Efficiency Comparison of DFT/IDFT Algorithms by Evaluating Diverse Hardware Implementations, Parallelization Prospects and Possible Improvements

Danijela Efnusheva, Natasha Tagasovska, Aristotel Tentov, Marija Kalendar
SS. Cyril and Methodius University - Faculty of Electrical Engineering and Information Technologies
Rugjer Boshkovik bb, PO Box 574, 1000 Skopje, Macedonia
E-mail: {danijela, ntagasovska, toto, marijaka}@feit.ukim.edu.mk

***Abstract*—In this paper we investigate various algorithms for performing Fast Fourier Transformation (FFT)/Inverse Fast Fourier Transformation (IFFT), and proper techniques for maximizing the FFT/IFFT execution speed, such as pipelining or parallel processing, and use of memory structures with pre-computed values (look up tables - LUT) or other dedicated hardware components (usually multipliers). Furthermore, we discuss the optimal hardware architectures that best apply to various FFT/IFFT algorithms, along with their abilities to exploit parallel processing with minimal data dependences of the FFT/IFFT calculations. An interesting approach that is also considered in this paper is the application of the integrated processing-in-memory Intelligent RAM (IRAM) chip to high speed FFT/IFFT computing. The results of the assessment study emphasize that the execution speed of the FFT/IFFT algorithms is tightly connected to the capabilities of the FFT/IFFT hardware to support the provided parallelism of the given algorithm. Therefore, we suggest that the basic Discrete Fourier Transform (DFT)/Inverse Discrete Fourier Transform (IDFT) can also provide high performances, by utilizing a specialized FFT/IFFT hardware architecture that can exploit the provided parallelism of the DFT/IDF operations. The proposed improvements include simplified multiplications over symbols given in polar coordinate system, using sine and cosine look up tables, and an approach for performing parallel addition of N input symbols.**

***Keywords:* Cooley-Tukey, DFT/IDFT, FFT/IFFT, Intelligent RAM, look up tables, OFDM, pipeline and parallel processing, polar coordinate system.**

I. INTRODUCTION

The Fast Fourier Transform and the Inverse Fast Fourier Transform are widely used efficient and fast techniques for computing the Discrete Fourier Transform and the Inverse Discrete Fourier Transform, respectively. The reduced number of FFT/IFFT calculations required for performing the same set of DFT/IDFT objectives, results in decreasing the execution complexity from $O(N^2)$ to $O(N\log_2 N)$, on algorithmic level [1][2]. Consequently, FFT/IFFT modules are extensively used for analysis and implementations of communication systems with real-time data transmission requirements. Additionally, the recent advances in chip production technologies, as well as in FFT/IFFT algorithms, have resulted with production of several FFT/IFFT chips that allow significant processing speed up (multi Gb/s) [3]-[6], as well as decreased chip area and reduced energy consumption [7]-[9].

The most important and used FFT/IFFT application is the orthogonal frequency division multiplexing (OFDM), which is the dominant transmission technique used in the 802.11 set of WLAN standards [10]. This advanced modulation technique divides the available spectrum into many overlapping subcarriers, thus allowing more effective channel utilization and reducing inter-symbol interference (ISI) and inter-carrier interference (ICI) caused by multi-path effect [11].

IFFT/FFT modules execute the main functionality in OFDM systems on sending/receiving side, allowing signals to be converted from frequency/time domain to time/frequency domain [2]. Actually, the process of modulation of the subcarriers in the channel with symbol information and making them orthogonal to each other is performed by means of IFFT on the sending side, whereas the FFT is used for efficient demodulation of the received signal. By including the IFFT and FFT modules in OFDM systems, the signal processing complexity is reduced (at both, transmitting and receiving side) and higher transmission rates are achieved.

Efficient FFT/IFFT implementation is a topic of continuous research in the recent years [3][4], and the main goal is reducing the processing complexity of the FFT/IFFT calculations. In general, there are two directions in this area of research. The first one refers to developing algorithms for FFT/IFFT and their optimization. Best known algorithms in this field, worth mentioning in this paper are radix-2, radix-4, radix-8 and split-radix variations of the Cooley-Tukey (C-T) algorithm [12][13] as well as Winograd algorithm [14] By increasing the base in C-T algorithm, the number of operations is decreased, resulting in FFT/IFFT split-radix implementation becoming superior compared to Winograd algorithm [15]. The second approach is focused on hardware architecture improvements and optimization of the FFT/IFFT module. This includes adequate techniques for parallel processing and pipelining, memory structures for preservation of previously calculated results, as well as

dedicated components (usually multipliers) for more rapid calculations [16][17].

The FFT/IFFT implementation, in order to obtain better performances, can be achieved with various hardware components, including digital signal processors (DSPs), general purpose processors (GPPs), smart memories (IRAM), application specific integrated circuits (ASICs) or specialized circuits implemented on FPGA. The research has shown that GPPs and DSPs [14] are programmable and flexible, but cannot completely satisfy the fast processing requirements. On the other hand, the IRAM architecture is insufficiently explored, but considering its abilities to provide high memory bandwidth and strided memory accesses, it is expected to provide promising results [18]. Most of the FFT/IFFT implementations are realized as specialized logic circuits, characterized with different forms of parallel computing. FPGA components are utterly suitable for implementation of this type of circuits, providing a tradeoff between speed, cost, flexibility and programmability [2][6][15][16][19].

The aim of this paper is to investigate various algorithms for FFT/IFFT computation and to discuss their hardware implementation that mostly satisfies the high speed processing requirements. Actually we talk about several parallelization techniques and methods, emphasizing the problem of data dependences in the FFT/IFFT calculations that limit the processing speed and the maximal utilization of the available hardware resources. Therefore, we suggest that the basic DFT/IDFT calculations are very suitable for parallelization, which can be further improved by implementing several optimizations of the addition and multiplication operations over complex numbers. In order to be effective, the provided modifications should be supported by a specialized processor-in-memory architecture. Initial ideas for implementing such approach, are presented in this paper.

This paper is organized in five sections. Section two discusses variety of FFT/IFFT algorithms for fast calculation of the DFT/IDFT and compares their efficiency. Section three provides an overview of different FFT/IFFT hardware implementations, discussing the achievable speed up by introducing parallelism. Section four presents several techniques for efficient hardware implementation of the basic DFT/IDFT computations, as well as possible DFT/IDFT improvements of the multiplication and addition operations, which are essential for performing the summation of products in the DFT/IDFT. The proposed improvements should involve maximal parallelism, during the execution of the DFT/IDFT computations. The paper ends with a conclusion, stated in section five.

