An Investigation of Cooley-Tukey Decompositions for the FFT

Gavin Haentjens

May 18, 2000

Abstract

The goals of the research discussed in this report are to determine the impact of different Cooley-Tukey decompositions on the performance of computer programs that compute the FFT, evaluate different methods for finding the most efficient decompositions, and determine the characteristics of the most efficient decompositions. Experiments are conducted using three different FFT programs and three dynamic programming (DP) methods for searching for efficient decompositions. The results show that even for FFTs of sizes under 2^{11}, the runtime for the average decomposition may be up to three times the runtime for the optimal decomposition. The results also show that a basic implementation of DP performs as well as an exhaustive search at finding fast decompositions for FFTs of sizes up to 2^{10}, and that the simple DP performs as well as two more sophisticated versions of DP for FFT sizes up to 2^{20}. Furthermore, the results show that for an out-of-place implementation of the FFT, right-expanded decompositions are optimal because they require memory storage only for the input and output data arrays, whereas other decompositions require additional temporary storage. Moreover, the results show that for an in-place implementation of the FFT, balanced decompositions are optimal if the algorithm is iterative, and right-expanded decompositions are optimal if the algorithm is recursive.

Acknowledgements

I would like to thank Prof. José M. F. Moura, Prof. Jeremy Johnson, Dr. Markus Püschel, and David Sepiashvili for all of their assistance. This research was funded by the Defense Advanced Research Projects Agency (DARPA).

Contents

1 Introduction **1**

1.1 Nature of the problem . . . 1

1.1.1 Cooley-Tukey Decompositions . . . 1

1.1.2 Multitude of Decompositions for a Given Size FFT . . . 3

1.2 The Cooley-Tukey FFT Algorithm . . . 4

1.2.1 The Tensor Product . . . 4

1.2.2 DFT Matrices . . . 4

1.2.3 Twiddle Factor Matrices . . . 5

1.2.4 Stride Permutation Matrices . . . 5

1.2.5 Tensor Product Formulation of the Cooley-Tukey FFT Algorithm . 6

1.2.6 Arithmetic Cost . . . 7

1.3 Research Goals . . . 8

2 Searching for Efficient Cooley-Tukey Decompositions **10**

2.1 Introduction . . . 10

2.2 Methodology . . . 10

2.2.1 An Original FFT Program . . . 10

2.2.2 Exhaustive Search . . . 12

2.2.3 Dynamic Programming Search . . . 13

2.2.4 N-Best Dynamic Programming Search . . . 13

2.2.5 Stride Sensitive Dynamic Programming Search . . . 14

2.2.6 Measuring Runtimes . . . 14

2.3 Results . . . 15

2.4 Conclusions . . . 19

3 Experiments with FFTW **24**

3.1 Introduction . . . 24

3.1.1 Motivation and Goals . . . 24

3.1.2 Introduction to FFTW . . . 25

3.2 Methodology . . . 28

3.2.1 Modifications to FFTW . . . 28

3.2.2 Modification to the Stride Sensitive DP Search Strategy . . . 29

3.2.3 Adding a Penalty for Twiddle Nodes . . . 30

3.2.4 Removing a Penalty for LCINL Nodes . . . 30

3.2.5 Removing All Twiddle Factor Multiplications . . . 31

3.3 Results . . . 31

3.4 Conclusions . . . 33

4 Experiments with Egner's FFT Program 37
4.1 Introduction 37
4.1.1 Motivation and Goals 37
4.1.2 Introduction to Egner's Program 38
4.2 Methodology 42
4.2.1 Improving the Order of the Twiddle Factors 42
4.2.2 Making the Program More Recursive 43
4.2.3 Adding the No-Twiddle Codelets from FFTW 43
4.2.4 Untransposing the Cooley-Tukey Algorithm 43
4.2.5 Adding the Twiddle Codelets from FFTW 44
4.3 Results 45
4.4 Conclusions 47

