRACTION_BITS)/(lhs)))
#endif
```

The fixed point arithmetic implementations are similar to the floating point implementations, only one implementation will be provided, as a reference.

FIR double buffer, fixed point arithmetic implementation:

```
q15_t fir_smpl_dbuf_q15( u16_t order, q15_t sample, const q15_t* coeffs, q15_t* buffer, i16_t* state )
{
 q15_t accu = 0;
 q15_t* pBuffer = buffer + *state;
 i16_t i = 0;
  buffer[order + *state] = *pBuffer = sample;
 for ( ; i < order; ++i ) {
  accu += mul_q15( *pBuffer++, *coeffs++ );
 }
 if ( --( *state ) < 0 )
  *state += order;
 return accu;
}
```

FIR circular buffer, assembly implementation:

```
.global _fir_smpl_circle_q15
.text
; Prototype of the funtion:
; i16_t fir_smpl_circle_q15(u16_t  order,          T0
;                           i16_t  sample, -       T1
;                           const i16_t* coeffs,   AR0
;                           i16_t*  buffer,  -     AR1
;                           i16_t*  state );       AR2
              .asg T0, order
              .asg T1, sample
              .asg AR0, coeffs
              .asg AR1, buffer
              .asg AR2, state
              .asg AC0, accu
```

```
_fir_smpl_circle_q15:
        psh mmap(ST1_55)
        psh mmap(ST2_55)
        psh mmap(ST3_55)
        bset SMUL, ST3_55
        or #0x340, mmap(ST1_55)
        bset AR1LC
        mov mmap(buffer), BSA01
        mov mmap(order), BK03
        add *state, buffer
        mov sample, *buffer+
        mar *(coeffs + order)
        mar *coeffs-
        sub #0x2, order
        mov order, CSR
        mpy *buffer+, *coeffs-, accu
        || rpt CSR
        mac *buffer+, *coeffs-, accu
        mov buffer, AC1
        sub mmap(BSA01), AC1
        mov AC1, *state
        pop mmap(ST3_55)
        pop mmap(ST2_55)
        pop mmap(ST1_55)
        mov hi(AC0), T0
        || ret
```