II. Software Optimization of Discrete Fourier Transform

The DFT/IDFT is the most important discrete transform, used to perform Fourier analysis in many practical applications. For example, it has a fundamental function for modulation and demodulation in OFDM systems [2]. This transformation deals with a finite discrete-time signal and a finite or discrete number of frequencies, so it can be implemented in computers by numerical algorithms or even -vector dedicated hardware.

Given n real or complex inputs $x_0,\ldots,x_{n-1}$, the DFT [2] [20], is defined as:

$$y_k = \sum_{0\le \ell < n} \chi_\ell \omega_n^{-k\ell},\ 0 \le k < n, \tag{1}$$

with $\omega_n = \exp(-2\pi i / n)$ and $i = \sqrt{-1}$. Stacking y_k and χ_ℓ into vectors $x = (x_0,\ldots,x_{n-1})^T$ and $y = (y_0,\ldots,y_{n-1})^T$ yields into the equivalent form of a matrix-vector product:

$$y = DFT_n x,\ DFT_n = [\omega_n^{kl}], 0 \le k, l > n \tag{2}$$

Straightforward computation of both DFT and convolution is a matrix-vector multiplication, given as $W \bullet \vec{g}$ and takes $O(N^2)$ operations, for N being a transformation size.

The breakthrough of the Cooley-Tukey algorithm's family derives from its capability of significantly cutting down DFT's $O(N^2)$ complexity to an order of $O(N\log_2 N)$, [13]. This advance in computational theory inspired and motivated a stream of researches targeting even further speed up and efficiency of performing DFT, and eventually a whole new class of algorithms was introduced, known as FFT algorithms. A common feature for most FFT algorithms is their order of complexity - $O(N\log_2 N)$.

A. Divide and Conquer Approach as a Basis for FFT Algorithms

The DFT usually arises as an approximation to the continuous Fourier transform, allowing functions sampling at discrete intervals in space or time. In order to make the DFT operation more practical, several FFT algorithms were proposed. The fundamental approach for all of them is to make use of the properties of the DFT operation itself, and thus reduce the computational cost of performing the DFT calculations. This is basically achieved by implementing the divide and conquer approach [21], which is a basis for most of the algorithms for effective computation of DFT.

We already stated that the discrete Fourier transform is a matrix product, whereas $x_0,\ldots,x_{N-1}$ is the vector of input samples, $x = (X_0,\ldots,X_{N-1})^T$ is the vector of transform values and W_N is the primitive Nth root of unity, so that $W_N = \exp(-2\pi i / N), i = \sqrt{-1}$ [20]. This product is given by the following equation:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N & W_N^2 & W_N^3 & \cdots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & W_N^6 & \cdots & W_N^{N(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \cdots & \cdots & W_N^{(N-1)(N-1)} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{3}$$

W_N is also referred to as the *twiddle factor* or *phase factor*. This value being a trigonometric function over discrete points around the 4 quadrants of a two dimensional plane, has symmetric and periodic properties [22]. Using these properties of the twiddle factor, unnecessary computations of the DFT can be easily eliminated.

The direct evaluation of the matrix-vector product requires N^2 complex multiplications and additions. In order to reduce this huge number of operations, a divide and conquer approach is implemented. The general idea of this methodology is to map the original problem into several sub-problems in such a way that the following inequality, [20], is assured:

$$\text{cost(subproblems)} + \text{cost(mapping)} < \text{cost(original problem)} \quad (4)$$

The real influence of this method is that, regularly, the division can be applied recursively to the sub-problems as well, thus leading to a reduction of the order of complexity.

The important point in (4) is that the divide and conquer scheme is consisted of two clear costs: the cost of the mapping (which can be zero when looking at the number of operations only) and the cost of the sub-problems. As a consequence, different types of divide and conquer methods make an effort to find balancing schemes between the mapping and the sub-problem costs [20]. As an example, the Cooley-Tukey radix-2 algorithm can be considered, where the sub-problems results being quite trivial (only sum and differences), although the mapping requires twiddle factors that lead to a large number of multiplications. On the contrary, in the prime factor algorithm, the mapping is done only by means of permutations (no arithmetic operation are required), while the small DFTs that appear as sub-problems indicate substantial costs since their lengths are co-prime numbers.

B. Families of FFT Algorithms

There are two core families of FFT algorithms: the Cooley-Tukey algorithms and the Prime Factor algorithms, [20]. These classes of algorithms differ in the way they translate the full FFT into smaller sub-transforms. Mostly two types of routines for Cooley-Tukey algorithms are used: mixed-radix (general-N) algorithms and radix-2 (power of 2) algorithms. All Radix algorithms are similar in structure, differing only in the core computation of the butterflies, [23]. Each type of algorithm can be further categorized according to additional features, such as whether it operates in-place or requires an additional scrape space, whether its output is in a sorted or scrambled order, and whether it uses decimation-in-time or -frequency iterations.

1) Cooley-Tukey Algorithms

The Cooley–Tukey set of algorithms comprises the most common fast Fourier transform (FFT) algorithms. They re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size $N = N_1N_2$ in terms of smaller DFTs of sizes N_1 and N_2, recursively, in order to reduce the computation time to $O(N\log_2 N)$ for highly-composite N (smooth numbers). Usually, either N_1 or N_2 is a small factor (not necessarily prime), called the radix (which can differ between stages of the recursion) [20]-[23]. If N_1 is the radix, the Cooley–Tukey algorithm is called decimation in time (DIT), whereas if N_2 is the radix, it is decimation in frequency (DIF).

A radix-2 DIT FFT is the simplest and most common form of the Cooley–Tukey algorithm, although highly optimized Cooley–Tukey implementations generally use some other forms of the algorithm. Radix-2 DIT decomposes a DFT of size N into two interleaved DFTs (hence the name "radix-2") of size N/2 with each recursive stage, eventually resulting in a combining stage containing only size-2 DFTs called "butterfly" [13], operations (so-called because of the shape of the data-flow diagrams, Fig. 1).

