Performance Evaluation on FFT Software Implementation

Xiangyang Liu[1], Xiaoyu Song[2], and Yuke Wang[1]

***Abstract*—It is known that FFT algorithms have complexity of O(*n*log$_2$*n*), where *n* is the input size. Many new algorithms claim certain theoretical advantage; however, their real performance in application is questionable. The paper presents a systematic performance evaluation on different FFT software implementations. Different code techniques such as recursive, iterative, TFBBGM, TFRM with further expansion are used for real application sizes from 512 to 2048 points. Contrary to the common belief that recursive programs are slower, we find that recursive programs are not necessarily slower for commonly used FFT. Our comparative study constitutes the first attempt to evaluate the real performance of different FFT approaches.**

***Index Terms*—FFT, DFT.**

I. Introduction

In digital signal processing, the discrete Fourier transform (DFT) plays an important role in the analysis, design and implementation of discrete-time signal-processing algorithms and systems [1, 2]. The fast Fourier transforms (FFT) are efficient algorithms to compute DFT. FFT is used widely in digital signal processing fields. Its performance is critical for many real-time applications.

The FFT algorithms are based on the principle of decomposing the computation of DFT into sequences of smaller DFTs. The first FFT algorithm was discovered by Guass in the 18th century and rediscovered by Cooley and Tukey [3] in 1960s. Significant advances include higher radix FFT algorithms [4], mixed-radix FFT algorithms [5], split-radix FFT algorithms [6][7], recursive FFT algorithm [8], and the decimation-in-time (DIT) and the decimation-in-frequency (DIF) FFT algorithms [9]. Most of these algorithms illustrate FFT with similar FFT diagrams, which are evolved from the nature of the FFT algorithms and constructed by basic butterfly structures, such as the 8-point radix-2 FFT diagram shown in Figure 1.

FFT algorithms can be implemented on multiple platforms. For example, FFT algorithms have been implemented on application specific integrated circuits (ASIC) as FFT processors [10] for high-speed or low power hardware design. However, FFT algorithms designed in hardware processor are often tailored to specific application, hence is not flexible. FFT has also been implemented in software on general-

Xiangyang Liu is with the Department of Computer Science, University of Texas at Dallas, Richardson TX, 75080, USA (e-mail: xxl063000@ utdallas.edu).

Xiaoyu Song, is with Department of Electrical and Computer Engineering, Portland State University, P.O.Box 751 Portland, OR 97207-0751, USA (e-mail: song@ee.pdx.edu).

Yuke Wang is with the Department of Computer Science , University of Texas at Dallas, Richardson, TX, 75080, USA (e-mail: yuke@utdallas.edu).

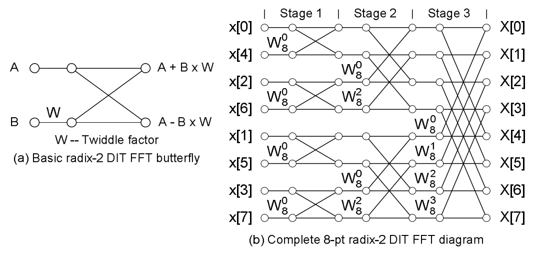


Figure 1. The 8-pt radix-2 DIT FFT diagram.

purpose processors as building block of simulation data processing systems [11]. Software-based implementations of FFT on general processors are less cost and flexible, but they are typically slower than hardware on comparable technologies. Digital signal processors (DSPs) are specific processors optimized for various signal-processing applications such as FIR, IIR filters and FFT. Software implementations of FFTs on DSPs are getting popular for their excellent tradeoff among cost, performance, flexibility, and implementation complexity.

It is known that FFT algorithms have complexity of O(*n*log$_2$*n*), where *n* is the input size. Many new algorithms claim some advantage in terms of a constant improvement; however, their real performances are unknown. The paper presents a systematic and synergic study on the efficiency of different implementations of FFT programming. In particular, we propose different ways to program FFT. Different code techniques such as recursive, iterative, TFBBGM [12], TFRM [13] with further expansion are explored. An extensive experiment is conducted for input size from 512 to 2048 points. Some important findings are obtained on 20 FFT codes on existing major DSP architectures. Further manual tuning optimizations are possible. Contrary to the common belief that recursive program is slower, we find that recursive programs are not necessarily slower for commonly used FFT. Instead its performances are determined by many other factors. Our comparative study constitutes the first attempt to understand the real performance evaluation of different approaches.

The paper is organized as follows. In Section II, we give the preliminaries of DIF/DIT FFT algorithms and code techniques such as TFRM, TFBBGM, etc. Section III describes the implementations of twenty FFT codes. Experiment results are shown in Section IV. Section V concludes the paper.

II. Preliminaries

We first present the basic ideas of DIT FFT and DIF FFT. Then we describe two code structures: iterative and recursive codes. Two methods of TFRM and TFBBGM are introduced to reduce the number of memory references due to twiddle factor.

A. DIT and DIF FFT

The DFT of discrete signal x[n] can be directly computed as $X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}$, $k = 0,1,\dots,N-1$ (1) where $W_N^{nk} = e^{-j(2\pi/N)nk}$, $x[n]$ and $X[k]$ are sequences of complex numbers, and j^2 = -1.