5 Conclusions and Future Work 52

Bibliography 54

Chapter 1

Introduction

1.1 Nature of the problem

In many areas of digital signal processing, there is great demand for highly efficient implementations of the discrete Fourier transform (DFT). With the advent of the Cooley-Tukey (CT) algorithm [1] for the *fast Fourier transform* (FFT), the efficiency of DFT implementations was increased considerably. The algorithm reduces the arithmetic cost of the computation of a DFT of size N from $O(N^2)$ to $O(N\ \log(N))$ by recursively decomposing the problem of size N into smaller problems that can be solved more efficiently. However, there are many different ways to decompose a DFT. Although all of the possible decompositions have equal arithmetic cost, they vary significantly in terms of data access patterns. Therefore, since data access costs have become increasingly expensive relative to the cost of floating point operations on modern computer architectures, it has become increasingly important to choose a good decomposition.

1.1.1 Cooley-Tukey Decompositions

In [1] it is shown that a DFT of size N can be computed using the following three steps:

1. Compute R DFTs of size S, where $N = R \cdot S$.
2. Multiply the result by *twiddle factors*, which are N-th roots of unity.
3. Compute S DFTs of size R.

Essentially, the FFT is computed by *decomposing* a problem of size N into smaller problems of size R and S. The procedure can be applied recursively, meaning that the problems of size R and S can be further decomposed into smaller problems, as long as they are not prime. The problems that are not further decomposed are referred to as *base cases* of the recursive algorithm. The smallest possible base cases for the CT algorithm are DFTs of size two. As shown in [1], each time a problem of size N is decomposed into smaller problems, there is a reduction in the number of complex additions and multiplications required to compute the DFT. Therefore, in order to compute a DFT with minimal arithmetic cost, the problem must be decomposed as fully as possible. To simplify matters, only DFTs of sizes that are powers of two are considered in this report. Therefore, all of the problems can be decomposed so that all base cases are DFTs of size two.

Cooley-Tukey decompositions are represented graphically in this report using a binary tree structure. An example of a single tree is shown in Figure 1.1. The *nodes* of the tree are the circles in the figure. The number listed in each node is an exponent that indicates a DFT size. In this report, all lower-case letters indicate exponents with a base of two. That is $N = 2^n$, $R = 2^r$, etc. The tree in Figure 1.1 represents a DFT of size 2^n that is decomposed into problems of size 2^r and 2^s, where $n = r + s$.

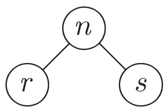


Figure 1.1: Tree Representation of a Cooley-Tukey Decomposition

Two important terms used when discussing the trees are *parent nodes* and *child nodes.* If a DFT of size 2^n is decomposed into DFTs of size 2^r and 2^s, then the node labeled n is the parent node and nodes labeled r and s are the child nodes. Figure 1.2 illustrates the concept of *parent* and *child* nodes. It is important to note that a decomposition with r as the left child and s as the right child is distinct from a decomposition with s as the left child and r as the right child, unless $r = s$. It is also important to note that child nodes can also be considered parent nodes if they have children of their own.

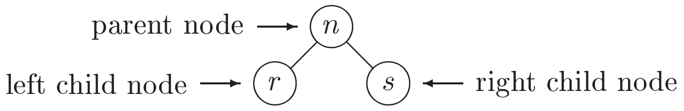


Figure 1.2: Parent and Child Nodes for a Single Decomposition

Three other important terms used when discussing the trees are *root node*, *leaf node*, and *top-level decomposition.* The root node is simply the node that has no parents; that is, the node at the top of a tree. In Figure 1.3, the root node of each tree is the node labeled with a 4. The leaf nodes are the nodes that do not have any children. In Figure 1.3, the leaf nodes are the nodes labeled with 1's. The top-level decomposition refers to the root node of a tree and its two child nodes. Figure 1.3 shows two distinct decompositions that have the same top-level decomposition. In each tree, the top-level decomposition consists of a node labeled with a 4 as the parent, a node labeled with a 1 as the left child, and a node labeled with a 3 as the right child.