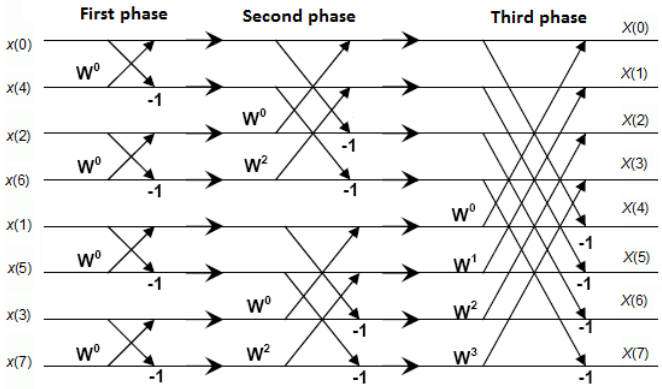

Fig. 1. Flow graph of 8-point DIT FFT radix-2 Cooley–Tukey algorithm.

Besides the radix-2 Cooley-Tukey algorithm, other implementations with radixes of 4 and 8 are also used. Actually, the value of the radix (2, 4, 8) indicates that the total number of points used for the transformation can be expressed as 2^x, 4^x or 8^x, accordingly [15]. Therefore, the C-T algorithm can execute parallel and independent butterfly operations with 2, 4 or 8 input/output values, in each of the algorithm phases (for radix x, the number of phases is $\log_x N$,).

Mixed-radix (also called split-radix) algorithms work by factorizing the data vector into smaller lengths. These can then be transformed by FFTs with small number of points, noted as small-N FFT [24]. Typical programs include FFTs for small prime factors, such as 2, 3 or 5, which are highly optimized. Actually, the idea of this algorithm is to use many multiplied small-N FFT modules and combine them in order to make longer transforms. If the small-N modules are supplemented by an $O(N^2)$ general-N module then an FFT of any length can be computed. Of course, any lengths which contain large prime factors would perform only as $O(N^2)$.

The well-known radix-2 Cooley–Tukey algorithm is a simplified version of the mixed-radix algorithm, realized by the use of FFT modules whose lengths are only power of two. Radix-2 algorithms have been the subject of much research into optimizing the FFT. Many of the most efficient radix-2 routines are based on the “split-radix” algorithm [15]. This is actually a hybrid which combines the best parts of both radix-2 ("power of 2") and radix-4 ("power of 4") algorithms, for computing distinctive partitions of the Fourier’s transformation.

2) Prime Factor Algorithms

The prime-factor algorithm (PFA), (also known as the Good–Thomas algorithm), is a FFT algorithm that re-expresses the DFT of a size $N = N_1N_2$ as a two-dimensional $N_1 \times N_2$ DFT, but only for the case where N_1 and N_2 are relatively prime. These smaller transforms of size N_1 and N_2 can then be evaluated by applying PFA recursively or by using some other FFT algorithm. Although this algorithm has a very simple indexing scheme, it only works in the case where all factors are co-prime, which makes it suitable as a

specialized algorithm for given lengths, [25]. PFA is also closely related to the nested Winograd FFT algorithm, where the latter performs the decomposed N_1 by N_2 transform via more sophisticated two-dimensional convolution techniques [26]. Winograd algorithm is requiring the least known number of multiplications among practical algorithms for moderate lengths DFTs.

Another less known algorithm is Fast Hartley Transform (FHT) [27]. This effective algorithm cannot be classified in formally presented families, since its core characteristic is what moves them apart. For the DFHT, the kernel is real, unlike the complex exponential kernel of the DFT. For complex data, each complex multiplication in the summation requires four real multiplications and two real additions using the DFT. For the DFHT, this computation involves only two real multiplications and one real addition.

It can be noticed that performing efficient and fast discrete Fourier transform, truly inspired researchers and burst many peoples' creativity. Sequentially, along with many ideas, came many experiments, evaluations and analysis. An effort was made to sum up the general conclusions in terms of complexity (meaning effectiveness) of the different types of FFT algorithms. Table 1 and Table 2 present the results.

From the results gathered in Table 1 and Table 2, an assumption can be made, that in general, split-radix algorithm achieves best performance. However, its irregular structure introduces some difficulties during the implementation [15]. Observing Table 1, another thing can be noticed, and that is the obvious decrement of the number of operations required for FFT, with each increment of algorithms' radix. Regarding the Prime Factor Algorithms, from Table 2 it can be concluded that Winogard is more efficient for smaller FFT sizes while for FFTs with input array sizes grater then 500 points, Prime Factor Algorithms present better results. Thus, for this family of algorithms a general winner or leading algorithm cannot be determined.

At this stage, we conclude FFT algorithms brief overview and efficiency evaluation, considering different manners of influencing DFT performances' only through algorithms, i.e. optimization in software. Nevertheless, these influences' impact depends on the characteristics of the platform the algorithms are executed on. Thus, this paper elaborates the FFT algorithms possible hardware implementations in section 3, as well.

C. FFT Parallelization and Optimization in Software

The emergence of multi-core processors also encouraged further research and optimization of FFT algorithms. The most popular of the previously discussed algorithms were rewritten introducing multi-threaded FFT programs. However, well known rule of thumb is that, the more the code is optimized, the harder it is to be realized in parallel. Most researches in this direction go with sequentially executing code until a parallel region is reached where multiple threads can be employed, [28]. This practice is truly more a need, since FFT algorithms' separate stages depend on each other's results.

An accent was also put on joint resources, like twiddle factors and bit reversal mapping which are commonly presented as lookup tables (LUT) [28][29]. But then again, memory may be saved, but usually the case is that memory accesses are far more expensive (in terms of speed of execution) then arithmetic operations, due to slower memories. The consequence of this type of implementation may not be the performance one hoped for. This particular fact was the starting point for the FFTW project, [30], where an effort was made for maximum utilization of computers' fast memory (cache and RAM), by introducing self-optimizing FFT algorithms on the behalf of specialized compiler, that adapt themselves in the form most suitable for particular architecture implementation.
Sdalksdf

III. Hardware Optimization of FFT Algorithms Implementations

Efficiency comparison on all previously discussed algorithms makes no real sense if they are not associated with their hardware application. By hardware, we mean certain architecture and particular platform. It was proven that the desired performance requirements will only be met if FFT algorithms are applied into suitable hardware set. Architectures vary in number of available processing units as well as in memory size and distribution. These two main features have inevitable impact on FFT algorithms efficiency, since they cover parts of FFT which proved to be the most intense – arithmetic operations and memory accesses.

A. Various Platforms for FFT Algorithms Hardware Implementations

Several architectures have been proposed [31][32], each of them trying to optimize the load of memory accesses and increase overall speed of FFT execution. These architectures effectiveness can be further evaluated by physical implementation. Today's cutting edge technology provides number of hardware platforms fit for deploying FFT algorithms. In this paper GPP, DSP as well as specialized hardware circuits – ASIC and FPGAs implementations are discussed.