The basic ideas of DIT and DIF FFT algorithms are to decompose the input sequence $x[n]$ and output sequence $X[n]$ of (1) into smaller sequences. E.g. the radix-2 DIT and DIF FFT algorithms are obtained by splitting the input sequence $x[n]$ and output sequence $X[n]$ into odd and even indexed elements. Figures 1(b) and 2(b) show the computation diagrams of the DIT and DIF algorithms, respectively.

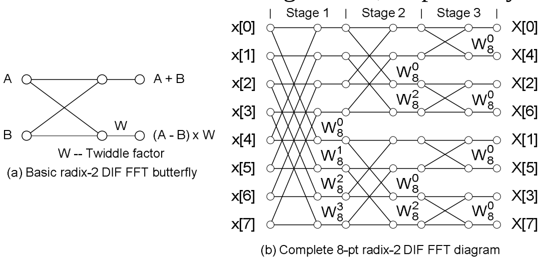
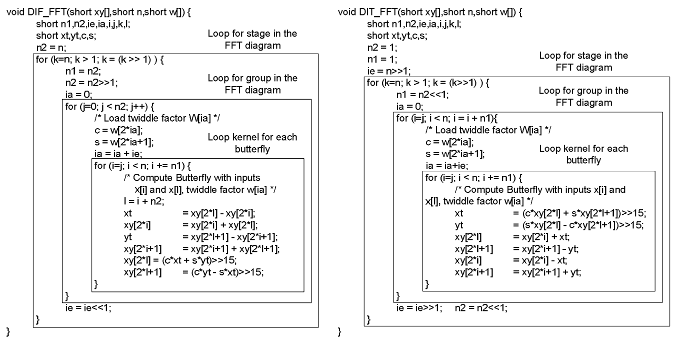


Figure 2. The 8-pt radix-2 DIF FFT diagram

The butterflies are computed according to the index order of the stages and groups by partitioning the radix-2 DIT and DIF FFT diagrams. Within the same group, the butterflies are computed from top to bottom. Figure 3(a) shows the iterative C code implementation of *n*-points radix-2 DIF FFT algorithm taken from TI's DSP library [14], where *n* is given as an input parameter to the C code. Figure 3(b) shows the corresponding iterative C code implementation of *n*-points radix-2 DIT FFT algorithm.



```
void DIF_FFT(short xy[],short n,short w[]) {
        short n1,n2,ie,ia,i,j,k,l;
        short xt,yt,c,s;
        n2 = n;
        for (k=n; k > 1; k = (k >> 1) ) {
                n1 = n2;
                n2 = n2>>1;
                ia = 0;
                for (j=0; j < n2; j++) {
                        /* Load twiddle factor W[ia] */
                        c = w[2*ia];
                        s = w[2*ia+1];
                        ia = ia + ie;
                        for (i=j; i < n; i += n1) {
                                /* Compute Butterfly with inputs
                                   x[i] and x[l], twiddle factor w[ia] */
                                l = i + n2;
                                xt          = xy[2*l] - xy[2*i];
                                xy[2*i]     = xy[2*i] + xy[2*l];
                                yt          = xy[2*l+1] - xy[2*i+1];
                                xy[2*i+1]   = xy[2*i+1] + xy[2*l+1];
                                xy[2*l] = (c*xt + s*yt)>>15;
                                xy[2*l+1]   = (c*yt - s*xt)>>15;
                        }
                }
                ie = ie<<1;
        }
}
```

Loop for stage in the FFT diagram
Loop for group in the FFT diagram
Loop kernel for each butterfly

(a) Conventional C code of radix-2 DIF FFT from [15]

```
void DIT_FFT(short xy[],short n,short w[]) {
        short n1,n2,ie,ia,i,j,k,l;
        short xt,yt,c,s;
        n2 = 1;
        n1 = 1;
        ie = n>>1;
        for (k=n; k > 1; k = (k>>1) ) {
                n1 = n2<<1;
                ia = 0;
                for (i=j; i < n; i = i + n1){
                        /* Load twiddle factor W[ia] */
                        c = w[2*ia];
                        s = w[2*ia+1];
                        ia = ia+ie;
                        for (i=j; i < n; i += n1) {
                                /* Compute Butterfly with inputs x[i] and
                        x[l], twiddle factor w[ia] */
                                xt          = (c*xy[2*l] + s*xy[2*l+1])>>15;
                                yt          = (s*xy[2*l] - c*xy[2*l+1])>>15;
                                xy[2*l]     = xy[2*i] + xt;
                                xy[2*l+1]   = xy[2*i+1] - yt;
                                xy[2*i]     = xy[2*i] - xt;
                                xy[2*i+1]   = xy[2*i+1] + yt;
                        }
                }
                ie = ie>>1;    n2 = n2<<1;
        }
}
```

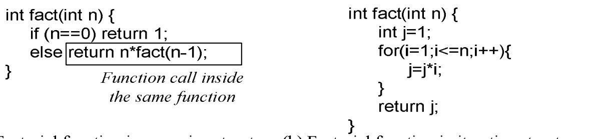
(b) Similar C code of radix-2 DIT FFT

Figure 3. The C code of radix-2 DIF FFT and radix-2 DIT FFT.

The C code in Figure 3 shows a three-loop iterative structure: 1) the outermost loop, the *k*-loop, counts the stages, loops for $\log_2 N$ times; 2) the second loop, the *j*-loop, counts the groups within each stage and decides which twiddle factor to load; 3) the innermost loop, the *i*-loop counts the number of butterflies within each group. Variables *k* and *j* indicate the stages and group number, respectively. Variables *i* and *l* indicate the upper and lower input indexes of the butterfly computed by the innermost loop, respectively. Variable *ia* indicates the index of the twiddle factor to be loaded.

