View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repository of the Academy's Library

Software Implementation of the Recursive Discrete Fourier Transform

Márton Kovács, Zsolt Kollár
Budapest University of Technology and Economics
Budapest, Hungary
Email: kollar@hvt.bme.hu

***Abstract*—The Discrete Fourier Transform is one of the fundamental operations in digital signal processing. This paper presents a software based implementation of the less known Recursive Discrete Fourier Transform on the PC x86 architecture and Microchip PIC30 microcontroller. The results are compared with an efficient/optimized Fast Fourier Transform implementation. Both algorithms require complex operations with real valued adders and multipliers, thus a complex result has to be calculated separately for the real and imaginary parts. During the implementation a floating-point number representation is applied. The comparison is performed based on the required resources and computational time.**

I. Introduction

The Discrete Fourier Transform (DFT) is used in many engineering applications: it is not limited only to the field of signal processing, but it can be widely adopted for solutions in telecommunications, data compression and measurements as well. Direct implementation of the DFT is computationally inefficient, thus signal processing applications tend to implement one of the various Fast Fourier Transform (FFT) algorithms instead. The basic operation of these algorithms is block oriented, but there are some problems which require a continues, sample-by-sample calculation of the spectra. Such problems can be spectral sensing for cognitive radios [1] where accurate and continues measurements are required for the fast response in the usage of the radio frequency or the processing of DTMF signals using Goertzel's algorithm [2].

The Recursive DFT (R-DFT) provides an alternative computation method to acquire the DFT components of a signal, which applies the observer-based structure [3] and recalculates the spectral values for every new incoming sample in a sliding manner. This paper presents a possible implementation based on the x86, Streaming SIMD Extension (SSE) and 16-bit architectures and compares the results with an optimized FFT.

The paper is organized as follows. First, in Section II, a short theoretical background is given for the DFT, FFT and R-DFT methods and a comparison of their computational complexity is presented. Section III presents efficient implementations of the R-DFT on the different architectures. The comparison of the two algorithms implementation performance is given in Section IV. Finally, the results are concluded and possible further improvements are discussed in Section V.

II. Theory

In this section the operation of the DFT and the R-DFT are given. The main difference between the operation of the DFT and the R-DFT is that the DFT operates on a block of input samples with a length of N elements, thus it has to wait for the entire block before it can be executed, meanwhile the R-DFT operates as closed loop control system, thus it operates in a sample-by-sample manner.

A. DFT and FFT

Mathematically, the ith spectral component of an N-point DFT (X_m) can be calculated for an N-sample long discrete signal ($x[k], k = 0, 1, ..., N-1$) as

$$X_m = \sum_{k=0}^{N-1} x[k] e^{-j\frac{2\pi}{N}km}, \qquad m = 0, 1, ..., N-1, \tag{1}$$

where $j = \sqrt{-1}$.

The calculation of one spectral component requires N complex multipliers and $N-1$ complex adders, so the computational complexity for N frequencies is $\mathcal{O}\left(N^2\right)$.

The FFT provides mathematically the same results as the DFT, but it has a modified structure, which reduces the number of elementary operations. There are many FFT algorithms like split-radix FFT, Prime-factor FFT, Brunn's FFT algorithm, etc. [4]. Its computational complexity, depending on the number of samples, can be reduced to $\mathcal{O}(N \log_2 N)$, which is significantly lower than the requirements for the DFT.

B. R-DFT

The R-DFT realizes the observer-based structure [3], which calculates the DFT values of the incoming signal for every single new input. The observer theory is based on the prediction-correction scheme, where the observer is a system, which can copy the state variables of the observed system. By suggestion of Péceli the DFT can be also realized in an observer-scheme. If $g_m[k]$ and $c_m[k]$ are given as

$$g_m[k] = \frac{1}{N} e^{-j\frac{2\pi}{N}km}, \qquad m = 0, 1, ..., N-1, \tag{2}$$

$$c_m[k] = e^{j\frac{2\pi}{N}km}, \qquad m = 0, 1, ..., N-1, \tag{3}$$

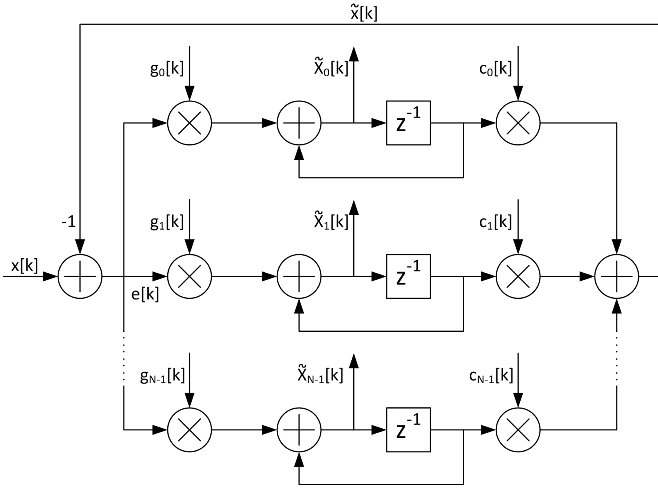

Fig. 1: Schematic structure of the observer-based R-DFT

then $X_m[k]$ gives the DFT of the last $k' = k - N, \ldots, k - 1$ samples.