TABLE I
Comparison of Multiplication and Addition Complexity of DFT and Cooley-Tukey Based FFT/IFFT Algorithms

	DFT			**C-T Radix-2**			**C-T Radix-4**			**C-T Radix-8**			**C-T Split Radix**		
Points	*Real Adds*	*Real Muls*	***Total***	*Real Adds*	*Real Muls*	***Total***	*Real Adds*	*Real Muls*	***Total***	*Real Adds*	*Real Muls*	***Total***	*Real Adds*	*Real Muls*	***Total***
16	992	1024	**2016**	152	24	**176**	148	20	**168**				148	20	**168**
32	4032	4096	**8128**	408	88	**496**							388	68	**456**
64	16256	16384	**32640**	1032	264	**1296**	976	208	**1184**	972	204	**1176**	964	196	**1160**
128	65280	65536	**130816**	2504	712	**3216**							2308	516	**2824**
256	261632	262144	**523776**	5896	1800	**7696**	5488	1392	**6880**				5380	1284	**6664**
512	1047552	1048576	**2096128**	13576	4360	**17936**				12420	3204	**15624**	12292	3076	**15368**
1024	4192256	4194304	**8386560**	30728	10248	**40976**	28336	7856	**36192**				27652	7172	**34824**

TABLE II
COMPARISON OF MULTIPLICATION AND ADDITION COMPLEXITY OF DFT AND PRIME FACTOR BASED FFT/IFFT ALGORITHMS

	DFT			**Prime Factor**			**Winograd**		
Points	*Real Adds*	*Real Muls*	***Total***	*Real Adds*	*Real Muls*	***Total***	*Real Adds*	*Real Muls*	***Total***
60	14280	14400	**28680**	888	200	**1088**	888	136	**1024**
240	229920	230400	**460320**	4812	1100	**5912**	5016	632	**5648**
504	1015056	1016064	**2031120**	13388	2524	**15912**	14540	1572	**16112**
1008	4062240	4064256	**8126496**	29548	5804	**35352**	34668	3548	**38216**

Architectural and design issues which are to be considered for diverse FFT hardware implementations are: precision of data, number of points for FFT size, and memory usage. Then performance evaluation is done according to power consumption, required circuit board area and desired circuit frequency. The value of these variables depends on the application which makes use of FFT i.e. employs it, and what's even more is that these same parameters have impact on the whole system (FFT algorithm + platform) performances.

1) General Purpose Processors

Many research efforts have been performed in this area [30][33] being just part of it. What is interesting to notice is that a greater part of the execution time was spent on load and store operations, compared to actual arithmetic computations [20]. This is a straightforward consequence of the GPP architecture, designed to satisfy many diverse applications' needs. Thus, from all implementations, this has proven to be the least effective one. Nevertheless researches utilizing these platforms continue, since many upper layer applications, using everyday technology (consisting GPPs) require FFT. Hence, a great advancement is made with the FFTWs' development, which provides an opportunity for one to choose an optimal algorithm for particular GPP, due to its adaptive features, by benchmarking FFTW library. From the gathered results it is obvious that for different FFT size, different platform is more convenient.

2) Digital Signal Processors

DSPs offer the best flexibility, but limited performance. These processors sturdily service multiply/accumulate based algorithms. Unluckily, this is not the case of any FFT algorithm (where sums of products have been changed to fewer but less regular computations) [20]. Still, DSPs now meet some of the FFT requirements, like modulo counters and bit-reversed addressing. If the modulo counter is general, it will help the implementation of all FFT algorithms, but it is often restricted to the Cooley-Tukey/SRFFT case only (modulo a power of 2), for which proficient timings are provided on nearly all available machines by manufacturers, at slightest for small to medium lengths. DSPs achieve low development cost compared to dedicated hardware solutions, although at the expense of medium performance and high power consumption.

3) Application Specific Integrated Circuit

Application Specific Integrated Circuits are famous for being the fastest platform for processing intense operations. Many hardware product vendors like Xilinx and Altera accepted this appropriateness of using ASIC chips for FFT implementations, and have developed high performance IP cores, from which designers benefit in reducing the time required for various product development. Compared to GPU and DSP they provide flexibility in terms of algorithms, meaning they are convenient not only for traditional Cooley-Tukey algorithms but also for PFA, [34]. Another reason to use them is their power efficiency requirements which are lowest of all.

4) Field Programmable Gateway Arrays (FPGA)

Fast Fourier transform has been playing an important role in digital signal processing and wireless communication systems, so the choice of FFT sizes is decided by different operation standards. It is anticipated to make the FFT size configurable according to the operation environment, since achieving a successful design means the system should be able to support different operating modes required by diverse applications with low power consumption requirement [35]. This is the reason why reconfigurable hardware has been paid more attention recently. FPGAs combine ASICs desired speed of operation with the flexibility provided from DSP, resulting in a perfect match for FFT implementations. Additionally, the capability of changing the source code according to current specific application makes this type of platform convincing winner for FFT implementation. Despite of all the upsides, a disadvantage must be mentioned, and that its power consumption.

Further, this paper presents another not so explored platform, but from our point of view, suitable for FFT implementation, and that is Intelligent RAM–IRAM.

5) Intelligent RAM

Intelligent RAM [18], is another merged DRAM-logic processor, designed at the Berkeley University of California by a small Patterson's team of students. This chip was designed to serve as multimedia processor on embedded devices. As a result of the design studies of the Berkeley's team research group, it was shown that most applicable architecture to multimedia processing is vector architecture, rather than MIND, VLIW, and other ILP organizations [36]. This is basically because of the vector processors' abilities to deliver high performance for data-parallel tasks execution, and to provide low energy consumption, simplicity of implementation and scalability with the modern CMOS technology.

The resulted IRAM chip is called Vector IRAM (VIRAM). VIRAM is processor that couples on-chip DRAM for high bandwidth with vector processing to express fine-grained data parallelism [18]. The VIRAM architecture (shown on Fig. 2) consists of MIPS GPP core, attached to a vector register file that is connected to an external I/O network and also to a 12MB DRAM memory organized in 8 banks.

