Report on

Analysis and Characterization of Fast Fourier Transform Algorithms

Aravind Ganapathiraju
Institute for Signal and Information Processing
Department of Electrical and Computer Engineering
Mississippi State University
Box 9571
434 Simrall, Hardy Rd.
Mississippi State, Mississippi 39762
Tel: 601-325-8335
Fax: 601-325-3149
email: ganapath@isip.msstate.edu

TABLE OF CONTENTS

1. ABSTRACT

A large number of Fast Fourier Transform (FFT) algorithms have been developed over the years. Among these, the most promising are the Radix-2, Radix-4, Split-Radix, Fast Hartley Transform (FHT), Quick Fourier Transform (QFT), and the Decimation-in-Time-Frequency (DITF) algorithms. In past benchmarking efforts, only the computation speed or number of mathematical operations have been used for assessing the efficiency of algorithms. Most of these benchmarks have been for special purpose CPUs like DSPs. With the rapid development of computer technology over the past few years, more CPU cycles per second are available now than before. This has allowed the use of these general purpose CPUs for many signal processing applications, of which the FFT is an integral part. We therefore see a need for a rigorous benchmarking of FFT algorithms on general purpose computers. In this report, we present a rigorous analysis of the aforementioned algorithms that includes the number of mathematical operations, computation time, memory requirements, and compiler effects.

2. INTRODUCTION

A large number of Fast Fourier Transform (FFT) algorithms have been developed over the years. The first major breakthrough was the Cooley-Tukey algorithm [1] developed in the mid-sixties which resulted in a flurry of activity on FFTs. It reduced the complexity of the algorithm from $O(N^2)$ to O(NlogN) which was tremendous savings of computer power of those days. Algorithms which followed have achieved this complexity reduction to varying degrees. The Cooley-Tukey algorithm was a Radix-2 algorithm. The next few radix algorithms developed were the Radix-3, Radix-4, and the Mixed Radix algorithm [8,9,10]. Further research led to the Fast Hartley Transform (FHT) [2,3,4] and the Split Radix (SRFFT) [5,9,10] algorithm. Recently, two new algorithms have also emerged: the Quick Fourier Transform (QFT) [6] and the Decimation-in-time-frequency (DITF) [7].

While there has been extensive research on the theoretical efficiency of these algorithms, there has been little research to-date comparing algorithms on practical terms. Efficiency is intricately related to how an algorithm can be implemented on a given architecture. There are different dimensions to this evaluation. The important issues to be considered in comparisons are the computation speed, memory, algorithm complexity, machine architecture and compiler design. We have therefore embarked on this work of providing a comprehensive benchmarking of a few commonly used algorithms.

One of the main tasks involved in this process of benchmarking is collecting a standard set of implementations of the algorithms we choose, Radix-2 (RAD2), decimation in frequency (DIF), Radix-4 (RAD4) DIF, SRFFT, FHT, QFT and DITF. The choice of these algorithms was influenced by the idea of comparing algorithms which vary significantly in their complexity as well as their history. The QFT and the DITF being the latest and the Radix-2 the oldest. This give us an opportunity to gauge the progress in this technology over 25 years. The FHT implementation was picked off world-wide-web(WWW) and is based on the algorithm developed by Hsieh S. Hou in 1987. The QFT implementation was provided by the inventors from Rice University. The remaining algorithms were implemented by us.

These algorithms were then put into a common framework. We chose to implement the software in C++ given the nature of the problem we had, where the algorithms were solving the same problem, namely FFT computation. The object oriented support provided by C++ was therefore our natural choice. This followed by deciding on the criteria for benchmarking. Since the main objective of our work was to benchmark on general purpose computers, we choose computation speed, memory usage, mathematical operations and compiler issues. We incorporated various features into our code to provide the required data efficiently. For timing measurements we found using the median value of multiple iterations was more robust than the average value. For the mathematical computations, we decided to use floating point operations, integer operations, and register shifts. Once the code was in place, we decided on the architectures to benchmark the algorithms. These varied from the commonly used processors like Sparc20, to the more recent fast processors like Sun UltraSparc, Pentium Pro and DEC Alpha. We choose to use GNU's **GCC** and **MS Visual C++** as our compilers of comparison.

In this report we provide the results of this benchmarking process on the 6 different algorithms. A theoretical analysis of the algorithms and the results is also provided. These results will help correlate the numbers predicted by theory, for the various criteria, to the practical values achieved. We have structured the code such that it forms a starting point for an algorithm which will provide the user with the optimal algorithm, to compute the FFT, given the resources in hand, like the CPU speed, on chip memory and compiler.The code generated from this work is available for public from our website at *www.isip.msstate.edu.*

3. ALGORITHM DESCRIPTION

Most of the FFT algorithms that have been developed, starting with the Cooley-Tukey algorithm, have used the symmetry properties of the complex exponential in the definition of the Discrete Fourier Transform. Most algorithms divide the problem in similar sub-problems and achieve a reduction in computational complexity. All radix algorithms are similar in structure differing only in the core computation of the butterflies. The FHT is different from the other algorithms in that it uses a real kernel unlike the other algorithms which use the complex exponential kernel. The QFT postpones the complex arithmetic to the last stage in the computation cycle by computing the Discrete Cosine Transform(DCT) and the Discrete Sine Transform(DST). The DITF algorithm uses the Decimation-in-TIme (DIT) framework and the Decimation-in-Frequency(DIF) framework for parts of the computation to achieve reduction in number of computations.

3.1. Radix-2 DIF Algorithm

The DIF radix-2 algorithm is obtained by using the divide-and conquer approach to the DFT problem. To start with we split the DFT formula into two summations, one of which involves the sum over the first N/2 data points and the other over the next N/2 data points. Thus we obtain the following formulae.

$$X(k) = \sum_{n=0}^{N/2-1} x(n) \cdot W_N^{kn} + \sum_{n=N/2}^{N-1} x(n) \cdot W_N^{kn} \qquad (1)$$

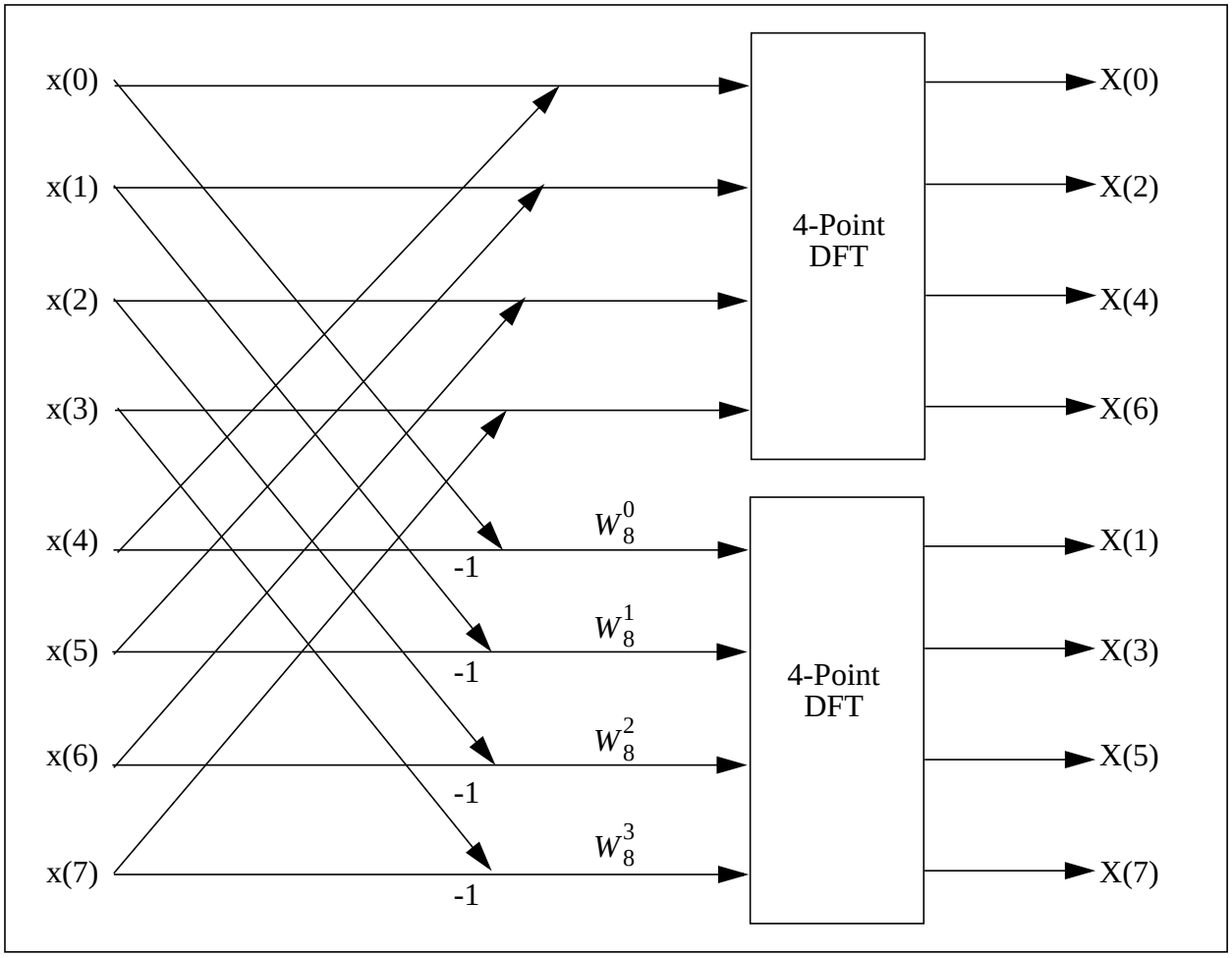

Figure 1. 8-point RAD2 computation

Since $W_N^k = e^{-(j2\pi k)/N}$, $W_N^{k(N/2)} = (-1)^k$, the above equation can be written as,

$$X(k) = \sum_{n=0}^{N/2-1} \left(x(n) + (-1)^k \cdot x\left(n + \frac{N}{2}\right)\right) \cdot W_N^{kn} \tag{2}$$

Considering the even and odd numbered frequency samples separately gives us:

$$X(2k) = \sum_{n=0}^{N/2-1} \left(x(n) + x\left(n + \frac{N}{2}\right)\right) \cdot W_{N/2}^{kn} \tag{3}$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} \left\{ \left(x(n) - x\left(n + \frac{N}{2}\right)\right) \cdot W_{N/2}^{kn} \right\} \cdot W_{N/2}^{kn} \tag{4}$$

These two equations can be represented in a block form as in Figure 1. for an 8 point FFT. The same computational procedure can be repeated through decimation of the N/2 point DFTs X(2k) and X(2k+1). The entire process involves $\upsilon = \log_2 N$ stages with each stage involving N/2 butterflies of form shown in Figure 2. Thus the radix-2 algorithm involves $N/2 \cdot \log_2 N$ complex multiplications and $N \cdot \log_2 N$ complex additions. We observe that the output of the whole process is out-of-order and requires a bit-reversal operation to get the frequency samples in the correct order.