The structure of the R-DFT scheme can be seen in the Fig. 1. It calculates an error signal $e[k]$ from the difference of the input and the feedback signals, which then will be modulated with $g_m[k]$ in every m^{th} branch and the integrators will be updated with the modulated value. The values of the integrators are demodulated with $c_m[k]$ and summed in order to create the feedback signal. In case where the state variables are perfectly estimated, the feedback is identical with the input, thus the error signal is zero.

The complexity depends on the number of branches: After receiving one input sample, all N resonators operate. The first valid spectral components are therefore acquired once $N+1$ input values have been received. So, the computational complexity is $\mathcal{O}\left(N^2\right)$. It can be seen that the complexity is the same as of the DFT's, but the calculation is equally distributed in time over the N samples. Also the spectra of the N sample long block which is shifted by one sample can be calculated in $\mathcal{O}\left(N\right)$.

The beneficial property of the R-DFT compared to other resonator-based solutions (like Goertzel's [2] or Jacobsen's solutions [5]) is that the arithmetical inaccuracy of the resonator's pole does not cause divergence due to feedback [6]. As the poles of system are located on the unit circle, inaccuracies due to the finite numerical representation of these values may lead to divergence or convergence to zero.

C. Comparison of the computational complexity

The theoretical comparison of the FFT and R-DFT methods can be discussed for various scenarios.

1) Offline: First, if an offline calculation is considered for a block of N samples, the computational complexity of the FFT is proportional to $N \log_2(N)$, while the complexity of the R-DFT is proportional to N^2. In the case of offline calculation, all data are available in memory, so it is not needed to wait for input signal.
2) OnlineSample by sample calculation: A second scenario is where the samples are coming from an A/D converter with a given sampling rate. In this case the FFT algorithm has to wait for N samples to proceed with the calculations, so the computational complexity remains unaltered compared to the previous case. On the other hand, for the R-DFT the feedback loop operates after each incoming sample, so after the arrival of the Nth sample, only N operations with a latency of a single stage is required to calculate the spectra, thus the operations are distributed over the block.
3) Sliding manner: When continuous evaluation of DFT is required, the FFT has to be calculated in sliding window manner over the sampled signal for each block repeatedly. In this case the R-DFT provides the spectral values much faster [1] with a significantly reduced complexity proportional to N.

The summary of the required computational complexity and latency in the various scenarios is presented in Table I.

III. Implementation

The PC implementation was developed on x86 architecture using C language and Microsoft Visual Studio 2013 development environment. Two solutions are investigated. The first realization uses only the x86 instructions to calculate the floating point numbers. After that the algorithm was optimized by SSE instruction set as follows: every single elementary complex operation (addition and multiplication) is converted to SSE instruction, which are calculated on 128-bits data units. In the second solution, the algorithm is also implemented on a Microchip PIC30 with a 16-bits architecture in C language under MPLAB X, to reveal how many operations are executed during DFT processing, which was not observable on PC, because the operating system hides this information.

A. x86

Most of today's desktop PC systems are based on the x86 architecture. In the first implementation only x86 instructions are used, which are then optimized by SSE instructions and these realizations are compared.

1) Direct implementation: First, the direct implementation maps the structural elements of R-DFT - as presented in Section II – into control statements. The algorithm contains two nested for-loops, which is iterating for every new input sample. This outer loop is responsible for the time domain, and increments the discrete time-variable k. In every discrete time slice, the spectral components are calculated in an embedded for-loop. The internal loop increments the m frequency variable for the frequency domain, modulates the error signal with $g_m[k]$ complex value, and the integrator is updated with the result of the modulator. Finally, all of the integrators are demodulated with the multiplication of $c_m[k]$ and the outputs of the demodulators are summed to generate the feedback signal.