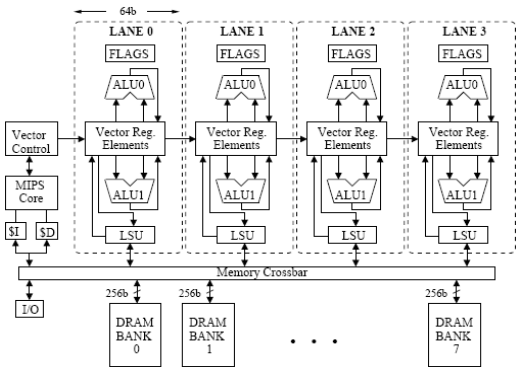

Fig. 2. Architecture of IRAM chip.

FFT computations are difficult to be realized with conventional GPP architectures, given that they require high memory bandwidth and strided memory accesses. Nevertheless, it was shown that the performance of computing floating-point FFTs on a VIRAM processor, [18], can be competitive with that of existing floating point DSPs and specialized fixed-point FFT chips.

From this section it can be concluded that there is no such thing as a universally optimal FFT/IFFT algorithm, architecture and implementation that is appropriate for all systems. Therefore, it is recommended to perform a search across the algorithm, architecture and implementation dimensions for each system.

B. Further FFT Speedup by Introducing Parallelization Techniques in Hardware

Traditional FFT hardware architectures include trade-offs among complexity, power consumption, die size, and other similar parameters. Still, these architectures don't have the scalability to encounter high speed processing requirements on FFT performance, for instance on emerging high data rate technologies in wireless communications, like OFDM. Progress in technology empowers development of architectures optimal for given class of algorithms, including FFT.

In general, there are three main techniques of improving overall FFT speed, all of them based on the Transpose algorithm, described in [37]:

1) *Increasing the order of the radix.*
   This approach improves both latency and throughput, but is expensive in terms of computational resource required for each processing unit.
2) *Cascading (pipelining) the processors.*
   Different processors operate over different stages. Cascading improves throughput, but not latency. Inter stage memory is necessary.
3) *Parallelizing the processors.*
   Parallelizing the processors, so that a single large FFT is divided into N smaller FFTs. Both latency and throughput are improved with this arrangement. Drawback is the data transfer between stages.

All these improvement methods require architecture modifications. Thus, in literature, commonly two architecture approaches are presented: memory-based FFT and pipeline-based FFT. Memory-based FFT uses only one butterfly and large memories for data storage. On the other hand, pipelined architecture uses many butterflies to improve the speed. There are several FFT hardware modules that have been proposed to implement both methods (memory-based design and pipelined design) in order to improve speed and minimize area. For example, [38] proposes six memory-based implementations, realized in FPGA. The authors give the name of RX2-B1, RX4-B1, RX2-B2, RX4-B2, RX2-B4, MXRX-B4 for these architectures, whereas the RX presents the used radix and Bi indicates that the architecture operates with i outputs in parallel. The techniques used for each of these architectures are based on memory sharing, conflict free memory addressing, in-place memory processing, continuous flow design, N-word memory size, fixed-point arithmetic and pre-computed twiddle factors stored in ROM. It was found that the fastest processors are the RX2-B4 and MXRX-B4 FFT models which can process four data samples per clock cycle.

Another direction in researching memory based architecture is towards distributed memory and processing units. This technique requires re-expressing the FFT algorithms to adapt them for parallel implementation. According to [21][39], where several algorithms were tested, it was proven that additional complexity is added to the system, as a consequence of the need for communication between different modules. Also, it was noticed that not all algorithms are suitable for parallelization, since only algorithms with regular structure (Cooley-Tukey Radix-2/4, Split Radix) are advisable, while the block complexity of Winograd algorithm makes it difficult and unsuited for parallel implementation.

Regarding the pipelined architectures, many works have presented optimizations to achieve high performance and low area consumption. The most famous architectures of pipeline implementations are multi-path delay commutator, radix-2 single path delay commutator and radix-2 single path delay feedback, [35]. The main difference between these architectures is in the number of inputs, outputs and butterflies used.

A different approach is applied by [40] proposing an array processing architecture for parallel FFT implementation. Key advantage of this architecture is the elimination of inter stage data transfer, so both, the system throughput and latency are improved. This idea is basically similar to IRAM, [18], which also includes vector processing units, capable to process the input data in parallel.

Although there have been several efficient designs, there is an inherent drawback related to the area (and consequently power) overhead. Another essential conclusion is the existing threshold, which delimits FFTs' input size after which this whole additional complication of FFT implementation system is rewarding, so this fact must also be taken into account while developing applications.

IV. HARDWARE OPTIMIZATION OF DISCRETE FOURIER TRANSFORM

The Discrete Fourier Transform is a universal instrument in science and engineering, including digital signal processing, communication, and high-performance computing. Applications employing it belong into the area of spectral analysis, image compression, interpolation, solving partial differential equations, and many other tasks,

[13]. Therefore, as a consequence from its extensive implementation, meeting its computational demands, is the high price applications employing DFT, have to pay.

As stated before, the Discrete Fourier Transform for an N-point sequence x(n) is given by (4), where X(k) and x(n) are complex numbers, n is the time index and k is the frequency index [1]. Actually, the DFT formula is expressed as a summation of complex numbers products.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{\frac{-j2\pi nk}{N}}, \; k = 0,1,...,N-1 \qquad (4)$$

By using this equation the DFT computation requires N^2 complex multiplications and N(N–1) complex additions, leading to a complexity of $O(N^2)$. Each complex multiplication requires four real multiplications and two real additions and each complex addition requires two real additions. Therefore a total of $4N^2$ real multiplications and N(4N-2) real additions are required [1]. Although the DFT employs the highest number of operations, comparing to various FFT implementations, all the operations are independent and can be executed in parallel. This is where we see a chance for maximal parallelization that will allow reaching better and faster results.

Generally, the FFT algorithms decrease the number of operations, but involve significant communication overhead, during the execution. This is a result of the divide and conquer approach which halts the system before proceeding to the next stage until all the output from the previous stage is generated [21]. For example, the execution of the Cooley-Tukey radix-x algorithm is performed in $\log_x N$ phases that must execute sequentially, although each phase employs parallel operations. The data dependences that appear between separate stages of the 8-Point Cooley-Tukey FFT radix-2 algorithm and DFT are shown in Fig. 3.