B. Recursive and Iterative Code Structures

Recursion plays an elegant role in solving problems in design and analysis of computer algorithms and complexity theory [15]. A complex problem can be decomposed into smaller problems of the same structure. Figure 4(a) shows the recursive code of factorial function. It is only the multiplication process that determines the code complexity. Hence, the complexity of the original problem can be decreased.



```
int fact(int n) {
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

Function call inside the same function

```
int fact(int n) {
      int j=1;
      for(i=1;i<=n;i++){
            j=j*i;
      }
      return j;
}
```

(a) Factorial function in recursive structure (b) Factorial function in iterative structure

Figure 4. Example of recursive code and iterative code.

As Figure 4(a) illustrates, the recursive code structure involves function call inside the same function, thus it needs memory stack operation to fulfill this task. Due to the expensiveness of memory operation in clock cycles, the recursive code structure also increases the number of clock cycles to some extent.

Iterative code structure is the common structure in which the same phase of code is executed multiple times. Figure 4(b) shows the iterative code of factorial function. It will not incur memory stack operation due to function call within the same function, hence it requires fewer clock cycles than recursive code. However, the iterative structure is more complex than recursive structure in code size, which also makes it require more clock cycles to some extent.

We explore the overall performance of these two code structures by performing thorough experiment on various iterative and recursive FFT codes.

C. TFBBGM

The TFBBGM (twiddle-factor-based butterfly grouping method) groups the butterflies in the radix-2 FFT diagram according to the twiddle factor. Each twiddle factor is loaded only once in the computation order, thus the number of redundant memory references due to twiddle factor in conventional radix-2 DIF FFT algorithm can be reduced. Since there are $\log_2 N$ twiddle factors for a *N*-points radix-2 DIF FFT algorithm, the computation requires only $\log_2 N$ steps.

From Figure 2(b), we have some important observations. There are *N*/2 different twiddle factors in the first stage of radix-2 *N*-points DIF FFT diagram, expressed as W_N^m , where *m* = 0, 1, 2, 3, …, *N*/2-1. The twiddle factors of odd m among *N*/2 twiddle factors in the first stage do not occur in later stages. At any stage *s*, the twiddle factor for any butterfly is $W_N^{\left(n \bmod \frac{N}{2^{s-1}}\right) \times 2^{s-1}}$, so it is clear that $\left(n \bmod \frac{N}{2^{s-1}}\right) \times 2^{s-1}$ will not be odd when *s* is greater than 1. Thus, butterflies of twiddle factor W_N^m with *m* = 1, 3, 5, … , *N*/2-1 can be grouped and thus *N*/4 butterflies are computed in the first stage of the TFBBGM.

In the *s*-th stage, except those butterflies computed in the first stage, butterflies with twiddle factors that do not occur after Stage *s* of radix-2 *N*-points DIF FFT diagram as in Figure 2(b) will be computed.