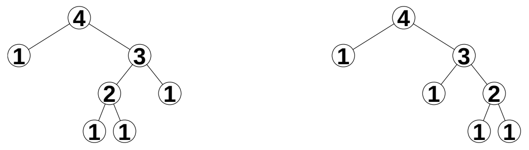


Figure 1.3: Two Distinct Decompositions with the Same Top-Level Decomposition

Three other important terms used when discussing the trees are *left-expanded*, *right-expanded*, and *balanced.* In a left-expanded decomposition, only one of the leaf nodes is a left-child. Simlarly, in a right-expanded decomposition, only one of the leaf nodes is a right-child. Finally, in a balanced decomposition, the two subtrees that are children of the root node are identical. In this report a decomposition is described to as "somewhat balanced" if those subtrees are similar but not identical. The decomposition on the far left of Figure 1.4 is left-expanded, while the decomposition on the far right of the figure is right-expanded, and the decomposition in the center is balanced.

1.1.2 Multitude of Decompositions for a Given Size FFT

There are numerous ways to fully decompose a DFT of size 2^n. A DFT is "fully decomposed" if none of the base cases can be further decomposed. In the case of a DFT of size 2^n, "fully decomposed" means that all of the base cases are 2-point DFTs. As shown in Figure 1.4, for an FFT of size 2^4 there are five possible *full decompositions*, that is, decompositions in which none of the base cases can be further decomposed. The number $T(n)$ of distinct full

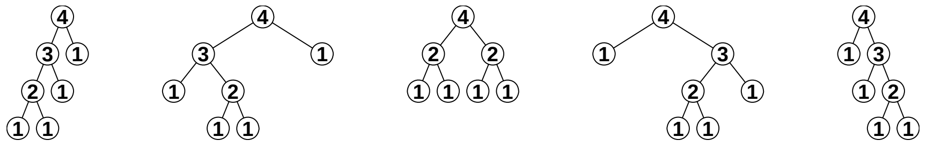


Figure 1.4: All Possible Decompositions for an FFT of Size 2^4

decompositions for a DFT of size 2^n is given be the recurrence

$$T(n) = \begin{cases} 1, & n = 1 \\ \sum_{i=1}^{n-1} T(i)T(n-i), & n > 1. \end{cases} \tag{1.1}$$

This formula follows from the fact that all decompositions for an FFT of size 2^n can be generated by combining all possible decompositions for an FFT of size 2^i as the left subtree of the root node, with all possible decompositions for an FFT of size 2^{n-i} as the right subtree of the root node, $i = 1, \ldots, n-1$.

It can be shown using standard methods that $T(n)$ is $\Theta(\frac{5^n}{n^{3/2}})$. That is, the function $\frac{5^n}{n^{3/2}}$ is an asymptotically tight bound for $T(n)$. More specifically, there exist constants C_1, C_2, and K, such that

$$0 \leq C_1 \frac{5^n}{n^{3/2}} \leq T(n) \leq C_2 \frac{5^n}{n^{3/2}} \tag{1.2}$$

for all $n \geq K$. The values of C_1, C_2, and K are fixed for the function $T(n)$ and do not depend on n. Table 1.1 lists the number of distinct decompositions for FFTs of size of 2^1 through 2^{20}. As shown in the table, there are over 10^9 distinct decompositions for an FFT of size 2^{20}.

n	$T(n)$	n	$T(n)$	n	$T(n)$	n	$T(n)$
1	1	6	42	11	16,796	16	9,694,845
2	1	7	132	12	58,786	17	35,357,670
3	2	8	429	13	208,012	18	129,644,790
4	5	9	1,430	14	742,900	19	477,638,700
5	14	10	4,862	15	2,674,440	20	1,767,263,190