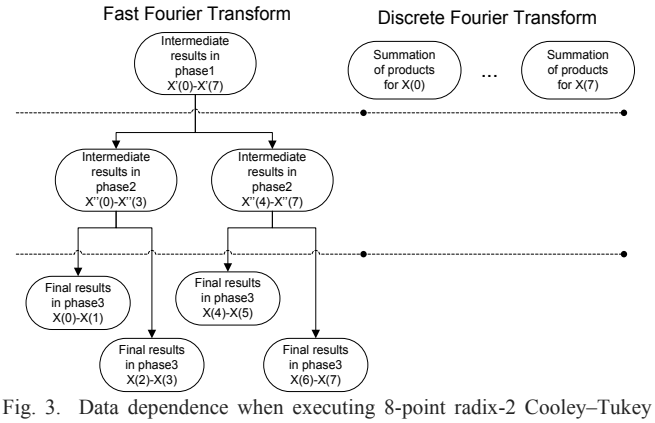

Fig. 3. Data dependence when executing 8-point radix-2 Cooley–Tukey FFT algorithm and DFT.

As can be seen on Fig. 3, the DFT is computed in only one phase. According to the fact that the DFT summations are computed simultaneously, it is of special interest to implement the DFT component on a hardware architecture that can support the provided parallelism (for example IRAM). Furthermore, the computation complexity of the DFT can be significantly reduced by utilizing some techniques that can simplify the multiplications of the inner products and/or parallelize the summations of different outputs.

A. Techniques for efficient hardware implementation of DFT

The strong interaction between the algorithm and its implementation indicates that the algorithm's execution efficiency depends on the hardware architecture and its abilities to support the provided parallelism of the algorithm. Considering the discussion given in the previous section, serious candidates for implementing the DFT are specific application integrated circuits or special purpose processors (similar to IRAM), implemented in FPGA platforms.

Most of the research done so far refers to efficient hardware implementations of FFT, by implementing various parallelization prospects and specialized components. In this section we are discussing several techniques (some of them already used for FFT hardware implementation) that can be applied when implementing the basic DFT computations in hardware. We suggest that the hardware architecture of the DFT module should be specially purposed to exploit the provided parallelism, when executing summation of products.

Assuming that each multiplication is consisted of several additions, it is obvious that the execution of multiplication operations is more complex than additions. Therefore, when implementing the products of the DFT summations many researchers have used specialized multipliers, purposed to improve the execution efficiency of multiply operations. Such approach is presented in [17], where the authors suggest the use of constant co-efficient multiplier. This multiplier utilizes small blocks of ROM that store 16 partial products relating to the fixed coefficient and then use a simple adder to combine these products. As a result, the co-efficient multiplier is less than one third of the size of full multipliers. Another multiplier that is also very efficient, (without using ROM memories) is the shift-and-add multiplier [8]. This multiplier decomposes the product in such a way that it requires only additions and multiplications or divisions by 2, which are realized in a computer by shifting bits left or right.

A different approach is presented in [41], where low power and area efficient adder and Vedic multiplier for computing FFT are presented. Generally, the speed of addition is limited by the time required for carry propagation through the adder, since the sum for each bit position in an elementary adder is generated after the previous bit position has been summed and a carry has propagated into the next position. Referring to [41] the carry select adder allows fastest execution of arithmetic operations by independently generating multiple carries and then selecting a carry to generate the final sum. Although this adder is not area efficient, its ability to decrease the propagation delay makes it popular for use. The combination of the carry select adder and Vedic multiplier provides fast execution and also reduction of FFT chip area and power [41]. The Vedic multiplier is based on Vedic mathematic principles, which employs a unique technique of calculations based on sixteen principles or word-formulae which are termed as Sutras. This mathematic is very suitable for digital signal processing [42].

Very efficient technique for calculation of DFT inner products is Distributed Arithmetic (DA) [43]. This method is used only if one of the vector operands is known,

allowing bit-serial computation that forms an inner product of a pair of vectors in a single direct step. The distributed arithmetic approach replaces the explicit multipliers by ROM look up tables and adders, thus providing considerable reduction in the power consumption. Although this technique appears to be slow because of its bit serial nature, it turns out that when the number of samples in each vector commensurate with the number of bits in each vector element, the DA is very fast. Another similar approach that utilizes look-up tables for pre-computing the products of the DFT summations is presented in [5]. It was shown that such DTF module can be applied to OFDM transmitter for optical communication, which is capable to achieve 101.5 Gb/s through fiber with 16QAM modulated sub carriers, by using two Xilinx Virtex 5 FPGA boards that work in parallel.

The speed of the DFT calculations can be also improved by utilizing the CORDIC (coordinate rotation digital computer) algorithm, which dynamically calculates the trigonometric functions, using processing elements to perform vector rotations [44]. The CORDIC component can also be used for polar to rectangular or rectangular to polar conversions, and as well for computing a vector magnitude. This algorithm operates in such a way that it executes vector rotations by arbitrary angles, using only shift and add operations, thus providing advantages in speed, accuracy, simplicity in design and other aspects of performance requirements.

B. Proposed improvement for efficient hardware implementation of DFT

The total processing time required for computing the DFT, depends on the multiplication time, which is generally far greater than the addition time. The multiplications and additions of the DFT are performed over complex numbers [7], which are usually represented in rectangular format:

$$(a+bj)\cdot(c+dj) = (ac-bd) + (ad+bc)j \quad (5)$$

$$(a+bj) + (c+dj) = (a+c) + (b+d)j \quad (6)$$

As given in equations (4) and (5) each multiplication of two complex numbers requires four multiplications of real numbers (imaginary and real component) and two additions of real numbers. On the other hand, each addition of two complex numbers requires two additions of real numbers. From here, it is obvious that multiplications are more complex than the additions.

There is an approach [13], which reduces the multiplication complexity to 3 multiplications and five additions. This is achieved by regrouping the multiplicands, as presented in equation (6):

$$(a+bj)\cdot(c+dj) = [(a+b)c - b(d+c)] + j[(a+b)c + a(d-c)] \quad (7)$$

Representing complex numbers in rectangular format is a common choice, basically because the arithmetic operations, such as: add, sub, mul and div can be easily implemented. On the other hand, having complex numbers represented in circular form (polar coordinates) can cause many difficulties, while performing adds/subs, but allows simpler implementation of muls/divs operations. Taking all this into account, in continuation we present a method for simplified implementation of the DFT/IDFT calculations.