There are *N*/4 butterflies in Stage s, *N*/8 butterflies in Stage *s*-1, *N*/16 butterflies in Stage *s*-2, …, and $N/2^{s+1}$ butterflies in Stage 1. Twiddle factors of the corresponding butterflies are W_N^m , where $m = 2^{s-1}$, $3\times2^{s-1}$, $5\times2^{s-1}$, … , $(N/2^{s-1})\times2^{s-1}$. Particularly, when *s* is 2, *N*/4 butterflies in Stage 2 and *N*/8 butterflies in Stage 1 of radix-2 DIF FFT diagram in Figure 2

(b) will be computed in stage 2 of the new method. Twiddle factors of these butterflies are W_N^m where m = 2, 6, 10, … , (N/2)-2.

The last stage computes totally N-1 butterflies with twiddle factor W_N^0, together in the radix-2 N-point DIF FFT diagram.

By using these new stages, the new method loads each twiddle factor only once in the computation. We redraw the computation diagram of radix-2 DIF FFT as shown in Figure 2(b) into Figure 5. From the new diagram, it is easy to see that totally N-1 butterflies with twiddle factor W_N^0 =1 will be computed without multiplication, which is conducive to reduce the number of clock cycles.

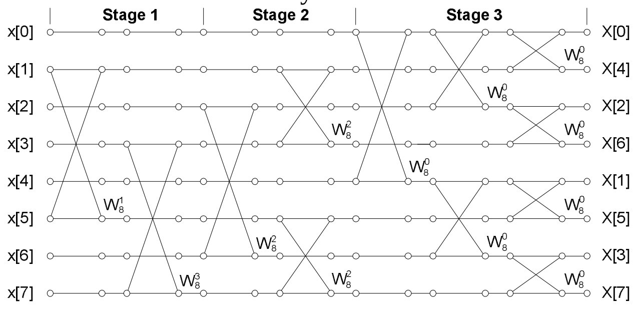


Figure 5. The radix-2 DIF FFT diagram with TFBBGM.

Similarly, TFBBGM can also be applied to radix-2 DIT FFT with TFBBGM. But due to the difference between radix-2 DIT FFT and radix-2 DIF FFT, butterflies with twiddle factor W_{16}^0 need to be grouped and computed before butterflies with other twiddle factors are grouped and computed. In the later step s, TFBBGM groups and computes butterflies with twiddle factor W_N^m, where $m = N/2^s$, $3\times N/2^s$, $5\times N/2^s$, … , $(2^{s-1}-1)\times N/2^s$. Figure 6 shows the radix-2 DIT FFT diagram redrawn by grouping the butterflies with identical twiddle factors.

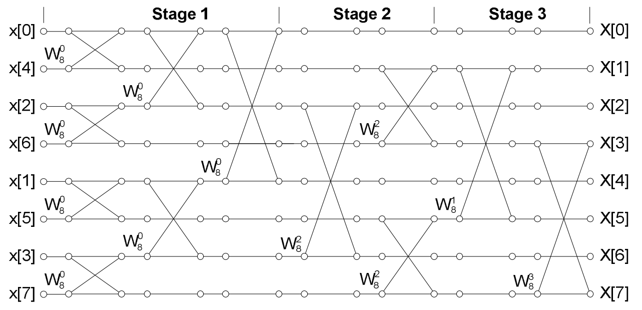


Figure 6. The 8-pts radix-2 DIT FFT diagram with TFBBGM.

D. TFRM

Based on the complex properties of the twiddle factor, TFRM (Twiddle factor Reduce Method) can reduce the number of twiddle factor to be referenced. For example, W_8^3 can be replaced by $-jW_8^1$ in Figure 1 and Figure 2 by using the property of the complex number, here is the derivation procedure:

$$W_8^3 = e^{-j\times(2\pi/8)\times 3} = e^{-j\times(2\pi/8)\times 2}\cdot e^{-j\times(2\pi/8)\times 1} = -j\cdot W_8^1$$

The similar derivation can be applied to W_8^2. Hence, only W_8^0 and W_8^1 are actually required in the computation of 8 points radix-2 DIF and DIT FFT.

By using the property of complex number, the twiddle factor has the following property:

$$W_N^m = \begin{cases} W_N^m & m\in[0,N/4) \\ W_N^{N/4}\cdot W_N^{m-N/4} = -j\cdot W_N^{m-N/4} & m\in[N/4,N/2) \\ W_N^{N/2}\cdot W_N^{m-N/2} = -W_N^{m-N/2} & m\in[N/2,3N/4) \\ W_N^{3N/4}\cdot W_N^{m-3N/4} = j\cdot W_N^{m-3N/4} & m\in[3N/4,N) \end{cases}$$

Also, as observed from radix-2 DIF FFT diagram in Figure 2(b), we know any single butterfly in the Stage s of radix-2 N-point DIF FFT can be illustrated in the diagram format as in Figure 7.

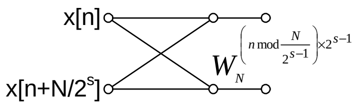

x[n]
x[n+N/2^s]
$W_N^{\left(n \bmod \frac{N}{2^{s-1}}\right)\times 2^{s-1}}$

Figure 7. Single butterfly in Stage s of N-pt radix-2 DIF FFT.