Table 1.1: Distinct Decompositions $T(n)$ for an FFT of size 2^n

In order to minimize arithmetic cost, all of the computer programs discussed in this report compute FFTs using decompositions that are fully expanded; that is, all of the FFTs are computed by decomposing the problem all the way down to two-point DFTs. However, most of the trees that represent the algorithms used by these computer programs have DFTs larger than two as the leaf nodes. Leaf nodes with DFTs larger than two simply indicate that a computer program switches to unrolled code at that point in the decomposition instead of continuing to make recursive function calls. The unrolled code is highly optimized and is used to compute FFTs as large as 2^6 in the programs discussed in this report. Using unrolled code is advantageous because it eliminates the overhead costs due to recursive function calls. However, using unrolled code is not advantageous for large size FFTs, since the code in that case is very long and competes with data for space in the level-two cache on most modern architectures.

Although using unrolled code for small size FFTs increases the efficiency of the FFT computation, it also significantly increases the number of distinct decompositions for a given size FFT. Table 1.2 lists the number of decompositions when the largest allowable base case is a size 2^6 DFT. As shown in the table, for an FFT of size 2^{20}, there are over 10^{11} distinct decompositions.

n	$T(n)$	n	$T(n)$	n	$T(n)$	n	$T(n)$
1	1	6	188	11	222,996	16	389,840,001
2	2	7	730	12	973,554	17	1,775,559,054
3	5	8	2,947	13	4,298,656	18	8,131,303,622
4	15	9	12,224	14	19,162,618	19	37,419,457,862
5	51	10	51,777	15	86,125,378	20	172,951,682,679

Table 1.2: Distinct Decompositions When the Largest Base Case is a Size 2^6 FFT

1.2 The Cooley-Tukey FFT Algorithm

In this subsection we present the Cooley-Tukey FFT algorithm. First, some notation must be introduced.

1.2.1 The Tensor Product

The tensor product of a matrix $\mathbf{A}$ of size $R \times S$ with a matrix $\mathbf{B}$ of size $M \times N$ is the matrix of size $MR \times NS$

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \cdots & a_{1,S}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \cdots & a_{2,S}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{R,1}\mathbf{B} & a_{R,2}\mathbf{B} & \cdots & a_{R,S}\mathbf{B} \end{bmatrix}. \tag{1.3}$$

It should be noted that the coarse structure of $(\mathbf{A} \otimes \mathbf{B})$ is determined by $\mathbf{A}$ while the fine structure is determined by $\mathbf{B}$.

1.2.2 DFT Matrices

The DFT matrix of order N, denoted by $\mathbf{F}_N$, is defined as (see [9])

$$\begin{aligned} \mathbf{F}_N &= \begin{bmatrix} w_N^{0\cdot 0} & w_N^{0\cdot 1} & \cdots & w_N^{0\cdot(N-1)} \\ w_N^{1\cdot 0} & w_N^{1\cdot 1} & \cdots & w_N^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_N^{(N-1)\cdot 0} & w_N^{(N-1)\cdot 1} & \cdots & w_N^{(N-1)\cdot(N-1)} \end{bmatrix}, \\ w_N &= e^{-j\,2\pi/N}. \end{aligned} \tag{1.4}$$

In particular we have

$$\mathbf{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{1.5}$$

hence $\mathbf{y} = \mathbf{F}_2 \cdot \mathbf{x}$ requires only two additions. A straightforward implementation of the transform $\mathbf{y} = \mathbf{F}_N \cdot \mathbf{x}$ would require $O(N^2)$ arithmetic operations. The Cooley-Tukey algorithm, which we will introduce in Subsection 1.2.5, reduces the arithmetic cost to $O(N \log N)$, if N is sufficiently composite.