3.2. Radix-4 Algorithm

The Radix-4 algorithm is very similar to the Radix-2 algorithm in concept. Instead of dividing the DFT computation problem into halves as in the Radix-2, it divides into quarters. Thus we split the N-point input sequence into four subsequences, $x(4n)$, $x(4n+1)$, $x(4n+2)$,and $x(4n+3)$. where n = 0, 1,..., $N/4 - 1$. Applying the above concept to the DFT computation we get,

$$X(k) = \sum_{n=0}^{N/4-1} x(n) \cdot W_N^{kn} + \sum_{n=N/4}^{N/2-1} x(n) \cdot W_N^{kn} + \sum_{n=N/2}^{3N/4-1} x(n) \cdot W_N^{kn} + \sum_{n=3N/4}^{N-1} x(n) \cdot W_N^{kn} \quad (5)$$

The Radix-4 butterfly is what gives the radix-4 algorithm savings in computations. The matrix formulation of the butterfly is as follows:

$$\begin{bmatrix} X(0,q) \\ X(1,q) \\ X(2,q) \\ X(3,q) \end{bmatrix} = \begin{bmatrix} W_N^0 & W_N^0 & W_N^0 & W_N^0 \\ W_N^q & -jW_N^q & -W_N^q & jW_N^q \\ W_N^{2q} & -W_N^{2q} & W_N^{2q} & -W_N^{2q} \\ W_N^{3q} & jW_N^{3q} & -W_N^{3q} & -jW_N^{3q} \end{bmatrix} \cdot \begin{bmatrix} F(0,q) \\ F(1,q) \\ F(2,q) \\ F(3,q) \end{bmatrix} \quad (6)$$

where,

$$F(l,q) = \sum_{m=0}^{N/4-1} x(l,m) \cdot W_{N/4}^{mq} \tag{7}$$

$$X(p,q) = X\left(\frac{N}{4} \cdot p + q\right) \tag{8}$$

$$x(l,m) = x(4m+l) \text{ l = 0, 1, 2, 3, q = 0, 1, 2,... } N/4 - 1 \text{ . and p = 0, 1, 2, 3} \tag{9}$$

Figure 2 shows the computation of a radix-4 butterfly. The decimation process is similar to the radix-2 algorithm. Using the above butterfly for the computation of the radix-4 FFT, uses $\upsilon = \log_4 N$ stages, where each stage has $\frac{N}{4}$ butterflies. Each butterfly involves 3 complex multiplications and 12 complex additions. point FFT using the radix-4 algorithm. Thus the overall computational burden can be quantified as $(3N)/8 \cdot \log_2 N$ complex multiplications and $(3N)/2 \cdot \log_2 N$ complex additions. This is a 25% savings in additions over the radix-2 algorithm. This however is the theoretical quantification only. The actual figure as will be shown in a later section is $4.25 \cdot N \cdot \log_2 N$ floating point operations which is 15% less than the corresponding value for the radix-2 algorithm.

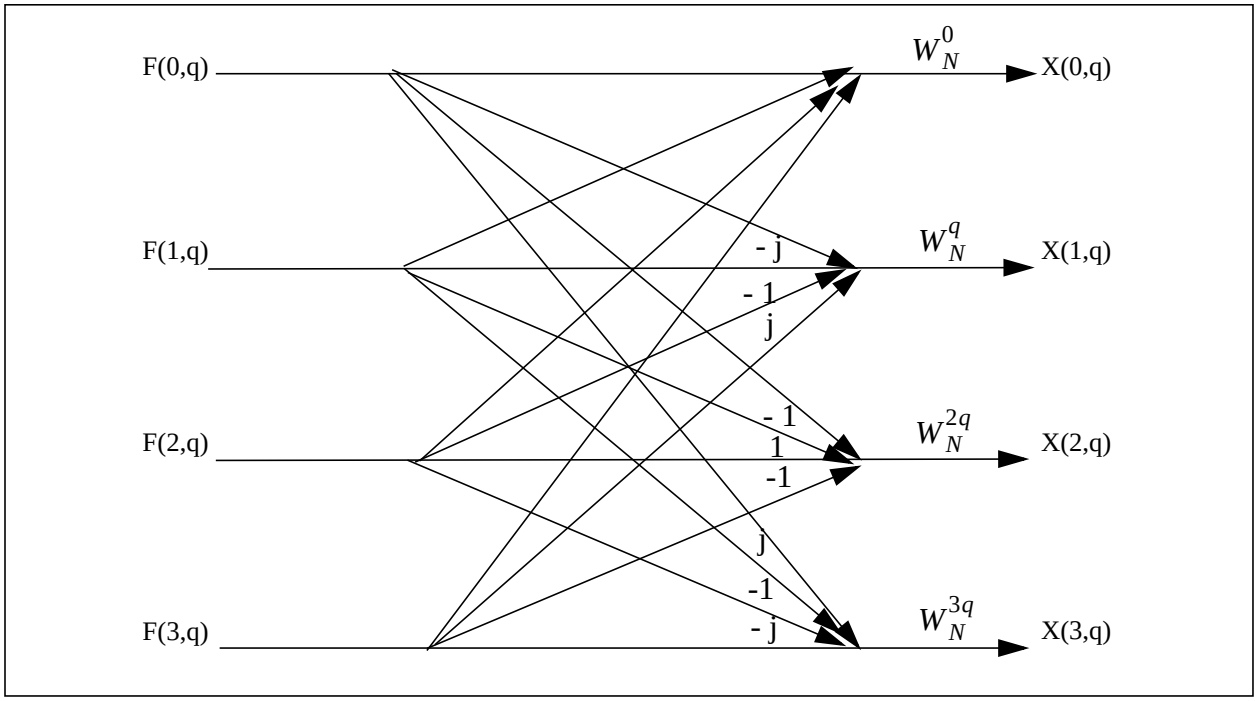

Figure 2. Radix-4 Butterfly involving 3 complex multiplications and 12 complex additions.

3.3. Split-Radix Algorithm

Standard radix-2 algorithms are based on the fast synthesis of two half-length DFTs. Radix-4 algorithms are based on the fast synthesis of four quarter-length DFTs. The Split-Radix algorithms is based on the synthesis one half-length DFT together with two quarter-length DFTs. This works because, if we notice the radix-2 computations, the even numbered points can be computed independent of the odd numbered points. The split-radix algorithm uses the radix-4 algorithm to compute the odd numbered points.

Thus, the N-point DFT is decomposed into one $N/2$ point DFT and two $N/4$ point DFTs. The decomposition is shown in the following equations.

$$X(2k) = \sum_{n=0}^{N/2-1} \left(x(n) + x\left(n + \frac{N}{2}\right)\right) \cdot W^{4kn} \tag{10}$$

$$X(4k+3) = \sum_{n=0}^{N/4-1} [g(n) + jf(n)]W^{3n} \cdot W^{4nk} \tag{11}$$

$$X(4k+1) = \sum_{n=0}^{N/4-1} [g(n) - jf(n)]W^{n} \cdot W^{4nk} \tag{12}$$

where,

$$g(n) = x(n) - x\left(n + \frac{N}{2}\right) \quad \text{and} \quad f(n) = x\left(n + \frac{N}{4}\right) - x\left(n + \frac{3N}{4}\right) \tag{13}$$

The N-point DFT is obtained by successive use of these decompositions up to the last stage. Figure 4 illustrates the computational breakup of a 32-point FFT. Here we treat the computational process as a Cooley-Tukey algorithm with the unnecessary intermediate DFT computations eliminated. It can be shown that each computation of length L DFT requires 6L floating point operations. This can be verified from the split-radix butterfly in Figure 3. The computational process can also be viewed as the DFT computations involved in each stage of the FFT. Figure 4 depicts this idea. For each q, the associated line segments indicate which L-point DFTs are computed where, $L = 2^q$. Thus at step $q = 2$, five 4-point DFTs are computed. Note that the data is in the bit reversed order to start with. After some math, we can show that, for the split-radix algorithm, approximately $4N \cdot \log_2 N$ computations are required as compared to $4.25N \cdot \log_2 N$ for radix-4 and $5N \cdot \log_2 N$ for radix-2 algorithms. This measurement is obtained by looking at the software implementation of the core butterfly of this algorithm.