All modulators and demodulators cover complex multiplication operations. The values of $g_m[k]$ and $c_m[k]$ can be

TABLE I: Computational complexity and latency for the various scenarios

	FFT		R-DFT	
Scenario	Complexity	Latency	Complexity	Latency
Offline	$N \log_2 N$	$N \log_2 N \cdot t_{cFFT}$	N^2	$N^2 \cdot t_{cRDFT}$
Online: Block manner	$N \log_2 N$	$N \cdot t_s + N \log_2 N \cdot t_{cFFT}$	N^2	$N \cdot t_s + N^2 \cdot t_{cRDFT}$
Online: Sliding manner	$N \log_2 N$	$t_s + N \log_2 N \cdot t_{cFFT}$	N	$t_s + N \cdot t_{cRDFT}$

computed offline and saved into an array, so the modulators have to assign only the index of the pre-calculated operands. The algorithm operates on 64 bit double precision data units using the processors's floating point arithmetical unit.

2) SSE optimization: The second realization using the SSE, is a Single Instruction Multiple Data (SIMD) instruction set extension of the x86 architecture, which was developed by Intel in 1999 and the company's later processors processors support several version of SSE since Pentium III. The first version of instruction set offers 70 new instructions that can be used to calculate single precision values more efficient. The SSE2 already supports double precision calculations and SSE3 has operations supporting complex arithmetics [7].

SIMD means that the same instruction is executed on multiple data. SSE instructions perform their operations on 128 bit data units, which can be defined in various ways according to the applied data type. In this paper the decomposition to 2-element double vectors is applied.

This scenario uses the same program structure as the direct implementation, but it is optimized for complex additions and complex multiplications. The operations are using both x86 and SSE instructions. This solution is optimal in that sense that the execution is performed on 128 bit data units instead of 64 bit, which means that two double values can be stored in one register. Using the SSE operation ADDPD which adds two registers vertically, the complex addition can be performed with one ADDPD instead of two FADD instructions. Since both instruction's latency is 3 cycles, we are expecting to halve the duration of execution here.

The optimized complex multiplication works on the same data types as the non optimized version, but it is not enough to modify only the FMUL instruction to MULPD as for addition, because it can not be evaluated vertically for vectors. The structure of the calculation has to be redistributed, the resulting optimized source code is as follows:

```
inline void sse_complex_mul(__m128d *result, __m128d *x,
                            __m128d *y)
{
    __m128d aa = _mm_movedup_pd(*x);
    __m128d bb = _mm_movedup_pd(_mm_shuffle_pd(*x, *x, 1));
    __m128d cd = *y;
    __m128d dc = _mm_shuffle_pd(cd, cd, 1);

    *result = _mm_addsub_pd(_mm_mul_pd(aa, cd),
                            _mm_mul_pd(bb, dc));
}
```

The direct implementation of the multiplication contains 4 FMUL (5 cycles latency) and 2 FADD (3 cycles latency) operations, which results in a total latency of 26 cycles. In this case the MULPD instruction has a latency of 5 cycles as well. ADDSUBPD, MOVDDUP and SHUFPD need 1 cycle to be completed. Summarized, the complex multiplication takes 17 clock cycles, as result a maximal time reduction to 65% may be achieved. These estimations are rather optimistic, which do not take into account the load/store operations, and presumes that all data are available in the registers.

B. Pic30

On the x86 architecture, with the usage of an operating system only the execution time of the algorithm can be measured. Therefore the R-DFT is also implemented on a Microchip PIC30 16-bit architecture to compare the number of the executed instructions for each variant of algorithm, which define exactly its entire execution time. In the full paper the results of the R-DFT implementation on a 16-bit microcontroller architecture will be evaluated and the required resources and computational cycles will be discussed. The results will be presented in the final paper.

IV. Implementation results

The comparison presented in this section covers two run cases. First, the two previously introduced x86 implementation of the R-DFT are compared with each other, after that, the R-DFT and FFT implementations are analyzed and investigated.

In Figure 2 the execution time of the direct implemented and the SSE optimized algorithm of R-DFT depending on the signal length is shown, where $N = 512$, the sample rate is considered to be $100\ \mathrm{k}S/s$ and the analyzed signal lengths are between 1-10 seconds. The measurement show an average of about 64% time reduction in case of SSE runs, which is in the expected and pre-calculated 50-65% range. It can be also seen that execution time for both of the two implementation is linearly dependent on the length of the input signal.

The FFT and R-DFT use different concepts for computation. As with the FFT, a fixed N-size block is always defined which has to be filled with data first (and the program waits so long), while the R-DFT performs its operations for each new input sample. Therefore the algorithm of FFT has to be run using a sliding window on the input, which means that it has to push each next input value into the window, removing the oldest stored value at the same time. For the comparison we used the FFTW as an efficient implementation for the FFT. It is a C subroutine library for computing the DFT in one or more dimensions, of arbitrary input size, and of both real and complex data. FFTW is free software, which provide the FFT library of choice for most applications [8]. FFTW is used on one hand to compare the applied resources, rapidness of