1.2.3 Twiddle Factor Matrices

A twiddle factor matrix $\mathbf{T}_S^{RS}$, where $N = RS$, is a diagonal matrix with N-th roots of unity, called *twiddle factors*, on the diagonal. $\mathbf{T}_S^{RS}$ is given by

$$\mathbf{T}_S^{RS} = \bigoplus_{K=0}^{R-1} \begin{bmatrix} w_N^0 & & & \\ & w_N^1 & & \\ & & \ddots & \\ & & & w_N^{S-1} \end{bmatrix}^K, \tag{1.6}$$

where the symbol $\oplus$ indicates a direct sum. As shown in Equation (1.6), many of the exponents of the term w_N are zero, resulting in twiddle factors that are one. Specifically, the first matrix in the direct sum contains S ones, and the $R-1$ subsequent matrices in the direct sum each contain a single one. Therefore, the total number of twiddle factors in $\mathbf{T}_S^{RS}$ that are equal to one is given by

$$W_1(R,S) = S + R - 1 \tag{1.7}$$

Since the total number of twiddle factors in $\mathbf{T}_S^{RS}$ is RS, we can compute the total number of twiddle factors in $\mathbf{T}_S^{RS}$ that are *not* equal to one as

$$\begin{aligned} W_{n1}(R,S) &= RS - W_1(R,S) && (1.8) \\ &= RS - S - R + 1. && (1.9) \end{aligned}$$

The term $W_{n1}(R,S)$ is used later in this chapter to count the number of complex multiplications required to apply $\mathbf{T}_S^{RS}$, omitting multiplications by one.

1.2.4 Stride Permutation Matrices

The action of a stride permutation matrix $\mathbf{L}_R^{RS}$ on an arbitrary column vector $\mathbf{x}$ of length RS is to rearrange $\mathbf{x}$ as follows:

$$\mathbf{L}_R^{RS} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{RS} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_R \end{bmatrix}, \quad \mathbf{y}_i = \begin{bmatrix} x_{0 \cdot R+i} \\ x_{1 \cdot R+i} \\ \vdots \\ x_{(S-1) \cdot R+i} \end{bmatrix}. \tag{1.10}$$

In words, the i-th chunk of $\mathbf{y}$ contains the S elements of $\mathbf{x}$ collected at stride R, starting with the i-th element of $\mathbf{x}$, $i = 1, \ldots, R$.

The formula for a stride permutation matrix is the following:

$$\mathbf{L}_R^{RS} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_R \end{bmatrix}, \quad \mathbf{A}_i = \begin{bmatrix} \mathbf{e}_{0\cdot R+i}^{RS} \\ \mathbf{e}_{1\cdot R+i}^{RS} \\ \vdots \\ \mathbf{e}_{(S-1)\cdot R+i}^{RS} \end{bmatrix}, \tag{1.11}$$

where $\mathbf{e}_j^{RS}$ denotes a row vector of length RS with a one in the j-th coordinate and zeros elsewhere. It is important to note that stride permutation matrices are not actually constructed in the computer programs discussed in this report. Stride permutation matrices are only used in equations to indicate that elements in a vector are to be reordered. As is explained later in this report, the reordering of elements is rarely done as a stand-alone procedure. That is, when data in arrays are accessed at stride, processing is usually done on the data immediately, instead of after the entire array is reordered. This practice eliminates the need for an extra pass through the data, which could incur extra cache misses.

As shown in [9], if an $RS \times 1$ vector $\mathbf{c}$ can be represented as a tensor product of an $R \times 1$ vector $\mathbf{a}$ with an $S \times 1$ vector $\mathbf{b}$, then an elegant way to represent the action of $\mathbf{L}_R^{RS}$ on $\mathbf{c}$ is

$$\mathbf{L}_R^{RS}\mathbf{c} = \mathbf{L}_R^{RS}(\mathbf{a} \otimes \mathbf{b}) = \mathbf{b} \otimes \mathbf{a}. \tag{1.12}$$

1.2.5 Tensor Product Formulation of the Cooley-Tukey FFT Algorithm

In the previous sections, the tensor product operator, and DFT, twiddle factor, and stride permutation matrices were introduced. With these concepts in place, we now introduce the tensor product formulation of the Cooley-Tukey FFT algorithm. As shown in [9], the Cooley-Tukey FFT algorithm for a single decomposition can be represented in tensor product notation as

$$\mathbf{F}_{RS} = (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}, \tag{1.13}$$

where $\mathbf{I}_S$ is an identity matrix of size $S \times S$. It is important to note that all four terms in Equation (1.13) are sparse matrices. If this were not the case, then the decomposition would not be any more efficient than computation of the DFT by definition.