The butterfly with $x[n]$ as the upper input and $x[n+N/2s]$ as the lower input uses $W_N^{\left(n \bmod \frac{N}{2^{s-1}}\right)\times 2^{s-1}}$ as twiddle factor. For example, in the stage 1 of Figure 1, the butterfly with the upper input $x[2]$ and $x[2+8/21]$, namely $x[6]$ as lower input uses twiddle factor $W_8^{\left(2 \bmod \frac{8}{2^{1-1}}\right)\times 2^{1-1}} = W_8^2$.

Hence, we have the following property for the radix-2 DIF FFT diagram:

Two butterflies in stage s as illustrated in Figure 8(a) can be computed by loading one twiddle factor W_N^m, where $m = (n \bmod \frac{N}{2^{s-1}})\times 2^{s-1}$ as illustrated in Figure 8(b).

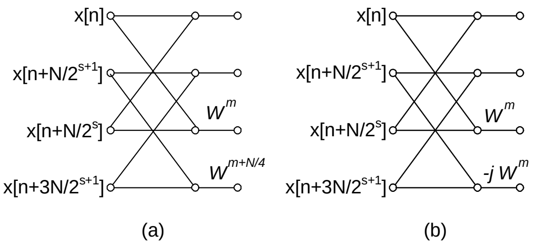


Figure 8. Two butterflies computation using one twiddle factor in DIF FFT diagram.

Likewise, after applying TFRM to DIT FFT diagram, we have the similar property for radix-2 DIT FFT diagram:

Two butterflies in stage s as illustrated in Figure 9(a) can be computed by loading one twiddle factor W_N^m, where $m = (n \bmod 2^s)\times 2^{s-1}$as illustrated in Figure 9(b).

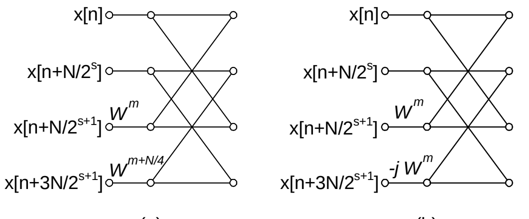


Figure 9. Two butterflies computation using one twiddle factor in DIT FFT diagram.

III. FFT Implementations

TFRM and TFBBGM can reduce the number of memory references due to twiddle factors, thus decreasing the number of clock cycles. However, they also increase the code complexity to some extent, thus increasing the number of clock cycles. In this section, these methods are implemented with iterative and recursive codes respectively. To perform thorough study on different FFT code performance, we further expand those iterative codes applied with TFBBGM to make them in different iterative structures which require fewer loops but take more code space.

A. *DIF/DIT FFT with TFBBGM and TFRM*

TFRM reduces the memory references due to twiddle factor by half in each stage of computation, and computes the butterflies in last two stages without twiddle factors [13], so the number of memory references due to twiddle factors is reduced to ($\log_2 N$-2) × N/4. TFBBGM reduces the numberof memory references due to twiddle factor by grouping the butterflies with identical twiddle factor and the butterflies with twiddle factor W_N^0 do not need twiddle factor to complete multiplication, so the number of memory references due to twiddle factor is N/2-1.

After applying TFRM and TFBBGM together, the number of memory references due to twiddle factor in the new radix-2 N-points DIF FFT code will be reduced to N/4-1. Figure 10 shows the computation diagram of a 8-points radix-2 DIF-FFT with TFRM and TFBBGM. Since the butterflies computed in the last step do not need twiddle factor due to the fact that W_N^0 = 1, only 1 twiddle factors are required.

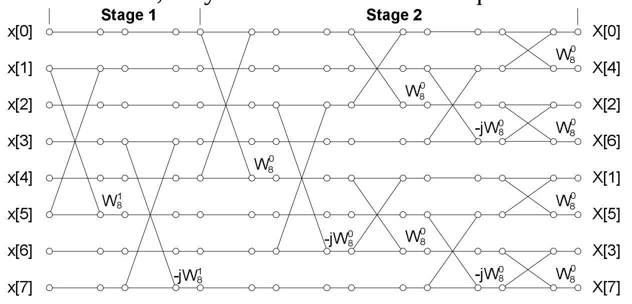


Figure 10. Radix-2 DIF FFT diagram with TFBBGM and TFRM.

We also apply TFBBGM and TFRM to radix-2 DIT FFT. Like radix-2 DIF FFT with TFBBGM and TFRM, the number of memory references due to twiddle factor will be greatly reduced. However, due to the difference of DIF and DIT FFT, the input of radix-2 DIT FFT should be in bit-reversed order before TFBBGM and TFRM are applied together to radix-2 DIT FFT. Figure 11 shows the computation diagram of 8-points radix-2 DIT FFT with TFBBGM and TFRM.