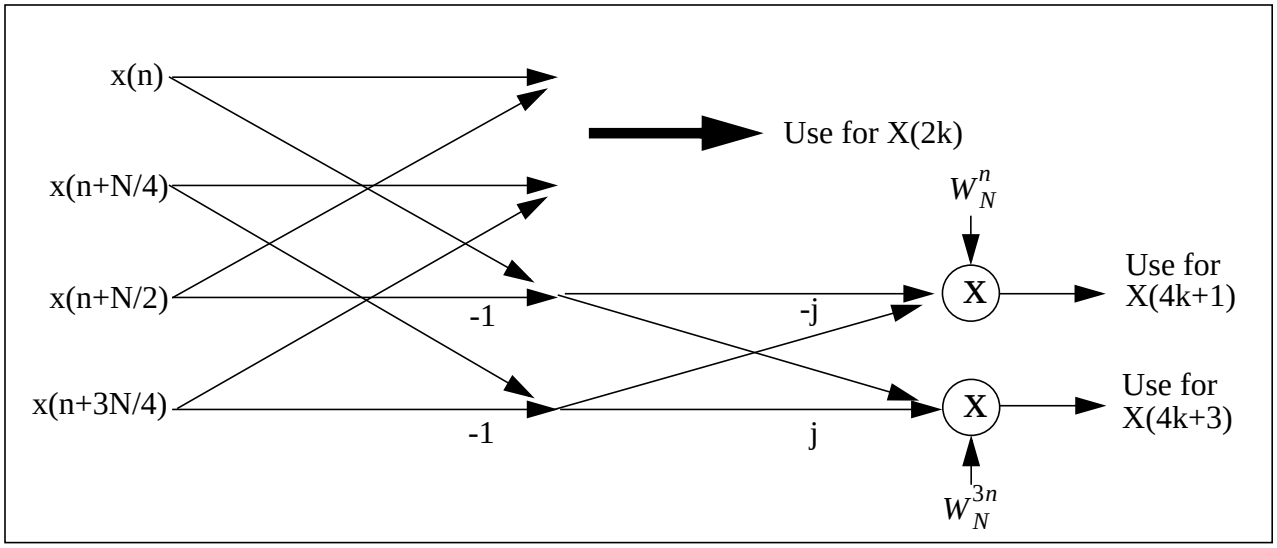

Figure 3. Split-Radix Butterfly

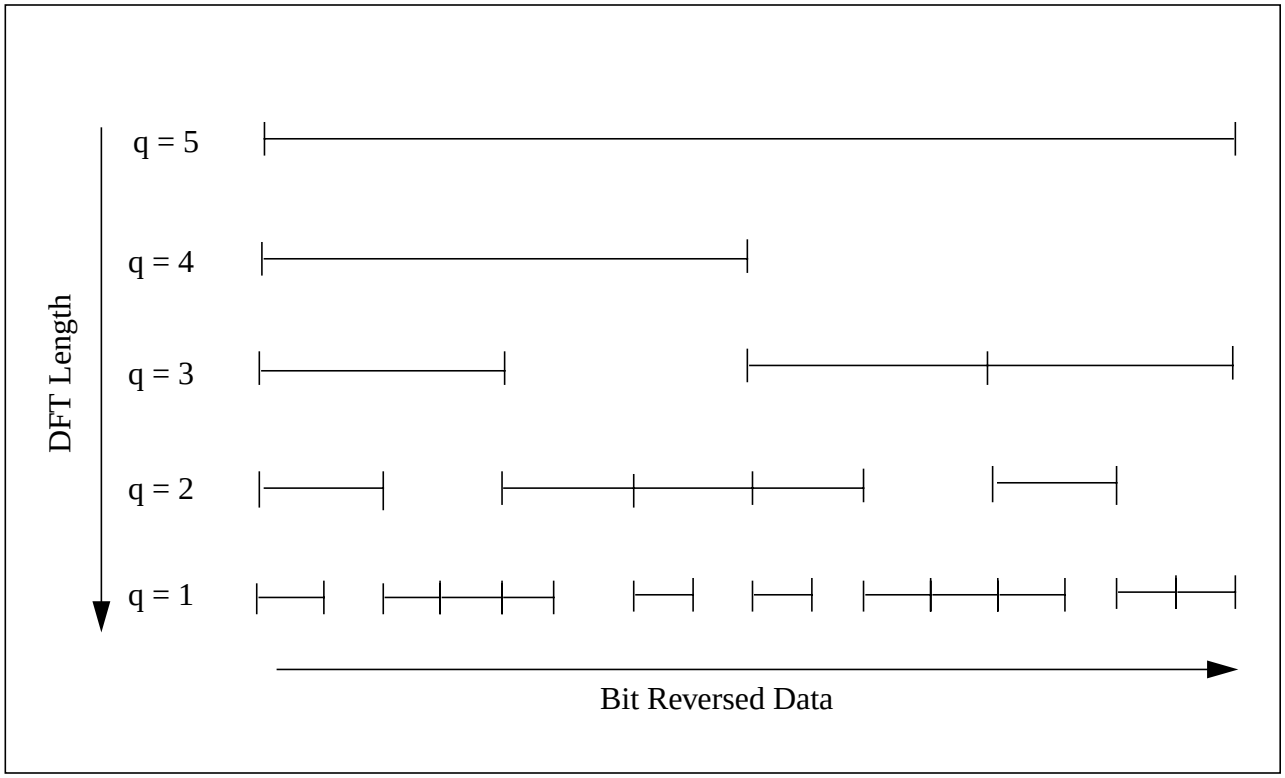

Figure 4. Split-Radix Computational Framework

3.4. Fast Hartley Transform

The Fast Hartley Transform evolved out of the same strategies used for the radix-2 Cooley-Tukey algorithm. The main difference between the other algorithms discussed here and the FHT is the core kernel which is real unlike the complex exponential of the DFT. Thus the FHT is intuitively simpler and faster as the number of computations reduces drastically when we replace all complex computations by real computations. Similar to the other recursive radix algorithms, the next higher order FHT is obtained by combining two identical preceding lower order FHTs.

The Discrete Hartley transform(DHT) of a sequence $x(n)$ of length N is defined in terms of the real kernel as:

$$X(k) = \frac{1}{\sqrt{N}} \cdot \sum_{n=0}^{N-1} x(n) \cdot [\cos((2\pi kn)/N) + \sin((2\pi kn)/N)] \tag{14}$$

A simple recursive fast algorithm can be defined for the Hartley Transform defined above by looking at the matrix formulation of some of the lower order DHTs. By plugging in the numbers in the above equation for $N = 2$, we get the following matrix representation.

$$\begin{bmatrix} X(0) \\ X(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \end{bmatrix} \tag{15}$$

Following a similar procedure for $N = 4$, we get the following matrix formulation.

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} \tag{16}$$

The above matrix can be arranged to:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(2) \\ x(1) \\ x(3) \end{bmatrix} \tag{17}$$

Comparing this matrix product with the matrix of $N = 2$, we note this matrix is repeatedly used in the matrix for $N = 4$ to compute the DHT. Thus a DHT of order 4 can be computed from a DHT of order 2. It can be shown that this is true for any N, which is a power of 2. Following the same principles, the computational process for $N = 8$ can be expressed as in Figure 5. In the figure the 4-point DHT computation is complete at stage 2 where the multiplication with $\alpha = 1/(\sqrt{2})$, is performed. The 4-point DHT computation is complete at stage 1 and this computation is done by using the simple 2-point DHT result. Fortunately, there is a straightforward relationship between the Hartley coefficients and the DFT coefficients. The properties of the real kernel, allow us to partition the DHT matrix into 4 quadrants. For the even numbered rows, the elements in the left half are equal to the corresponding ones in the right half. For the odd numbered rows, they are in the opposite sign. Likewise, for even numbered columns, the elements in the upper half are equal to the corresponding ones in the lower half. For the odd numbered ones they are in opposite sign. It can also be proved that the upper left quadrant of an NXN DHT matrix is the same as the N/2 X N/2 DHT matrix. Thus we see a simple recursive structure for the computation of the DHT. This structure follows very closely the ideas of the

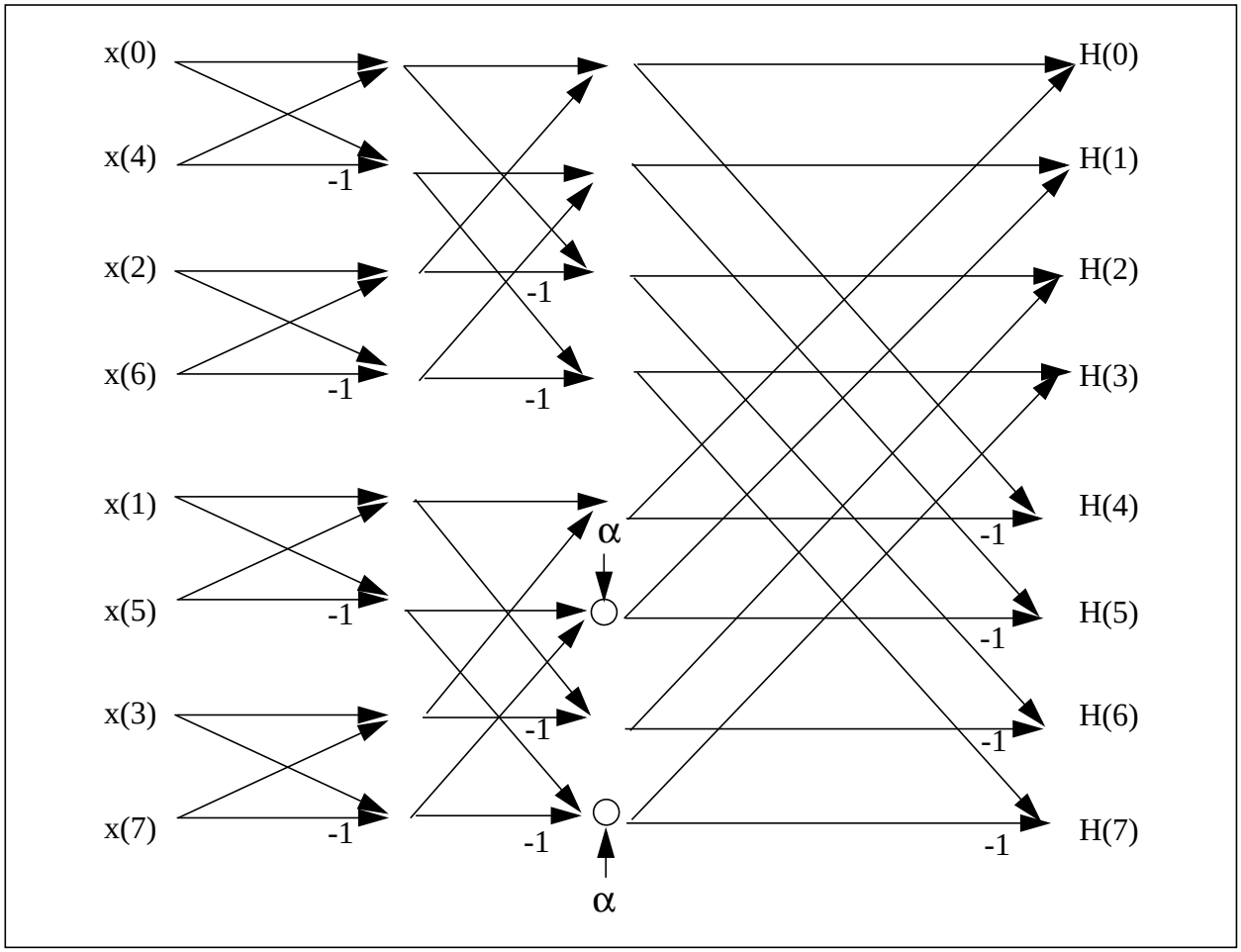

Figure 5. Butterfly diagram illustrating the computation of an 8-point FHT

Cooley-Tukey algorithm for the FFT. Also note that the output of this butterfly computation are the DHT coefficients. We formalize the aforementioned ideas in the following equations.

$$T(N) = \begin{bmatrix} Q\left(\frac{N}{2}\right) & Q\left(\frac{N}{2}\right) \\ S\left(\frac{N}{2}\right) & -S\left(\frac{N}{2}\right) \end{bmatrix} \tag{18}$$

where $T(N)$ is the Nth order DHT matrix and $Q(N), S(N)$ are square matrices of order N/2. The output sequence can be expressed in terms of x_p the first half of the input data and x_r the later half as follows:

$$\begin{bmatrix} \hat{y}_e \\ \hat{y}_o \end{bmatrix} = \begin{bmatrix} \hat{T}\left(\frac{N}{2}\right) & \hat{T}\left(\frac{N}{2}\right) \\ \hat{S}\left(\frac{N}{2}\right) & -\hat{S}\left(\frac{N}{2}\right) \end{bmatrix} \cdot \begin{bmatrix} x_p \\ x_r \end{bmatrix} \tag{19}$$

where the elements with *hats* the original elements in a rearranged order. $\hat{S}$ can be computed from $\hat{T}$ using the following equation.

$$\hat{S}\left(\frac{N}{2}\right) = \hat{T}\left(\frac{N}{2}\right) \cdot K\left(\frac{N}{2}\right) \tag{20}$$

where $K\left(\frac{N}{2}\right)$ is defined as,

$$K\left(\frac{N}{2}\right) = Diag(\cos\phi_k) + Diag(\sin\phi_k) \cdot P, \phi_k = \frac{2\pi k}{N} \tag{21}$$

P is the permutation matrix. Equation 19 forms the basis for the recursive implementation of the FHT.

$$Re(DFT(k)) = \frac{DHT(k) + DHT(N-k)}{2} \tag{22}$$

$$Im(DFT(k)) = \frac{DHT(k) - DHT(N-k)}{2} \tag{23}$$

Equations (22) and (23) give the simple relationship between the DFT coefficients and the DHT coefficients. Note that this conversion will not not be needed in some applications where the real and the imaginary components of the DFT are not required explicitly. This is especially true in applications that perform convolution or that need the power spectral density.