Two important terms in Equation (1.13) are $(\mathbf{I}_R \otimes \mathbf{F}_S)$ and $(\mathbf{F}_R \otimes \mathbf{I}_S)$. We now present examples to illustrate what these terms represent. First consider the operation

$$\mathbf{y} = (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{x}, \tag{1.14}$$

where $\mathbf{x}$ and $\mathbf{y}$ are both vectors of size $RS \times 1$. The operation in Equation (1.14) fills the i-th chunk of $\mathbf{y}$ with the S-point DFT of the i-th chunk of $\mathbf{x}$, $i = 1, \ldots, R$. Two examples of $\mathbf{y} = (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{x}$ are illustrated in Figure 1.5.

Next consider the operation

$$\mathbf{y} = (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{x}, \tag{1.15}$$

where $\mathbf{x}$ and $\mathbf{y}$ are again vectors of size $RS \times 1$. The operation in Equation (1.15), fills $\mathbf{y}$ at stride S starting from element i with the R-point DFT of the elements of $\mathbf{x}$ taken at stride S starting from element i, $i = 1, \ldots, S$. Two examples of $\mathbf{y} = (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{x}$ are illustrated in Figure 1.6.

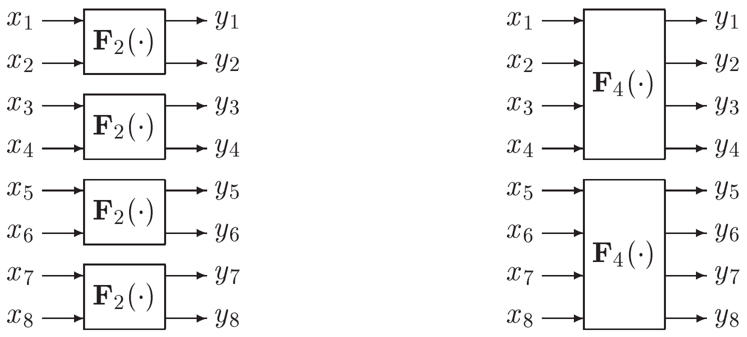


Figure 1.5: Left: $\mathbf{y} = (\mathbf{I}_4 \otimes \mathbf{F}_2)\mathbf{x}$, Right: $\mathbf{y} = (\mathbf{I}_2 \otimes \mathbf{F}_4)\mathbf{x}$

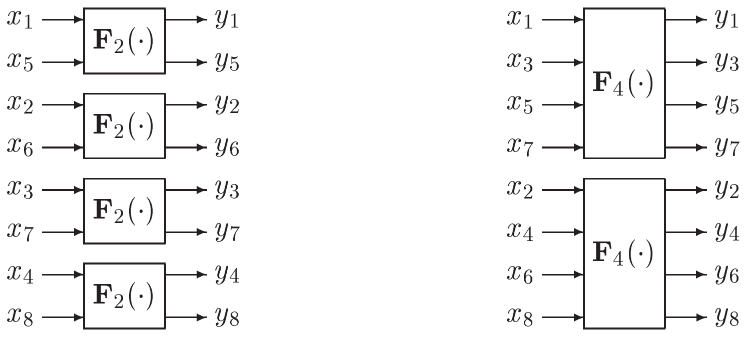


Figure 1.6: Left: $\mathbf{y} = (\mathbf{F}_2 \otimes \mathbf{I}_4)\mathbf{x}$, Right: $\mathbf{y} = (\mathbf{F}_4 \otimes \mathbf{I}_2)\mathbf{x}$

1.2.6 Arithmetic Cost

The *arithmetic cost* of a decomposition is the number of complex additions and multiplications required to compute an FFT using that decomposition. As is shown next, the arithmetic cost of the Cooley-Tukey algorithm for an FFT of two-power size is independent of the particular decomposition that is used. To the author's knowledge, the following two lemmas and proofs do not exist in the literature.