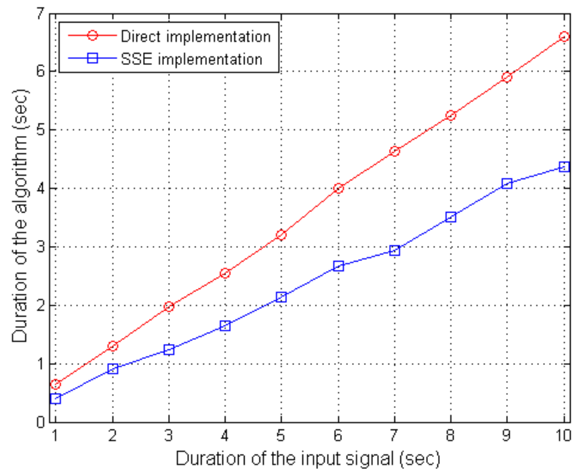

Fig. 2: Execution time of the direct and the optimized version of the R-DFT implementation at $N = 512$

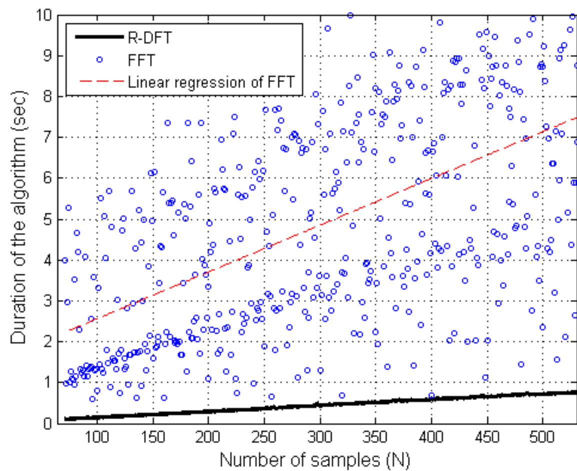

Fig. 3: Execution time of FFTW (FFT) and R-DFT in function of N

algorithms, on the other hand to verify the accuracy of output values.

Figure 3 shows the execution time of the FFT and the R-DFT on 1 second input signal, which sample rate is $100\ \mathrm{k}S/s$, depending on its spectral resolution in the range $N = 64-530$. The measurement indicate that the R-DFT is almost always faster then FFT if a sliding DFT calculated for each new sample is required. The other notable result, shown in Fig. 3, is that the execution time of FFT has a large deviation from its expected values, but it has a growing trend in function of N. By contrast, the execution time of R-DFT seems linearly proportional to N. The reason of this effect derive from the algorithmic specialty, because if N is a prime or it has a large (larger then 11) prime divisor, than the "divide and conquer" principle can not be applied, while the number of R-DFT's adders and multipliers is proportional to N.

V. Conclusion

This paper presents possible implementations of the less-known R-DFT on different architectures, the optimization possibilities of the algorithm are investigated. The results show that the R-DFT is capable to calculate the spectral components of input signal significantly faster in a sliding manner than the point-by-point usage of the FFT.

In the future work the implantation of the R-DFT algorithm on graphics processing unit (GPU) and field-programmable gate array (FPGA) is to be considered; the hardware implementation should further improve the presented software implementations.

Acknowledgment

This work was supported by the János Bolyai Research Fellowship of the Hungarian Academy of Sciences.

References

[1] L. Varga, Zs. Kollár, and P. Horváth, "Recursive Discrete Fourier Transform based SMT receivers for cognitive radio applications," in *2012 19th International Conference on Systems, Signals and Image Processing IWSSIP*, Apr. 2012, pp. 130–133.

[2] N. Bhavanam and V. Midasala, "DTMF tone generation and detection using goertzel algorithm with MATLAB," in *Proceedings of International Conference on Innovation in Electronics and Communications Engineering (ICIECE-2013)*, Aug. 2013, pp. 47–52.

[3] G. Péceli, "A common structure for recursive discrete transforms," *IEEE Transactions on Circuits and Systems*, vol. 33, no. 10, pp. 1035–1036, Oct. 1986, DOI: 10.1109/TCS.1986.1085844.

[4] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[5] E. Jacobsen and R. Lyons, "The sliding DFT," *IEEE Signal Processing Magazine*, vol. 20, no. 2, pp. 74–80, Mar. 2003, DOI: 10.1109/MSP.2003.1184347.

[6] B. Csuka, I. Kollár, Zs. Kollár, and M. Kovács, "Comparison of signal processing methods for calculating point-by-point discrete fourier transforms," in *2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA)*, April 2016, pp. 217–221.

[7] *Intel 64 and IA-32 Architectures Software Developers Manual*, Intel Corporation, 2016, URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[8] FFTW, "FFTW (Fast Fourier Transform library)," http://www.fftw.org/, 2017.