3.5. Quick Fourier Transform

In most of the radix based algorithms we have seen that the periodic properties of the cosine and sine functions have been exploited to reduce the computational complexity of the algorithms. In the Quick fourier Transform (QFT) the symmetry properties of these functions are used to derive an efficient algorithm.

$$X_{DCT}(k) = \sum_{n=0}^{N} x(n) \cos \frac{\pi n k}{N}, \tag{24}$$

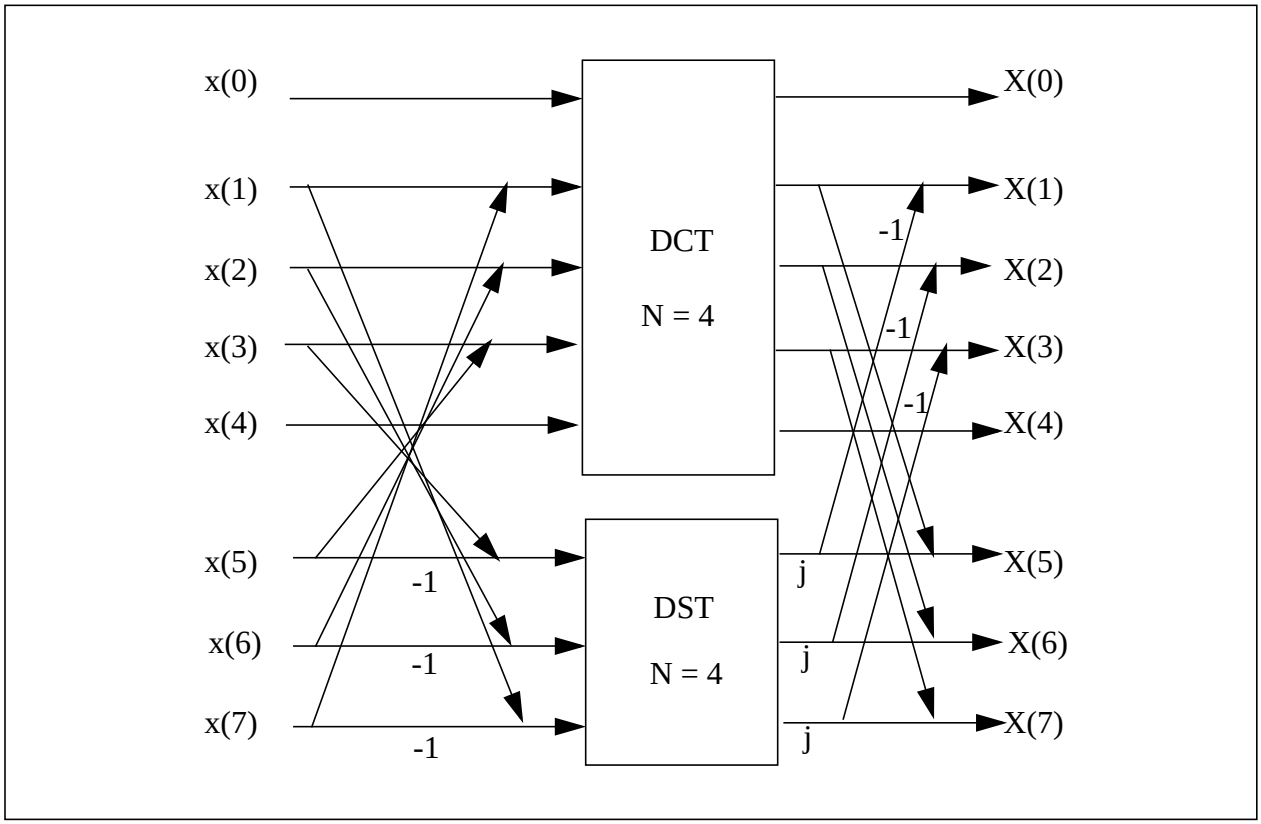

Figure 6. 8-point QFT computation involving breakup into 4-point DCT and DST computations

Equation (24) is the definition of an $N + 1$ point discrete cosine transform(DCT). An $N - 1$ point discrete sine transform (DST) also has a similar definition given by Equation (25).

$$X_{DST}(k) = \sum_{n=1}^{N-1} x(n) \sin \frac{\pi n k}{N}, \text{ k = 1,..., N-1} \quad (25)$$

It is interesting to note the following symmetry relations of the sine and the cosine functions:

$$\cos\left(\frac{2\pi(N-n)k}{N}\right) = \cos\left(\frac{2\pi n k}{N}\right) \quad (26)$$

and,

$$\sin\left(\frac{2\pi(N-n)k}{N}\right) = -\sin\left(\frac{2\pi n k}{N}\right) \quad (27)$$

We then divide N point input sequence into its even and odd parts as:

$$x_e(0) = x(0), x_e(N/2) = x(N/2) \quad (28)$$

$$x_e(k) = x(k) + x(N-k), \text{ k = 1,....,N/2-1} \quad (29)$$

$$x_o(k) = x(k) - x(N-k), \text{ k = 1,....,N/2-1} \quad (30)$$

Using equations 16-20, we can define the N-point DFT as:

$$X(k) = X_{DCT}(k) - jX_{DST}(k), \text{ k = 1, 2,..., N/2 -1} \quad (31)$$

$$X(N-k) = X_{DCT}(k) + jX_{DST}(k), \text{ k=1, 2,..., N/2 -1} \quad (32)$$

and the special cases being:

$$X(0) = X_{DCT}(0), X(N/2) = X_{DCT}(N/2) \quad (33)$$

The above equations give us a way to compute the DFT using the DCT and the DST. We can now develop a recursive formulation for the computation of the DCT and the DST. It is this process which helps achieve the reduction in computational complexity for the QFT algorithm.

The following properties of the cosine function lay the basis for the recursive DCT algorithm:

$$\cos\left(\frac{\pi(N-n)k}{N}\right) = \cos\left(\frac{\pi nk}{N}\right), \text{k = 0, 2,., N} \quad (34)$$

$$\cos\left(\frac{\pi(N-n)k}{N}\right) = -\cos\left(\frac{\pi nk}{N}\right), \text{k = 1, 3,., N-1} \quad (35)$$

$$2\cos\left(\frac{\pi n(2k+1)}{N}\right)\cdot\cos\left(\frac{\pi n}{N}\right) = \cos\left(\frac{2\pi nk}{N}\right) + \cos\left(\frac{2\pi n(k+1)}{N}\right) \quad (36)$$

By splitting (24) into two summations and a change of variables, we get for an even k:

$$X_{DCT}(k) = \sum_{k=0}^{N/2} (x(N-k) + x(k))\cdot\cos\left(\frac{2\pi nk}{N}\right) \quad (37)$$

We can define a new sequence, x_e as:

$$x_e(k) = x(k) + x(N-k), \text{k = 0, 1,... N/2 - 1} \quad (38)$$

Also, the N/2th point of this sequence is the same as that of the original sequence. Thus we can formulate the recursive DCT for the even points as:

$$DCT(2k, N+1, x) = DCT(k, N/2+1, x_e), \text{k = 0, 1,... N/2} \quad (39)$$

Using (35) and (36) we can define a recursive equation for the odd DCT points using a new sequence x_o, defined as follows:

$$x_o(k) = \frac{x(k)-x(N-k)}{2\cos(\pi k/N)}, \text{k = 0, 1,..., N/2 -1} \quad (40)$$

Using the above definition the recursive equation for the odd DCT points is defined as:

$$DCT(2k+1, N+1, x) = DCT(k, N/2+1, x_o) + DCT(k+1, N/2+1, x_o)$$

$$\text{k = 0,1,... N/2 -1} \quad (41)$$

A similar recursive formulation can be derived for the DST using symmetry properties of the sine function.

$$DST(2k, N-1, x) = DST(k, N/2-1, x_o), \text{k = 1,..., N/2 -1} \quad (42)$$

$$DST(2k+1, N-1, x) = DST\left(k, \frac{N}{2}-1, x_e\right) + DST\left(k+1, \frac{N}{2}-1, x_e\right) \quad (43)$$

Since the complex operations occur only in the last stage of the computation where the DCT and DST is combined, the QFT is well suited for operation on real data. The inventor of this algorithm claims that the number of operations required to perform an N-point QFT is $11((N/2)\cdot\log N)-(27N)/4+2$. This however does not seem to take into account the operations required to compute the even and odd functions.

3.6. Decimation-in-time-frequency Algorithm

This is by far the least known FFT algorithm, though it was published only in 1994. The DITF algorithm is based on the observation that, in a DIF implementation of a radix-2 algorithm, most of the computations (especially complex multiplications) are done in the initial stages of the algorithm. In the DIT implementation of the radix-2 algorithm, most of the computations are done in the final stages of the algorithm. Thus starting with the DIT implementation and then shifting to the DIF implementation at some transition stage seems intuitively to be a computation saving process. A block diagram illustrating this implementation is as follows:

Equation 3 and Equation 4 define the DIF radix-2 computation. The DIT radix-2 computation is defined by the following equation.

$$X(k) = \sum_{n=0}^{N/2-1} x(2n)\cdot W_{N/2}^{kn} + \left(\sum_{n=0}^{N/2-1} x(2n+1)\cdot W_{N/2}^{kn}\right)\cdot W_N^k \quad (44)$$

Note that the first summation in the above equation is the $N/2$-point DFT of the sequence comprised of the even numbered points of the original sequence and the second summation is the $N/2$ -point DFT of the sequence comprised of the odd numbered points of the original sequence. It has been proved that the least number of multiplications occur when the transition stage has been chosen to be $(\log_2 N)/2$. There is a conversion from the DIT coefficients to the DIF coefficients at the transition stage.

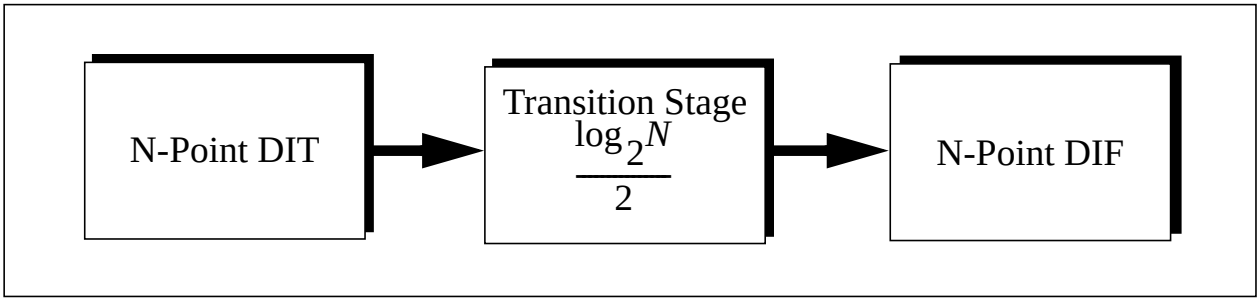

Figure 7. Block diagram illustrating the DITF algorithm

$$DIF(k) = W_N^{pq} \cdot DIT(k) \quad (45)$$

where p is the index of the set to which k belongs and q is the position of k in that set. The indices of the sets need to bit-reversed. The following block diagram for a 16-point DITF better illustrates the operation.