The purpose of the DFT module is to perform the summation of products, given in equation (4), over input set of complex numbers. In our approach we consider that the DFT/IDFT module inputs and twiddle factor angles are provided in circular format, with polar coordinates (angle ϕ and amplitude r). All the twiddle factors magnitudes are 1s, while the angles are pre-computed and placed in a look up table. This way, the multiplications between complex numbers and twiddle factors are executed with only one add operation. Actually, each product term of the DFT summations is calculated as:

$$r \cdot e^{i\phi} \cdot e^{\frac{-i2\pi nk}{N}} = r \cdot e^{i(\phi - \frac{2\pi nk}{N})}, \; k = 0,1,...,N-1 \quad (8)$$

The calculation of the products in the DFT is not data dependent, so it can be executed in parallel. In order to compute the appropriate DFT output symbols, subsets of the calculated products, should be summed up. Given that the addition of circular format complex numbers is very complex, a conversion to rectangular format is made:

$$r \cdot e^{i(\phi - \frac{2\pi nk}{N})} = r(\cos(\phi - \frac{2\pi nk}{N}) + i\sin(\phi - \frac{2\pi nk}{N})) \quad (9)$$

The provided conversion involves cosine and sine operations and two multiplications of real numbers. The cos and sin operations can be calculated by utilizing look-up tables with pre-computed cosine and sine values, while the two multiplications can be performed by using only shift and add operations, as the authors of [14], suggested. The conversion is executed in such a way that firstly the *cos* and *sin* tables are looked-up and then the multiplications are performed. The conversions of different products are independent of each other and can be performed simultaneously.

After the real and imaginary components of each product are computed, next is to sum up the results, and produce the output complex numbers. The number of the outputs depends on the number of points, used in the DFT/IDFT module. Each output is generated by performing a sum over N products. In order to parallelize the summation of N inputs, we propose a parallel algorithm for that purpose.

The proposed algorithm performs parallel additions of N inputs with M bits, without having to wait for the carries generated during the additions. In fact, it allows simultaneous additions over each bit position of the numbers. Therefore, the summations of the i-th position (i=0,..,M-1) bits of each number are performed in parallel and each of the results is placed in a separate register, and shifted by i places. If there is a carry, the result register will have one or more 1s on some of the [i+1, i+M-1] positions. In that case the algorithm is performed again, and the appropriate additions are executed once more. Actually, the algorithm stops only when the generated output of each of the N registers consists only one 1 on the i-th position or all

0es. Other terminating condition of the algorithm can be the state when the results stored in all M registers, are completely the same as the results stored in all M registers, but in the previous iteration. In the final step of the algorithm, each of the registers holds the i-th cipher of the result.

The given algorithm was introduced to work with binary numbers, which sometimes can take a long time. In order to reduce the number of algorithms' iterations, one can work with numbers given in hexadecimal format. However this approach will complicate the design, requiring additional adders for performing the intermediate summations of N hexadecimal numbers.

The proposed modifications are made to allow parallel processing during the execution of the operations. However, the given improvements will only be effective if the hardware implementation of the DFT/IDFT module is consisted of many processing engines that can support the provided add and mul operations. The DFT/IDFT module will also include three look-up tables for holding the twiddle factor angles, and for performing the sine and cosine look-ups. Our opinion is that the processor-in-memory architecture can be very suitable solution for implementing the proposed DFT/IDFT improvements.

V. Conclusion

The never-ending aspiration for more efficient calculation of DFT, motivated by its truly widespread expansion in applications, could not disregard this opportunity. Although there are different variations of architectures of parallel systems, the number of researches made in this area, as well as the number of diverse tracks and ideas within it, is certainly surprising. Different combinations have been made, starting from FFT algorithms properties or the properties of parallel systems; architectures, further adjusting one to another in the development process.

All the work done in this area proves that by making the right combination of FFT algorithm, architecture and platform, desired performance results can be achieved. In general, all the efficient FFT implementations require specific hardware architecture that is tailored to support the computations involved in the FFT algorithm used.

In this paper we consider that most FFT algorithms include data dependent operations that limit the execution speed of the algorithm. Therefore, we suggest that the execution of the basic DFT/IDFT computations can provide execution speed-up, despite the fact that the DFT/IDFT involves more computations than many FFT algorithms. This is basically because the DFT/IDFT computations are characterized with high level of parallelism, so most of them can be executed simultaneously.

Considering that the basic DFT/IDFT computation is represented as summation of products, we propose possible improvements of the multiply and add operations over complex numbers. In our approach the multiplications involve only one add operation, since the operands are given in polar coordinate system. The additional cost that should be paid for this is the conversion to rectangular form, which involves two multiplications and calculation of sin and cos functions, using look-up tables. Furthermore, we propose an algorithm for performing parallel additions of N inputs that excludes the timing overhead caused by the addition of carry bits. The proposed algorithm involves high level of parallelism.

The proposed improvements are essential for performing the summation of products in the DFT/IDFT. It is expected that they should involve maximal parallelism, during the calculation of the DFT/IDFT outputs. However, their execution should be supported by a specialized processor architecture that would be able to exploit the provided parallelism. We believe that processor-in-memory architecture is a good solution for efficient implementation of the DFT/IDFT module with the proposed modifications.

Even though many researches dedicated their work on improving DFT calculating performance, we noticed a gap in examining pure DFT prospects for optimization and parallelization, where we recognize a great potential. For future continuing on this work we plan on implementing the proposed design expecting meaningful results.

Acknowledgement

This work was partially supported by the ERC Starting Independent Researcher Grant VISION (Contract n. 240555).

References

[1] S. W. Smith, “The Scientist and Engineer's Guide to Digital Signal Processing,”. California: California technical publishing, 1997.

[2] K. Adzha and B. Kadiran, “Design and implementation of OFDM transmitter and receiver on FPGA hardware,” M.S. thesis, Faculty of electrical engineering, Universiti teknologi Malaysia, Malaysia, 2005.

[3] M. Bernhard and J. Speidel, “Implementation of an IFFT for an optical OFDM transmitter with 12.1 Gbit/s,” ITG Symposium on Photonic Networks, Germany, 2010.

[4] F. Buchali, R. Dischler, A. Klekamp, M. Bernhard, and D. Efinger, “Realisation of a real-time 12.1 Gb/s optical OFDM transmitter and its application in a 10^9 Gb/s transmission system with coherent reception,”, 35th European Conference on Optical Communication, Germany, 2009.

[5] R. Schmogrow and M. Winter, et al, “101.5 Gbit/s real-time OFDM transmitter with 16QAM modulated subcarriers,” OSA/OFC/NFOEC 2011, vol. A247, pp. 529-551, April 1955.

[6] C. Toal and S. Sezer, et al, “A 1Gbps FPGA-based wireless baseband MIMO transceiver,” IEEE International SOC Conference (SOCC'12), pp. 202-207, September 2012.