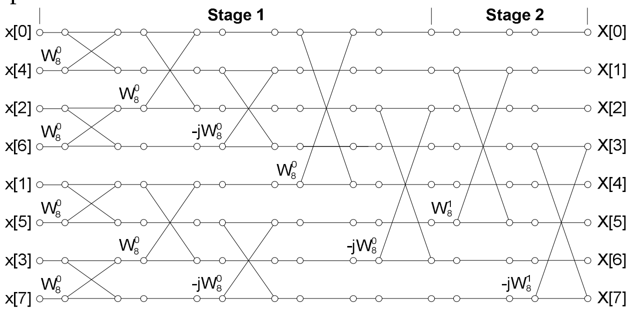


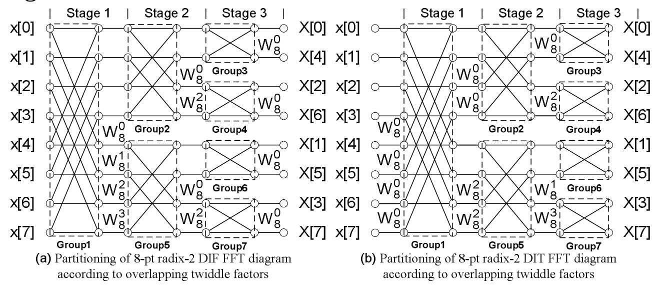
Figure 11. Radix-2 DIT FFT diagram with TFBBGM and TFRM.

From the diagram, it is easy to see that only 4 twiddle factor are used during computation. Since W_N^0 = 1, twiddle factor will not be loaded in Stage 1. Hence, only 1 twiddle factors are loaded during the computation.

B. *Recursive DIF and DIT FFT implementation*

N-point radix-2 DIF and DIT FFT can be programmed in recursive structures which will decrease the code complexity to some extent. Like iterative code structure, TFBBGM and TFRM can still be applied to the recursive code implementation of N-point radix-2 DIF and DIT FFT.

Recursive C code implementation is also based on the FFT diagram, but the way butterflies are grouped is different from Iterative C code. Figure 12 shows the computation and the partitioning of the 8-points radix-2 DIF and DIT FFT diagram.


Stage 1
Stage 2
Stage 3
Group1
Group2
Group3
Group4
Group5
Group6
Group7
(a) Partitioning of 8-pt radix-2 DIF FFT diagram according to overlapping twiddle factors
(b) Partitioning of 8-pt radix-2 DIT FFT diagram according to overlapping twiddle factors
Figure 12. Partitioning of DIF and DIT FFT diagram according to overlapping twiddle factors.

The diagram is also partitioned into stages and the butterflies in the same stage are grouped according to their positions, not according to the same twiddle factor. Thus, butterflies which overlap with each other are grouped. The butterflies in Stage s of N-point radix-2 DIF and DIT FFT are divided into 2^{s-1} groups.

All butterflies are computed according to the group order. E.g. butterflies in group 3 are computed after butterflies in group 2 are computed, and butterflies in the same group are computed from top to bottom. In Figure 12(a), twiddle factors are increased by value 2^{s-1} for butterflies in the same group in Stage s.

We apply TFRM, TFBBGM on recursive DIF FFT and recursive DIT FFT respectively, thus we have recursive DIF FFT with TFRM, recursive DIF FFT with TFBBGM, recursive DIT FFT with TFRM and recursive DIT FFT with TFBBGM. Since TFBBGM and TFRM can be applied together, we apply both of them to get recursive DIF FFT with TFBBGM & TFRM and recursive DIT FFT with TFBBGM & TFRM. Experiments of these codes with different input sizes are performed to get the clock cycle data in the following Section.

C. *DIF and DIT FFT with expansion*

In order to completely study how the code techniques reflect the performance, we further expand the iterative DIF and DIT FFT applied with TFBBGM (as illustrated in computation diagram in Figure 5 and 6) into 3 steps: For iterative DIF FFT with TFBBGM in Figure 5, the first step computes butterflies with twiddle factor as W_N^m, where m is odd number; the second step groups the butterflies with identical twiddle factors and compute them in only two loops, the third step computes the butterflies with twiddle factor W_N^0 =1 , namely computes butterflies without multiplication. For iterative DIT FFT with TFBBGM in Figure 6, the first step computes butterflies without multiplication like third step in iterative DIF FFT with TFBBGM, the second step is similar to the second step as iterative DIF FFT with TFBBGM with only the order changed, and the third step is the similar to step 1 in DIF, with order changed. Thus we get another two codes: iterative DIF FFT with TFBBGM in 3 steps and iterative DIT FFT with TFBBGM in 3 steps.