The total number of real multiplications involved in the DITF computation is

$$2N\log N - 10N + 8.\frac{N}{2^s} + 8 \cdot 2^s - 8 \text{, where S is the transition stage.} \quad (46)$$

On minimizing this expression, we get the optimal transition stage for minimum number of multiplications as $\frac{\log N}{2}$.

4. BENCHMARKING CRITERIA

Since most of the FFT algorithms have been around for nearly a quarter decade now and have been benchmarked several times by different researching groups, one would ask, why again?

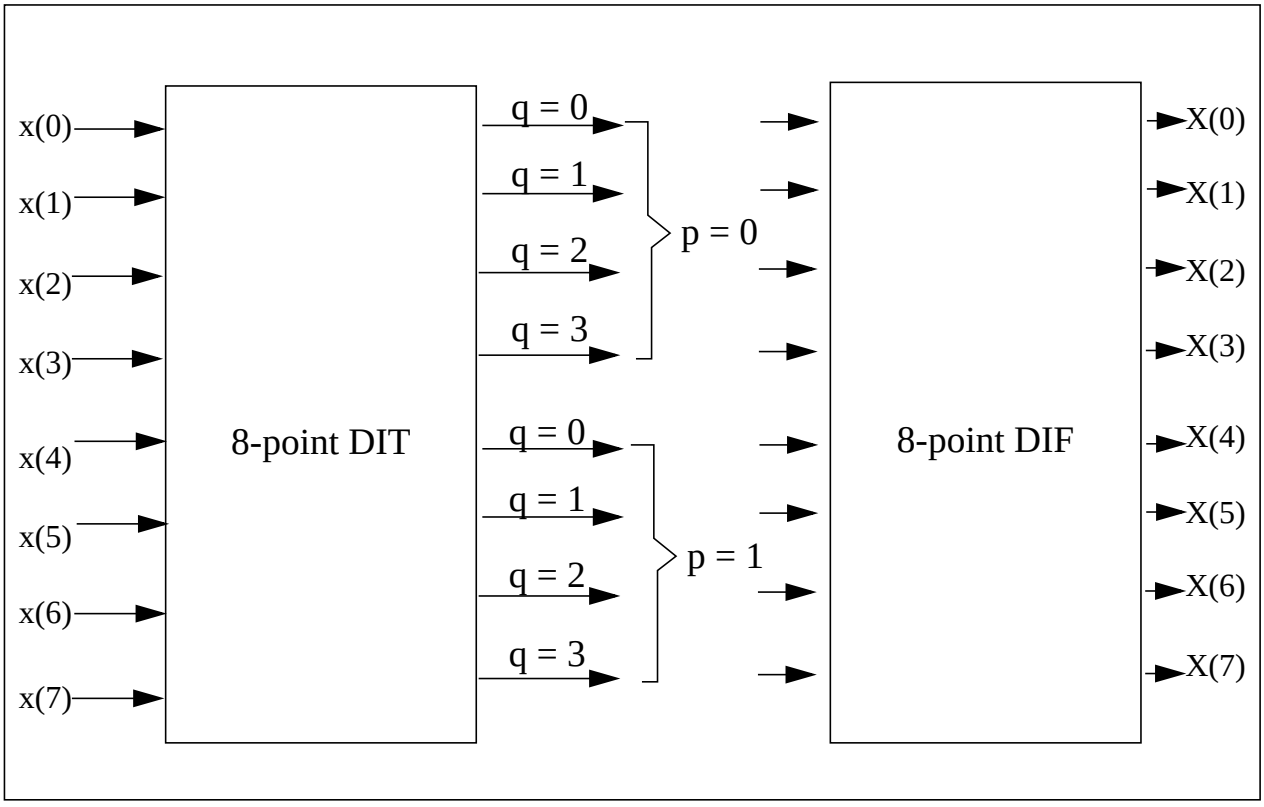

Figure 8. 8-point DITF implementation illustrating the transition stage generation.

Most of the FFT benchmarking has been done for special purpose hardware like DSPs. Most often the benchmarking criteria has been the number of simple mathematical operations or computation speed. Since a lot of DSP research still continues to be on general purpose computers, we realized a need for a comprehensive benchmarking of a large collection of FFT algorithms.

4.1. Traditional Approach

It has been a traditional belief that the efficiency of an algorithm is most influenced by the arithmetic complexity. It is usually expressed by the counts of real multiplications and additions in the whole algorithm. Often, this was used as the only measure of efficiency since the other factors were neglected.

Often, in many real situations, the neglected, or overlooked, operations have the same importance to the speed of the algorithm. Some of these operations include, number of data transfers, trigonometric function table generation, indexing (which translates to integer arithmetic and program control statements). Some of the parameters are mutually independent. Also, with the advancement of computer technology, we have at our disposal a myriad of architectures and compiler characteristics. These were not major issues in most previous benchmarks and hence were overlooked.

4.2. Dimension of the Problem

Having seen why we need a new benchmarking strategy to fit the needs of current FFT applications, we now give a picture of the whole benchmarking problem as we see it. As explained earlier, the computer technology used is a very important consideration. The actual algorithm complexity is by far the most important criterion, ideally we'd like it to be the only criterion. We then have the problem of different implementations of the same algorithm. One can think of some neat tricks to reduce arithmetic operations. In many cases, it turns out to outweigh the algorithm's true complexity. The issue of memory usage is also very important when an application is memory constrained. It is a well known fact that we can improve speed by trading off memory.

4.3. Criterion Used

Having analyzed the dimension of the problem and the various ways algorithms could be benchmarked, we decided on using the following criteria for benchmarking.

Number of Computations:

Since many general purposes CPUs have different performance on floating point data and integer data, we decided to separately account for floating point arithmetic and integer arithmetic. Also, most indexing and loop control is done using integer arithmetic. It is a well known fact that most architectures work faster on floating point data rather than integer data. Since there is a finite time used in shifting data in registers, and the fact that most FFT algorithms deal with multiples of 2 which incurs binary shifting, we decided to measure the number of binary shifts used in an algorithm.

Computation Speed:

In most present day applications, with the computer technology providing us with faster CPUs, the fastest algorithm is by far treated as the best algorithm, all other factors remaining the same. Thus an obvious choice to differentiate algorithms is the computation speed. One needs to however consider the CPU speed and implementation specifics before arriving at a conclusion regarding the efficiency of the algorithms. It has been seen in previous benchmarking efforts that the average time taken over a number of iterations is a better measure than a timing of one iteration. Using this iterative approach helps us deal with measurement noise and transients more effectively. It has also been seen recently that using the median measure is more accurate than using the average. We therefore report median CPU time in our benchmarks.

Memory Usage:

It is a well known fact that we can speed up algorithms considerably by trading off memory usage. This fact prompted us to include memory usage as one of the yardsticks to judge the effectiveness of algorithms. One needs to consider the fact that not many applications can afford unlimited memory space for the FFT algorithms usage since FFT algorithms are generally play only a small part in processing data.

Compiler Optimizations:

Memory usage is independent of the particular architecture and compiler characteristics the FFT algorithm runs on. However, the computation speed is highly dependent on the degree of optimization the compiler can achieve. Computation speed variations as high as 50% are possible if the compiler can optimize to a very high degree. We therefore chose to evaluate the performance of two of the commonly used compilers for C++, **GCC** and **MSVC++.** These compilers differ in many aspects internally.

Object Code Size:

Since most applications after initial testing on general purpose computers finally go onto an add-on card of some type, the object code size is an important aspect to consider, especially for systems with limited on-board memory. In cases when memory is very limited, the lesser complex algorithms could be the only choice for the user mainly because of their low object code size.

5. IMPLEMENTATION DETAILS

Apart from trying to benchmark the algorithms we choose, we also wanted to provide the general research community with a well implemented and a documented version of these algorithms. Modularity and portability were one of the main issues in designing the software. Since the problem we had in hand was poly-algorithmic in nature, the natural choice of the programming language was C++. It also facilitates some other features like reference counting, templatization etc. which were some of the concepts we wanted to explore as part of future research.

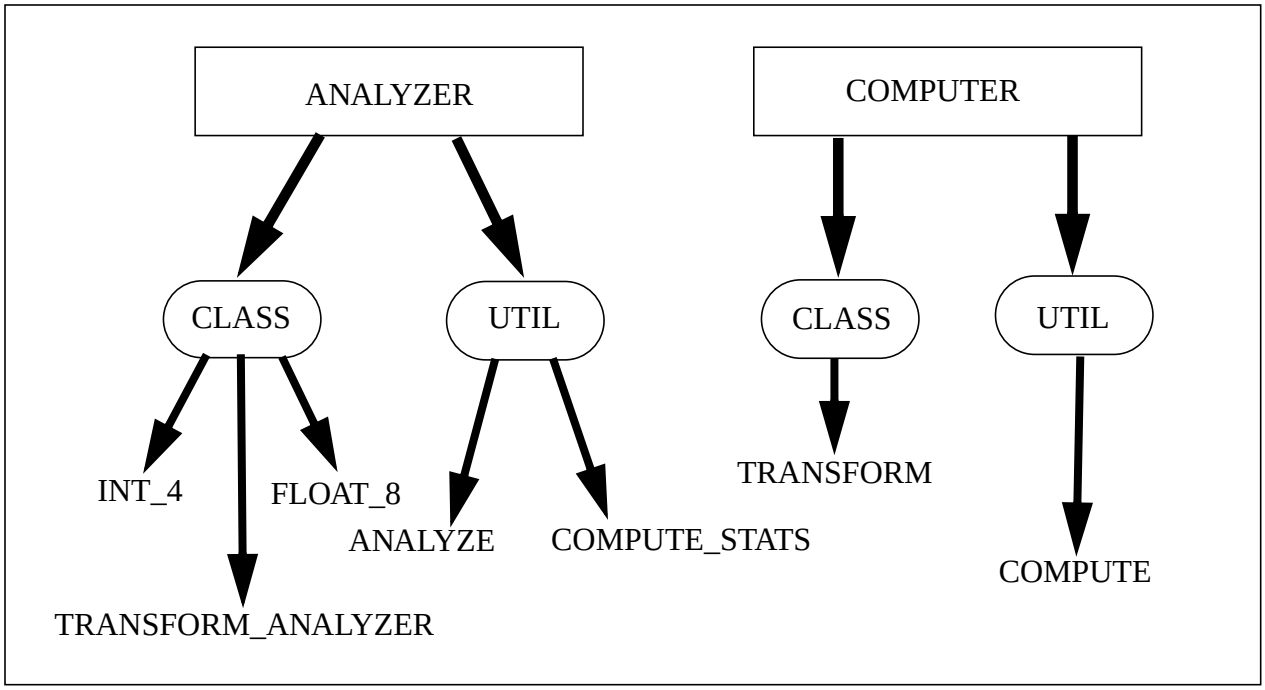

Figure 9. Benchmarking Software structure

5.1. Two-fold Approach

The top level software was divided into two parts. The first one was intended to be an analysis tool which would generate the benchmarking information, and the other was intended for the purpose of computing an FFT. The analysis tool had a lot of overhead added to it to help provide the benchmarking data. The part used to compute the FFT alone is made devoid of any of these frills.