[7] K. Maharatna, E. Grass, and U. Jagdhold, “A 64-point Fourier transform chip for high-speed wireless LAN application using OFDM,” IEEE Journal of Solid-State Circuits, vol. 39, March 2004.

[8] K. Maharatna, E. Grass, and U. Jagdhold, “A Low-power 64-point FFT/IFFT architecture for wireless broadband communication,” in Proc. 7th Int’l Conf. on Mobile Multimedia Communication (MoMuC), Tokyo, Japan, 2000.

[9] C. Lin, Y. Yu, and L. Van, “A Low-power 64-point FFT/IFFT design for IEEE 802.11a WLAN application,” IEEE International Symposium on Circuits and Systems, Greece, 2006.

[10] M. Bhardwaj, A. Gangwar, and D. Soni, “A review on OFDM: concept, scope & its applications,” IOSR Journal of Mechanical and Civil Engineering, Volume 1, Issue 1, pp. 07-11, 2012.

[11] L. Lit win and M. Pugel, “The principles of OFDM,” RF Signal Processing Journal, pp 30-48, 2001.

[12] T. Fung, “FPGA design and implementation of a memory based mixed-radix 4/2 FFT processor,” M.S. thesis, Dept. of Electrical Eng., Tatung University, Tatung, July 2008.

[13] R. E. Blahut, “Fast Algorithms for Signal Processing,” United Kingdom: Cambridge University Press, 2010.

[14] J. J. Fúster and K. S. Gugel, “Pipelined 64-point fast Fourier transform for programmable logic devices,” in Proc. International conference on advances in recent technologies in communication and computing, India, 2010.

[15] J. García1, J. A. Michell, G. Ruiz, and A.l M. Burón, “FPGA realization of a split radix FFT processor,” in Proc. SPIE, Microtechnologies for the New Millennium, vol. 6590, pp. 1-11, 2007.

[16] D. Ghosh, D. Debnath, and A. Chakrabarti, "FPGA based implementation of FFT processor using different architectures," International Journal of Advance Innovations, Thoughts & Ideas, 2012.
[17] M. Chandan, S. L. Pinjare, and C. Mohan Umapthy, "Optimised FFT design using constant co-efficient multiplier," International Journal of Emerging Technology and Advanced Engineering, 2012.
[18] R. Thomas, "An architectural performance study of the fast Fourier transform on vector IRAM," Berkeley University, California, Tech. Rep, 2000.
[19] Y. Ouerhani, M. Jridi, and A. Alfalou, "Implementation techniques of high-order FFT into low-cost FPGA", in Proc. IEEE 54th International Midwest Symposium on Circuits and Systems, Korea, 2011.
[20] P. Duhamel and M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," Signal Processing Journal, vol. 19, 1990.
[21] S. Meiyappan, "Implementation and performance evaluation of parallel FFT algorithms," School of Computing. National University of Singapore, Singapore.
[22] S. K. Palaniappan, "Design and implementation of radix-4 fast Fourier transform in ASIC chip with 0.18 μm standard CMOS technology," M.S. thesis, Malaysia, 2008.
[23] M. Soni and P. Kunthe, "A General comparison of FFT algorithms," Pioneer Journal Of IT & Management, 2011.
[24] B. Gough, "FFT algorithms," Tutorial paper, 1997
[25] K. M. Pavan Kumar and P. Jain, et al, "FFT algorithms: a survey," The International Journal Of Engineering And Science, India, 2013.
[26] S. Winograd, "On computing the discrete Fourier transform," Proceedings of the National Academy of Sciences of the United States of America, 1975.
[27] F. Piccinin, "The fast Hartley transform as an alternative to the fast Fourier transform," Australia, Technical Memorandum, SRL-0006-TM, 1988.
[28] R. Vincke and S. V. Landschoot, et al., "Calculating fast Fourier transform by using parallel software design patterns," Tech. Report, CW 627, October 2012.
[29] A. Cortés, I. Vélez, M. Turrillas, and J. F. Sevillano, "Fast Fourier transform processors: implementing FFT and IFFT cores for OFDM communication systems," Fourier Transform - Signal Processing, 2012.
[30] M. Frigo and S. G. Johnson, "The fastest Fourier transform in the west," Tech. Report, MIT-LCS-TR-728, 1997.
[31] Bevan M. Baas, "An approach to low-power, high-performance, fast Fourier transform processor design," PhD Thesis, Stanford University, USA, 1999.
[32] Y. Zhao and T. Ahmet, et al., "Architectural evaluation of flexible digital signal processing for wireless receivers," Signals, Systems and Computers, 2000.
[33] A. Ganapathiraju, J. Hamaker, J. Picone, and A. Skjellum, "Contemporary view of FFT algorithms," Mississippi State University.
[34] J. F. Herron, "Design and development of a high-speed Winograd fast Fourier transform processor board," M.S. thesis, Air University, USA, 1992.
[35] C. Li, "Design and implementation of variable-length fast Fourier transform processor," M.S. thesis, Tatung University, 2006.
[36] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," In proc. of the 35th International Symposium on Microarchitecture, Instabul, Turkey, November 2002.
[37] T. Lippert, K. Schilling, et al., "Transpose algorithm for FFT on APE/Quadrics," In Proc. of HPCN Europe, pp. 439-448, 1998.
[38] H.G. Yeh and G. Truong, "Speed and area analysis of memory based FFT processors in a FPGA," Wireless Telecommunications Symposium, pp. 1-6, 2007.
[39] E. Brachos, "Parallel FFT libraries," M.S. thesis, Edinburgh University, 2011.
[40] S. Johnsson and D. Cohen, "Computational arrays for the discrete Fourier transform," In Proc. 22nd Computer Society International Conference, 1981.
[41] C.A. Silambarasan and L. Vanitha, "Design and implementation of low power and area efficient adder and Vedic multiplier for FFT," International Journal of Communications and Engineering, Vol. 1, 2012.
[42] S. S. Kerur and Prakash Narchi, et al., "Implementation of Vedic multiplier for digital signal processing," International Journal of Computer Applications, 2011.
[43] S. A. White, "Applications of distributed arithmetic to digital signal processing: a tutorial review," IEEE ASSP Magazine, 1989.
[44] T. Sung, H. Hsin, and L. Ko, "Reconfigurable VLSI architecture for FFT processor," WSEAS Transactions on Circuits and Systems, 2009.