We also apply the same further expansion to iterative DIF FFT with TFBBGM & TFRM and iterative DIT FFT with TFBBGM & TFRM to get iterative DIF FFT with TFBBGM & TFRM in 3 steps and iterative DIT FFT with TFBBGM & TFRM in 3 steps.

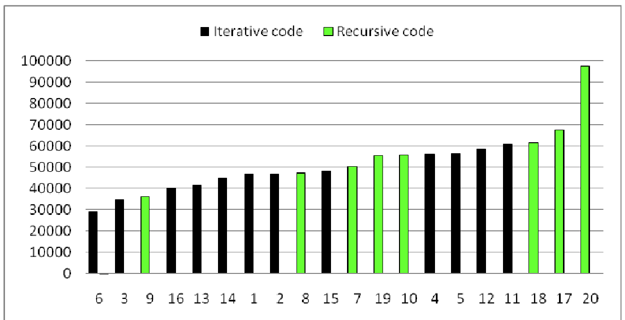
VI. Performance Evaluation

We have made total 20 different FFT codes based on different integrations of iterative, recursive, TFBBGM, TFRM with further expansion. Due to the space limitation, 20 FFT codes are identified using shortcut. Code 1: Iterative DIF FFT, Code 2: Iterative DIF FFT with TFBBGM, Code 3: Iterative DIF FFT with TFBBGM in 3 steps, Code 4: Iterative DIF FFT with TFRM, Code 5: Iterative DIF FFT with TFBBGM and TFRM, Code 6: Iterative DIF FFT with TFBBGM and TFRM in 3 Steps, Code 7: Recursive DIF FFT, Code 8: Recursive DIF FFT with TFBBGM, Code 9: Recursive DIF FFT with TFRM, Code 10: Recursive DIF FFT with TFBBGM and TFRM, Code 11: Iterative DIT FFT, Code 12: Iterative DIT FFT with TFBBGM, Code 13: Iterative DIT FFT with TFBBGM in 3 steps, Code 14: Iterative DIT FFT with TFRM, Code 15: Iterative DIT FFT with TFBBGM and TFRM, Code 16: Iterative DIT FFT with TFBBGM and TFRM in 3 Steps, Code 17: Recursive DIT FFT, Code 18: Recursive DIT FFT with TFBBGM, Code 19: Recursive DIT FFT with TFRM and Code 20: Recursive DIT FFT with TFBBGM and TFRM.

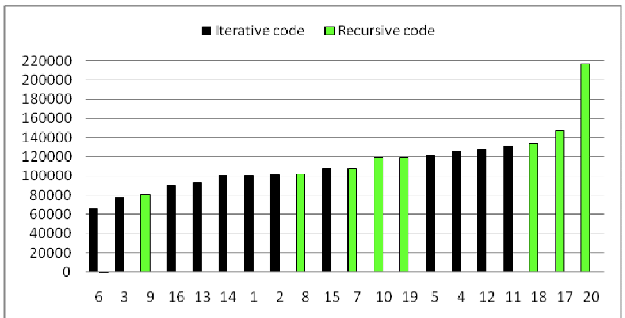
These 20 codes are tested on 4 major DSP processors in the industry: TI TMS320C64xx, ARM ARM7TDMI processor, ADSP-TS101 TigerSHARC processor, and freescale SC3400 StarCore DSP.

After testing each FFT code on 4 DSP processors at different input sizes of 512, 1024 and 2048, we sort the 20 FFT codes for each DSP processor in terms of clock cycle. As illustrated in Figure 13, we find that the recursive codes always interleave between iterative codes. This is contrary to the belief that recursive code is always slower than iterative code. Due to space limitation, we use digit 1 to 20 to identify code 1 to code 20 in Figure 13.

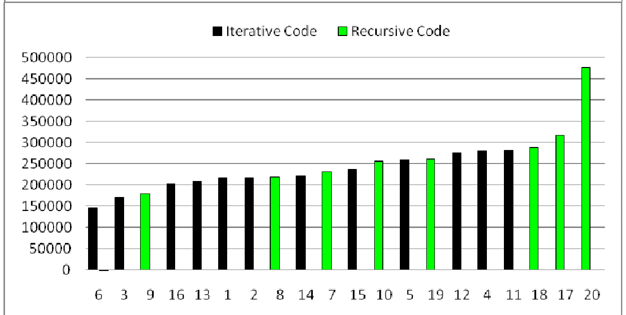
We also compare the slowest code and the fastest code within recursive FFT and iterative FFT codes. But for the space limitation, we could not list the table to compare their speed, here the fastest code is compared with slowest code through Figure 13 only. For instance, we compare the number of clock cycle of the fastest recursive code with the slowest recursive code when platform is freescale SC3400 StarCore DSP and the input is 2048-pts, we find the fastest one is 2.1 faster than slowest one, and similar result applies to the rest cases. Such as 20 FFT codes tested on the platform TI TMS320C64xx processor at input size of 512, the fastest iterative code is 2.08 times faster than the slowest itetative code. Hence, it is clear that he code techniques presented in this paper can increase the FFT code execution speed around 2 times within iterative codes or recursive codes.