The general structure of the code is illustrated in Figure 9. . The main classes comprising the analysis tool are the **transform_analyzer, int_4** and the **float_8.** Objects of the **int_4** class behave exactly as a normal 4-byte integer, except that they are capable of incrementing the necessary operation counts when they are involved in an arithmetic operation and memory usage counts when they are instantiated. This could also be implemented in the standard reference counting mechanism in C++. We chose to stick to the simpler solution of using *static* counters in the class. Objects of the **float_8** class behave just like a double precision floating point number with additional features the **int_4** objects possess. Apart from the classes, we have some utilities to performing the benchmarking. The **analyze** utility does all the I/O for the process. It is here that the timing information is computed and processed. One reason why the utility gathers the timing information is that we feel that the lookup table creation, swap space initialization and other such secondary processes involved in a algorithm should also be accounted with the algorithm. In fact we treat the arithmetic involved in the lookup table generation as being part of the overall computational load of the algorithm. We use the **compute_stats** utility for this purpose which does a mean and median computation of the cpu time taken.

```
isip01_[2]: analyze_fft.exe -help
name: analyze_fft
synopsis: analyze_fft [options]
descr: tests all FFT algorithms supported in the fourier transform class
example: analyze_fft -algo df -order 16 -debug 1

arguments:
-algo: fourier transform algorithm (default = discrete fourier)
-order: transform order (default = 256)
-datatype: data type (0=real;1=complex;default=real)
-debug: debug level (default = 0)
-help: display this help message

man page: none
```

Figure 10. Command line interface to the software

```
isip01_[2]: analyze_fft.exe -order 16
algorithm = df
order = 16
data type = 0
debug_level = 0

the number of float multiplications:    512
the number of float additions:          512
the number of integer multiplications:    0
the number of integer additions:       1056
the number of shifts:                     0
the number of bytes of memory:        19132
median of time taken per iteration: 1100

<end of analyze_fft.exe>
```

Figure 11. Sample output of the fourier_transform_analyzer

5.2. Specifics

The code structure for the **computer** is however much more simple since its only application is to compute the FFT. It comprises of the **transform** class which uses the standard integer and floating point arithmetic. The software can handle both real and complex input data. However, it always returns complex output data. We have followed the convention followed in most DSP applications of providing data in a *packed* format, wherein the data array is an array of complex numbers i.e. the real part followed by the imaginary part for each data point.

Since portability to various platforms, especially machines running WindowsNT and SunOS, was a requirement for our work care had to be taken to adhere to the ANSI standards in the coding style. Especially crucial were the static object declaration which was allowed by **GCC** but was not by **MSVC++.** To keep matters simple we have used a command line control to the software. The user can choose the algorithm he/she wishes, the order of the FFT and the input datatype. Default values for all these parameters can be set in a constants file. The help file in Figure 10. illustrates the interface to the software. Figure 11 shows a sample output of a typical **analyzer** usage.

One of the driving forces for our work was to investigate how well advanced object-oriented concepts could be used is a poly-algorithmic problem like the FFTs. This led us to implement the algorithms in template structure. Using templates allows the compile-time composition of the problem which is very efficient in terms of speed when compared to run-time composition of standard C++ implementations. To completely templatize the problem we had to create templates for each algorithm. We then create a wrapper around these algorithms which does the I/O for us. We note from the previous algorithm descriptions that most algorithms can use the same lookup table to get the necessary twiddle factors. If an application uses two algorithms of the same order simultaneously, it makes sense not to compute the lookup table twice but share the lookup table The reference counting mechanism is an elegant way to achieve this functionality. Under this paradigm we have a *handle* to a lookup table created each time a lookup table is instantiated. Note that this is different from creating a new lookup table. It is through the *handle* that the data in the lookup table is accessed. Once the need for the lookup table is over, we delete the handle.

We can extend this concept to one level higher by creating a list of these reference counted tables, one each for a lookup table of an unique order. We have implemented this concept using a **MAP** from the Standard Template Library in C++. A MAP provides a quick way to access elements in a list since it is implemented as Red-Black binary search tree which has a low search complexity compared to a linear search. Extending this approach is part of our future work.

6. BENCHMARKING RESULTS AND ANALYSIS

Having implemented all the algorithms, we chose for the benchmarking purpose, under a common framework, we comprehensively benchmarked these algorithms. In the process we noticed many results which were confirming to what the theory suggested, but we also saw many counter-intuitive results. Many of these can be attributed to the compiler optimizations rather than discrepancies in the algorithms or the measurement process. Some algorithm implementations tend to be more amenable to optimizations than others. In implementing the algorithms, we have tried to use uniform techniques for operations like bit-reversal, lookup table generation etc. so that the difference we see in performance can be attributed only to algorithm efficiency.

6.1. Computation Speed

In most present day applications computation speed is by far the most important aspect of an algorithm one is interested in. Not only can we benchmark algorithms in terms of computation speed, but we can also benchmark processors using the same criterion. It is interesting to see how

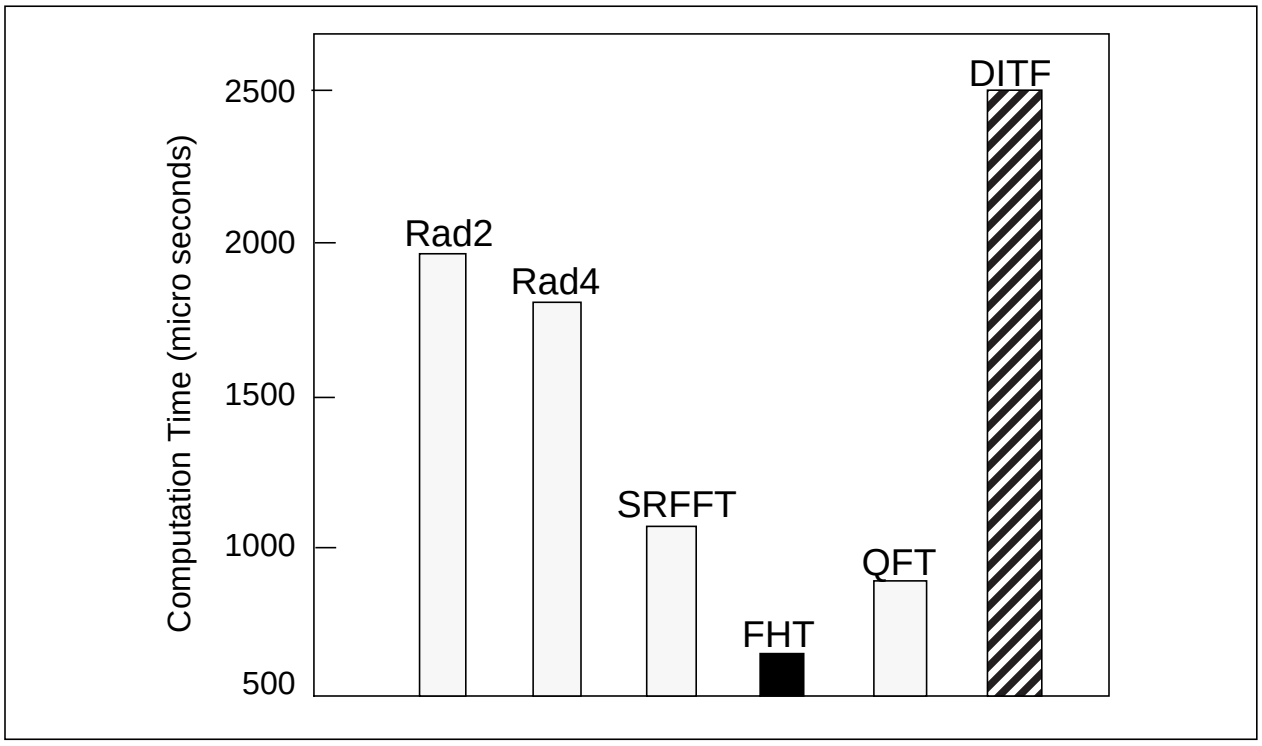

Figure 12. Computation time across different algorithms on the same machine for a 1024 point FFT

the performance of the algorithms scales up in terms of processor speed. This is very heavily influenced by the architecture, cache usage etc.

Algorithm	**FFT ORDER**					
	16	64	256	1024	4096	16384
RAD2	20	60	280	1960	10900	97100
RAD4	20	60	250	1800	9720	58220
SRFFT	20	40	160	1060	6140	38100
FHT	**20**	**40**	**140**	**640**	**3800**	**38100**
QFT	20	40	160	880	6560	44020
DITF	20	60	360	2500	12320	104080

Table 1 Computation time of various algorithms in computing several orders of FFT (in μs)

Computation Speed Across Algorithms

We ran each of the algorithms compiled using the same compiler (**GCC** version 2.7.2.1) on a 2 processor PentiumPro 200Mhz processor machine with 256MB RAM. The implementations used were without any overhead in terms of benchmarking information like arithmetic count etc.

Figure 12. shows the computation speed while computing a 1024 point real FFT. Level-2 optimization was used for the compilation process which tries to maximize speed without space trade-off. As can be seen from the plot, the ratio of the speed difference between the worst (FHT) and the worst (DITF) algorithms is as high as 4. It has been consistently seen in our benchmarking that the FHT is by far the most efficient algorithm in terms of computation speed. The ranking of the algorithms does however change when benchmarked on a different machine. For example when run on an UltraSparc2 machine SRFFT performs better than QFT for a 1024 point FFT. This is possibly attributed to the paging mechanism and cache usage on the different architectures and how well these algorithms are susceptible to these issues. Table 1 shows the performance variations of these algorithms across various orders.