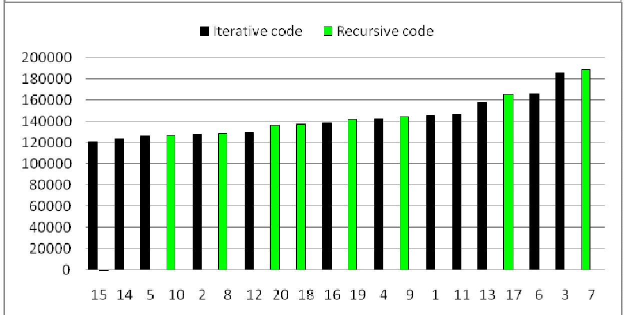
20 FFT codes on TI TMSC320C64x (sorted by 512 pts)



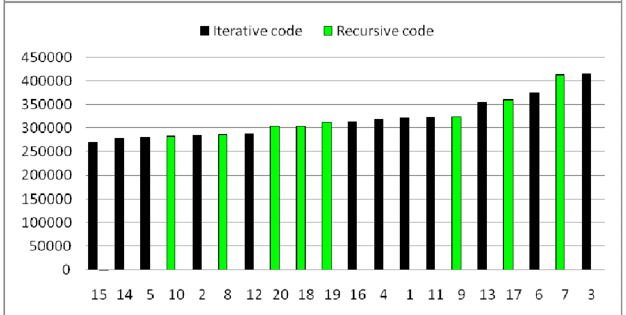
20 FFT codes on TI TMSC320C64x (sorted by 1024 pts)



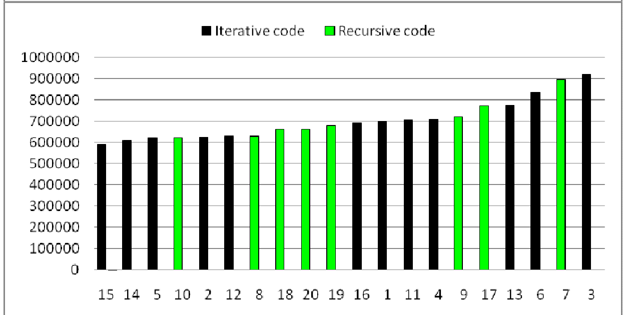
20 FFT codes on TI TMSC320C64x (sorted by 2048 pts)



20 FFT codes on ARM7TDMI(sorted by 512 pts)



20 FFT codes on ARM7TDMI(sorted by 1024 pts)



20 FFT codes on ARM7TDMI(sorted by 2048 pts)

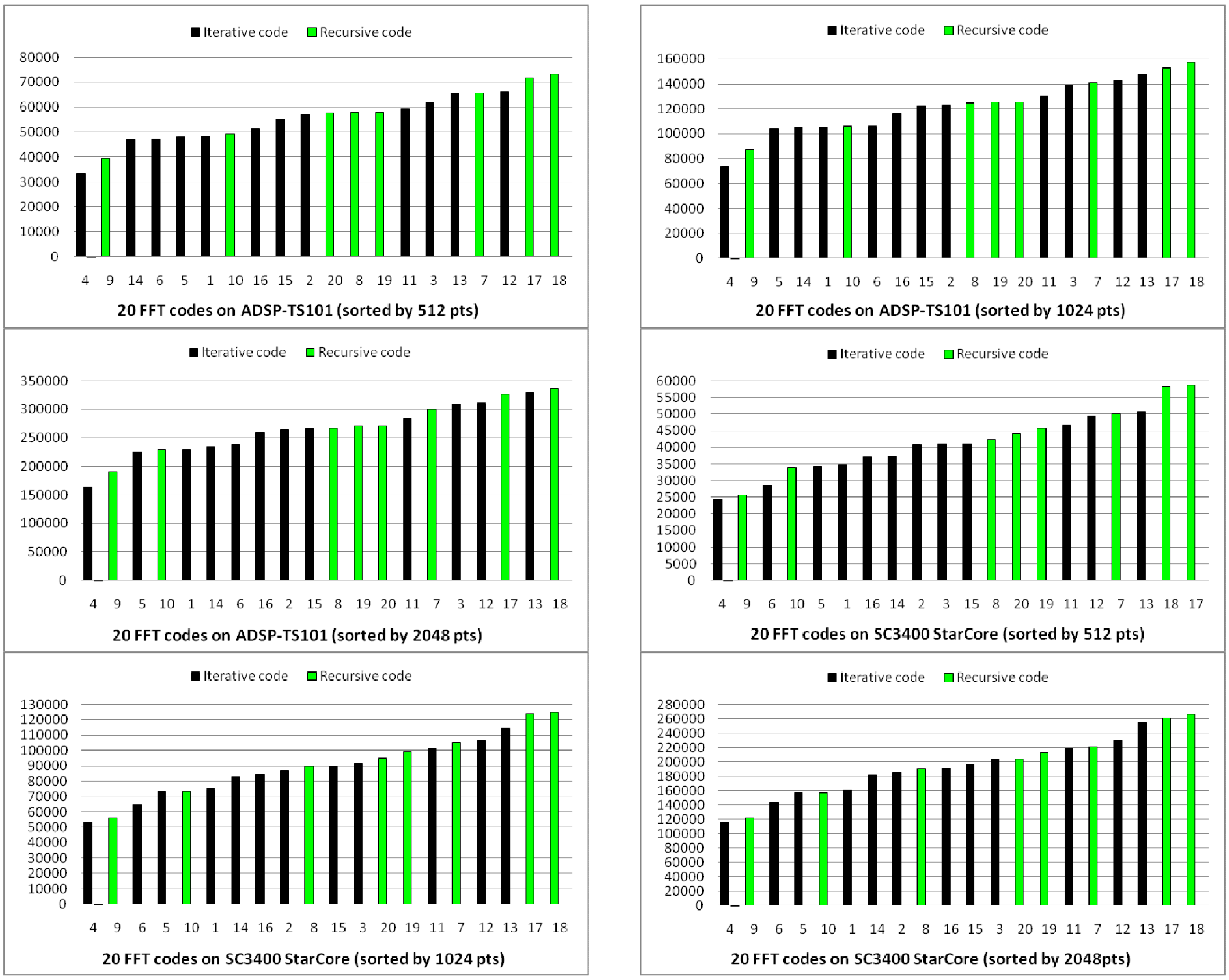


Figure 13. 20 FFT codes' Clock cycles on 4 DSP architectures at input of 512, 1024 and 2048 pts.

V. Conclusion

We presented a systematic and synergic study on the efficiency of different FFT software implementations. Different code techniques such as recursive, iterative, TFBBGM, TFRM with further expansion were explored. An extensive experiment was conducted for input from 512 to 2048 points. We implemented 20 FFT codes on 4 different DSP processors. Contrary to the common belief that recursive programs are slower than iterative programs, we find that recursive programs are not necessarily slower for commonly used FFT. Instead its performances are determined by many factors. Also we find the code techniques presented in this paper can increase FFT code execution speed 2 times for both iterative codes and recursive codes.

References

[1] C.S. Burrus and T.W. Parks, "DFT/FFT and Convolution Algorithms and Implementation," *NY John Wiley & Sons*, 1985.

[2] A. V. Oppenheim and C. M. Rader, 2nd ed., Discrete-Time Signal Processing. Upper Saddle River, NJ: Prentice-Hall, 1989.

[3] J.W. Cooley and J.W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Compu.*, vol. 19, pp. 297-301, 1965.

[4] G.D. Bergland, "A Radix-Eight Fast-Fourier Transform Subroutine for Real-Valued Series," *IEEE Trans. Electroacoust.*, vol. 17, no. 2, pp. 138-144, June 1969.

[5] R.C. Singleton, "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Trans. Audio Electroacoust.*, vol. 1, no. 2, pp. 93-103, June 1969.

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

[7] D. Takahashi, "An Extended Split-Radix FFT Algorithm," *IEEE Signal Processing Letters*, vol. 8, no. 5, pp. 145-147, May 2001.

[8] A. R. Varkonyi-Koczy, "A Recursive Fast Fourier Transform Algorithm," IEEE Trans. Circuits and Systems, II, vol. 42, pp. 614-616, Sep. 1995.

[9] A. Saidi, "Decimation-in-Time-Frequency FFT Algorithm," *Proc. ICAPSS*, pp. III:453-456, April 1994.

[10] B.M. Baas, "A low-power, high-performance, 1024-point FFT processor" IEEE J.Solid-State Circuits, vol. 34, issue 3, pp.380-387. March 1999.

[11] Mathwork Inc., Matlab function reference – FFT, http://www.mathworks.com/access/helpdesk/help/techdoc/ref/fft.shtml?BB=1

[12] Y. Jiang, Y. Tang, and Y. Wang, "Twiddle-factor-based FFT algorithm with reduced memory access," *Proc. IDPDS*, pp. 653-660, 2002.

[13] Y. Tang, L. Qian, Y. Wang, and Y. Jiang, "Twiddle Factor Based Memory Reduction Method for FFT Implementation on DSP." to be published

[14] Texas Instrument, "TMS320C64x DSP Library Programmer's Reference (Rev. B)," SPRU565A, Oct. 23, 2003.

[15] Tanner.R. "A recursive approach to low complexity codes" IEEE Transactions on Information Theory.