Computation Speed Across Machines

Thus far we have seen the performance of the algorithms independent of the machine they are run on. However, in many applications, the designer knows in advance the type of architecture the intended algorithm will run on. We can use the information related to performance of these

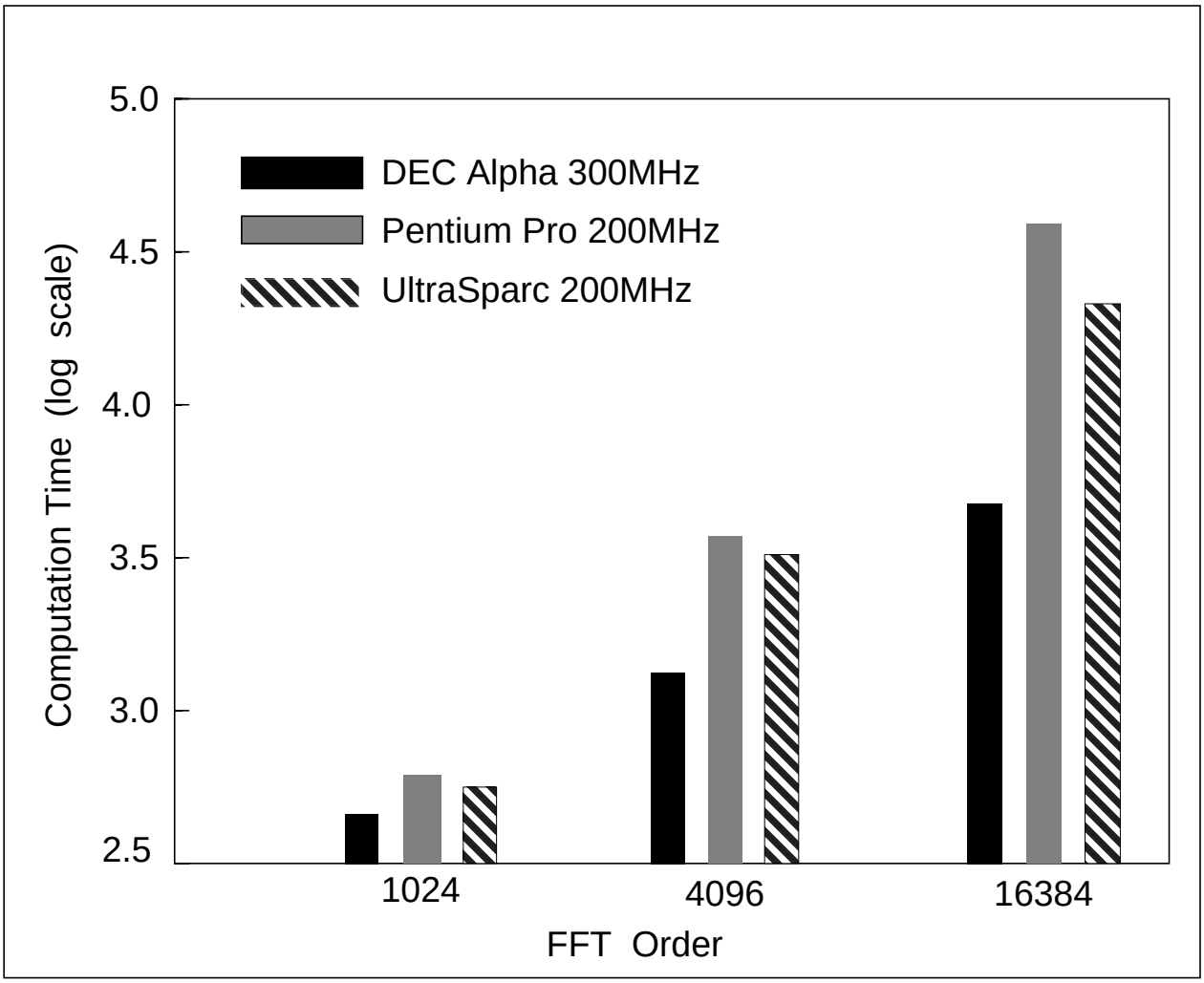

Figure 13. Comparison of computation speed across three different CPUs for the FHT algorithm

algorithms in terms of machine CPU architecture. For our benchmarking task we choose, UltraSparc, DEC Alpha 21164 and Pentium Pro. Of these UltraSparc and the Pentium Pro were 200 MHz processors with 256MB and the DEC Alpha was a 300MHz processor with 128MB RAM. The Alpha machine was running WindowsNT and the UltraSparc and Pentium Pro were running SunOS. The DEC Alpha machine has a 2MB cache. The Pentium Pro has a 512KB/CPU and the UltraSparc2 has 1MB/CPU. This feature will have a bearing on the computation speed of algorithms for large input data sizes. A comparison of the computation time for the FHT algorithm on the three machines is shown in Figure 13. Note that in the CPU times taken on the DEC Alpha has been normalized to represent 200MHz processor speed for fairness in comparisons.

6.2. Compiler Effects

Compiler technology has advanced greatly over the past decade or so. Earlier benchmarks were not affected by compiler optimizations as much as we have been now. During preliminary tests we found that going from no optimizations to level-1 optimizations in **GCC** improved the computation speed by as much as 300%. This prompted us to explore this facet of benchmarking more closely. The more commonly used compilers being **GCC** and **MSVC++**, we decided to benchmark performance of algorithms when compiled using these compilers. Some of the important modifications present day compilers try to achieve are listed below:

1) *Tail recursion elimination* - converts self-recursive procedures into iterative procedures, saving manipulation time

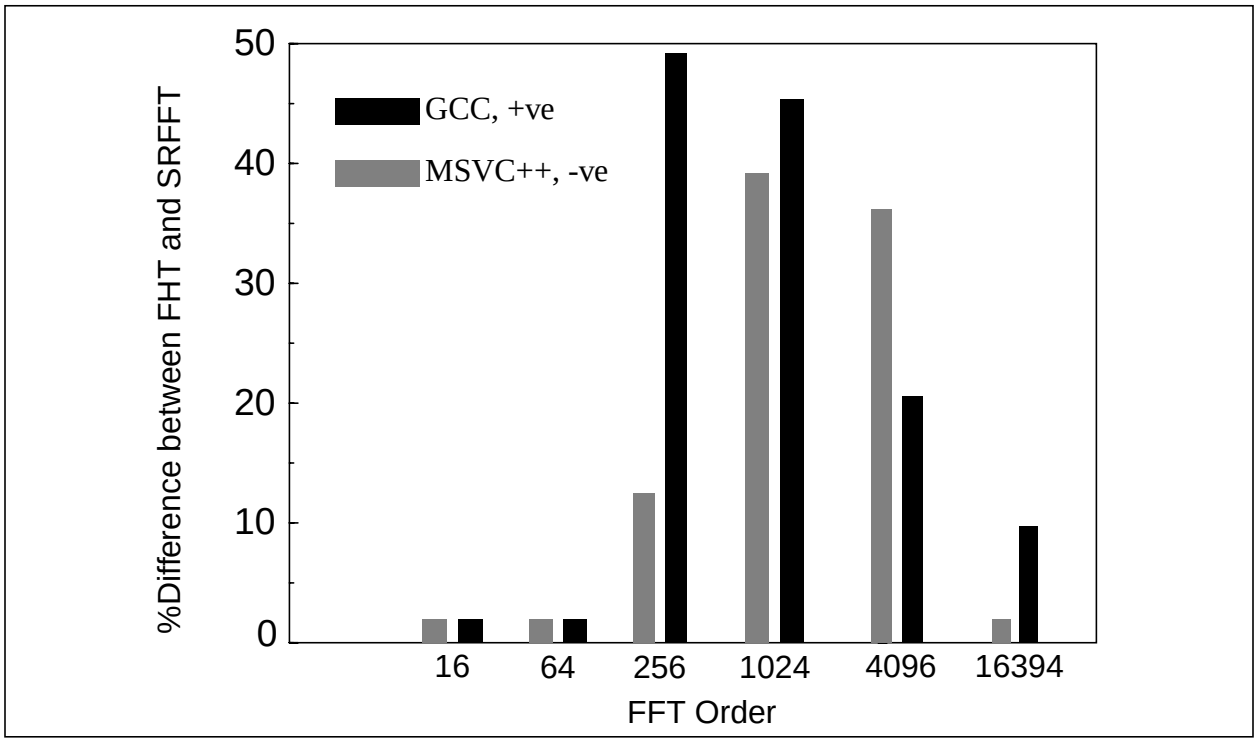

Figure 14. Difference between computation time of FHT and SRFFT when compiled using GCC and MSVC++

2) *Loop-invariant code motion* - locates and removes computations that yield the same result

3) *Profiling* - allows optimizations to adapt themselves to program behavior

4) *Induction-variable strength reduction* - replaces slower operations y faster ones

5) *Loop unrolling* - reduce run-time by reducing loop overhead and increasing opportunities for more efficient instruction pipelining

6) *Loop inversion* - convert pre-test loops into post-test loops, reduce the number of branches required per iteration

Optimizations 2 and 5 seem to most effect computation speed in FFT algorithms because of the nature of the problem wherein it is common to see loop cores with intermixed additions and multiplications. Such computations provide for significant pipelining opportunities which is by far the most important issue in iterative computation intensive algorithms.

GCC has three levels of optimizations which try to achieve higher speed and lower code size to a varying extent. Level 3 turns on all optimizations. Level 2 tries out all optimizations which do not involve a speed-space trade-off, unlike Level 1. **MSVC++** has similar optimization flags to achieve various optimizations. Table 2 shows the effect of compiler optimizations on different algorithms where we show the percentage change in computation speed for each algorithm depending on the optimization level. We use **GCC** to illustrate this observation. The baseline timing is the Level 1 timing for all the algorithms.

Algorithm	Level 2	Level 3
RAD2	**1.0**	**1.0**
RAD4	**9.2**	**7.9**
SRFFT	**13.2**	**13.2**
FHT	**3.0**	**3.0**
QFT	**2.0**	**0.0**
DITF	**2.3**	**3.1**

Table 2% change in computation time of algorithms for a 1024-point FFT

We could call this the intra-compiler effect. We can look at this problem from a different perspective, the inter-compiler effect. Different compilers can optimize algorithms to a varying extent. This is a very important aspect to consider. An application developer could look at benchmarks performed using code compiled on **GCC** and assume the effects to translate smoothly to code compiled using **MSVC++**. Unfortunately this does not happen. Figure 14.

proves this fact by comparing the computation speed difference between SRFFT and FHT on real data.

6.3. Memory Usage and Object Code size

One of the key issues in portable applications is memory usage. Each process in the application needs to be optimized for memory usage. In most large applications like speech recognition, FFT accomplishes only a small part of the processing. Thus most application designers would like to keep memory usage to a minimum. Keeping this in mind we benchmarked the memory used by each of these algorithms. We feel that the object code size also has a similar effect on the design process and hence report this measure along with memory usage. When we distribute libraries, object code size becomes very crucial.

In computing memory usage, we also account for the input and output data arrays, lookup tables and any intermediate swap space used by the algorithm. Since we wanted to keep the structure of algorithms uniform, we have implemented all algorithms with lookup tables. Thus, any difference in memory usage can thus be attributed to the actual swap space usage difference. Object code size is also a direct measure of the complexity of an algorithm. Most of the faster algorithms have a large object code size. Table 3 shows the memory usage for a 1024 point FFT on a PentiumPro compiled with **GCC.**

Algorithm	Memory Usage (Bytes)	Object Code (Bytes)
RAD2	**72440**	**5190**
RAD4	72536	5293
SRFFT	72508	6275
FHT	72652	11506
QFT	122072	9800
DITF	78632	8691

Table 3 Memory usage and object code size in computing a 1024-point FFT

As we have already mentioned the simpler algorithms have the advantage of low memory usage and small object code size. We see this from the above table where RAD2 is the most memory efficient algorithm. Also note that QFT is the worst algorithm as far as memory usage goes.This is accounted for by the large work space required to perform the recursion in the DCT and the DST algorithms. The FHT is the most complex algorithm in the present implementations due to the way the trigonometric values are computed (it uses the Euler’s approximation to compute the cosines and sines from lower order values). Using the approximate formula gives it added speed losing on accuracy in the process.

6.4. Number of Computations

The number of arithmetic computations have been the traditional measure of algorithm efficiency. With the advent of new computer technology and implementation techniques, their importance has dwindled over the years. As will be discussed later the number of computations derived form the typical butterfly diagrams is not of much use when an implementor chooses to use a neat trick to avoid some redundant computations. In any case, we have generated concrete numbers for the arithmetic computations involved in the various algorithms. Since showing the numbers for all the algorithms and all FFT orders will clutter the implicit pattern amongst algorithms, we chose to provide numbers for a 1024 real FFT in Table 4 and compare and contrast these number amongst themselves and those predicted by theory. The least operations in each category is highlighted.

One very intuitive observation from the table above is the fact that the faster algorithms perform lesser number of computations than other algorithms. In section we mentioned that the RAD4 algorithm requires 25% lesser multiplications than the RAD2 algorithm. This is confirmed here since 14336 is approximately 75% of 10480. Notice the excessive number of integer adds in the QFT. Most of the integer arithmetic is accounted for by loop control or re-indexing. In the QFT implementation we have, the recursion DCT and DST recursion is implemented by moving pointers in a common workspace. This results in the large integer operations. The large number of operations for the DITF algorithm are because of the numerous bit-reversal stages which the data undergoes. In the numbers quoted by the author[7], he seems to have overlooked the bit-reversal overhead.

Algorithm	Float Adds	Float Mults	Integer Adds	Integer Mults	Binary Shifts
RAD2	14336	20480	19450	2084	1023
RAD4	8960	14336	12902	3071	277
SRFFT	**5861**	**5522**	12664	2542	1988
FHT	7420	8841	**3235**	2048	**12**
QFT	9026	2560	29784	**1048**	144
DITF	14400	17664	20333	1076	1074

Table 4: Number of computations involved in computing a 1024-point FFT

One should however not be carried away by the performance of FHT. The main drawback of the FHT is the need for a call to the real routine twice to compute a complex FFT. QFT also uses a similar methodology. The number of computations doubles in these cases as against a moderate change for the other algorithms.

6.5. Accuracy

In the mad rush for numbers, there generally is a very important aspect of algorithm design, **accuracy,** which gets masked. There have been many tricks developed over the years for fast

computation of the trigonometric values. One amongst them is the Euler's approximation formula, which use previously computed values to get a rough estimate of the current value. The current implementation of FHT uses such a technique. This prompted us to look at the accuracy issues when such an approximation is used in with the algorithm.

We used the following simple DFT properties

$$DFT[x(n)] = y(n) \tag{47}$$

Then,

$$1/N \cdot DFT[y(n)] = x(n)\big|_N \tag{48}$$

which says the DFT of the DFT of $x(n)$ is a circularly convolved version of $x(n)$.

Using the above definitions we compute the SNR for a randomly generated input sequence. Since we have used double precision arithmetic, we expect a large SNR, typically in the range of 300dB. Figure 15. shows the% change in SNR from an order 64 FFT to an order 16394 FFT.

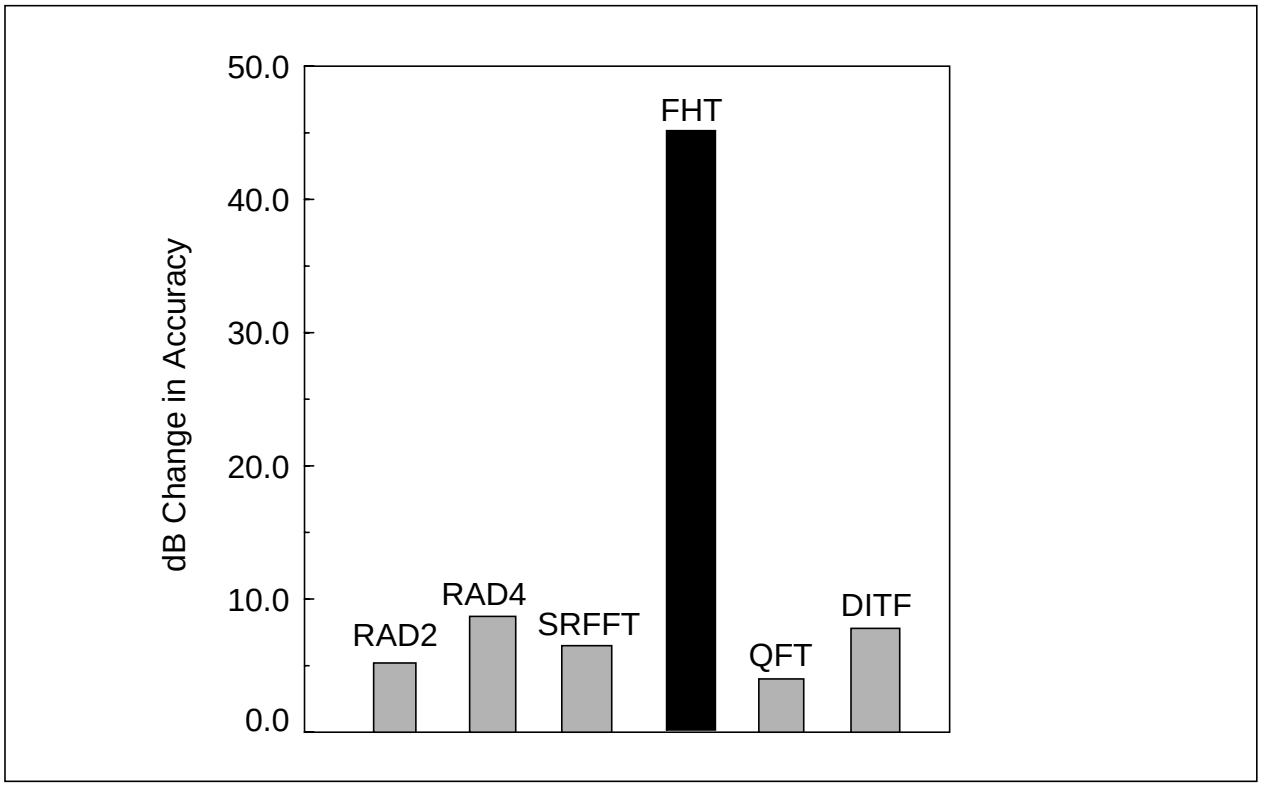

Figure 15. Comparison of change in accuracy form a 64 point FFT to a 16384 point FFT

7. CONCLUSIONS

We have completed the creation of a test bed to demonstrate the high-performance portability in libraries using some of the well known object oriented concepts. We have a wide spanning collection of FFT algorithms, including Radix-2, Radix-4, Split-Radix, Fast Hartley, Decimation-in-time-frequency and the Quick Fourier transforms, implemented under a common framework to facilitate a poly-algorithmic selection process to achieve this portability. For the purpose of this selection, we needed a comprehensive set of benchmarks including the computational complexity of the algorithms, memory usage and compiler dependence.

We have benchmarks for DEC Alpha 300MHz, Sun UltraSparc 200MHz and Pentium Pro 200Mhz machines. The compilers compared are the GNU's GCC and MSVC++. We have compared the algorithms based on the number of computations involved, computation speed, memory usage and their behavior towards compiler optimizations.

The results indicate that the **overall best** algorithm for DFT computations is the **FHT** algorithm. It is the fastest algorithm on all platforms with a reasonable memory requirement. If the choice of an algorithm is to be made solely based on the memory requirements, the Radix-2 algorithm is the best. The **SRFFT** and the **FHT** are comparable in terms of **number of computations** and are the **most efficient.** As can be seen from the results on the number of computations, the floating point operations tend to influence the computation speed the most.

From the comparison of the machine performance, the difference between Pentium Pro and the others increases drastically for longer length FFTs. This can be attributed to the worse cache utilization of the Pentium Pro. We have demonstrated the effect compiler optimizations could have on the performance of algorithms. The SRFFT algorithm responds the best to compiler optimizations. We have noticed an improvement in computation speed as high as 13.2% based solely on compiler optimizations.

Some of the more controversial findings in this work is that the QFT and DITF do not perform as well as quoted in the literature. In the literature, overhead involved in the computations using these algorithms is neglected, especially array indexing, and data preparation like computation of even and odd components of a data set.

As part of future work, we have started converting the algorithm implementations into a template based library so that better abstraction is possible. From the results of the benchmarks we have realize the need to include a cache model in our code design to have a better explanation of cache and compiler related issues. As noted earlier, compiler effects were very drastic and we could not explain the underying phenomenon based on the algorithm implementations.

All the code generated through this project is available from our website at *www.isip.msstate.edu.*

8. ACKNOWLEDGEMENTS

This work was supported by DARPA through US Air Force's Rome Laboratories under contract F30602-96-1-0329. I would like to express my appreciation to the constant guidance provided by

Dr. Joseph Picone and Dr. Anthony Skjellum in the past one year, involving both going through the code and discussing algorithm related issues several times during the course of this project. I would also like to thank Mr. Shane Hebert and Mr. Gerhard Lehnerer from ICDCRL, Department of Computer Science, Mississippi State University, for their help with the machines. I would also like to thank Mr. Jonathan Hamaker for implementing the DITF algorithm and helping in designing the software structure.

9. REFERENCES

[1] J.W. Cooley and J.W. Tukey, "An Algorithm for Machine Computation of Complex Fourier Series," *Math. Comp.*, vol. 19, pp. 297-301, April 1965.

[2] R.N. Bracewell, *The Hartley Transform*, Oxford Press, Oxford, England, 1985.

[3] R.N. Bracewell, "Fast Hartley Transform", Proceedings of IEEE, pp. 1010-1018, 1984.

[4] H.S. Hou, "The Fast Hartley Transform Algorithm", *IEEE Transactions on Computers*, pp. 147-155, 1987.

[5] P. Duhamel and H. Hollomann, "Split Radix FFT Algorithm," *Electronic Letters*, vol. 20, pp. 14-16, Jan. 1984.

[6] H. Guo, G.A. Sitton, and C.S. Burrus, "The Quick Discrete Fourier Transform," *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, vol. III, pp. 445-447, Adelaide, Australia, April 1994.

[7] A. Saidi, "Decimation-In-Time-Frequency FFT Algorithm," *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, vol. III, pp. 453-456, Adelaide, Australia, April 1994.

[8] C.S.Burrus and T.W.Parks, *DFT/FFT and Convolution Algorithms: Theory and Implementation*, John Wiley and Sons, New York, NY, USA, 1985.

[9] J.G. Proakis, D.G. Manolakis, *Digital Signal Processing - Principles, Algorithms and Applications*, Macmillan Publishing Company, NY, USA, 1992.

[10] A.V. Oppenheim, R.W. Schafer, *Digital Signal Processing*, Prentice-Hall International Inc., Englewood Cliffs, NJ, USA, 1989.

[11] C.V. Loan, *Frontiers in Applied Mathematics - Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[12] R.N. Bracewell, "Assessing the Hartley Transform," *IEEE Transactions on Acoustics Speech and Signal Processing*, pp. 2174-2176, 1990.

[13] M. Popovic, D. Sevic, "A new look at the Comparison of Fast Hartley and Fourier Transforms," *IEEE Transactions on Signal Processing*, vol. 42, pp. 2178-2182, August, 1994.

[14] P.R. Uniyal, "Transforming Real-Valued Sequences: Fast Fourier versus Fast Hartley Transform Algorithms," *IEEE Transactions on Signal Processing*, vol. 42, pp. 3249-3253, November, 1994.