Lemma 1 *The number $A(w)$ of complex additions required to compute a F_{2^n} that is fully decomposed is independent of the decomposition and is given by*

$$A(n) = n \cdot 2^n. \tag{1.16}$$

Proof We will prove Equation (1.16) using induction on n. First consider the base case $n = 1$. As shown in Equation (1.5), multiplication by $\mathbf{F}_2$ requires $2 = 1 \cdot 2^1$ additions, as desired. Next we will count the number of complex additions required to compute $\mathbf{F}_{n+1}$, where $n \geq 1$, using an arbitrary decomposition $n + 1 = (n + 1 - s) + s$, where $1 \leq s < n$. The tensor product formulation for the decomposition is given by

$$\mathbf{F}_{2^{n+1}} = (\mathbf{F}_{2^{n+1-s}} \otimes \mathbf{I}_{2^s})\mathbf{T}_{2^s}^{2^{n+1}}(\mathbf{I}_{2^{n+1-s}} \otimes \mathbf{F}_{2^s})\mathbf{L}_{2^{n+1-s}}^{2^{n+1}}. \tag{1.17}$$

The terms in Equation (1.17) that involve additions are the matrices $\mathbf{F}_{2^{n+1-s}}$ and $\mathbf{F}_{2^s}$. As shown in the equation, the $\mathbf{F}_{2^{n+1-s}}$ is applied 2^s times, and the $\mathbf{F}_{2^s}$ is applied 2^{n+1-s} times. Using the induction hypothesis for $n + 1 - s$ and s, we compute the number of additions required as:

$$2^s \cdot A(n + 1 - s) + 2^{n+1-s} \cdot A(s)$$

$$\begin{aligned}
&= 2^s \cdot (n+1-s) \cdot 2^{n+1-s} + 2^{n+1-s} \cdot s \cdot 2^s \\
&= 2^{n+1} \cdot (n+1-s) + 2^{n+1} \cdot s \\
&= (n+1) \cdot 2^{n+1} \\
&= A(n+1),
\end{aligned}$$

which completes the proof.

Lemma 2 *The number $M(n)$ of complex multiplications required to compute a F_{2^n} that is fully decomposed is independent of the decomposition and is given by*

$$M(n) = (n-2) \cdot 2^{n-1} + 1 \tag{1.18}$$

Proof We will again use induction on n to prove Equation (1.18). First we consider the base case $n = 1$. As shown in Equation (1.5), multiplication by $\mathbf{F}_2$ requires $0 = (1-2) \cdot 2^0 + 1$ multiplications, as desired. Next we will count the number of complex multiplications required to compute $\mathbf{F}_{n+1}$, where $n \geq 1$, using an arbitrary decomposition $n+1 = (n+1-s)+s$, where $1 \leq s < n$. The terms in the tensor product formulation given in Equation (1.17) that involve multiplications are the DFT matrices $\mathbf{F}_{2^{n+1-s}}$ and $\mathbf{F}_{2^s}$ and the twiddle matrix $\mathbf{T}_{2^s}^{2^{n+1}}$. As shown in the equation, the $\mathbf{F}_{2^{n+1-s}}$ is applied 2^s times, the $\mathbf{F}_{2^s}$ is applied 2^{n+1-s} times, and the $\mathbf{T}_{2^s}^{2^{n+1}}$ is applied once. Using the induction hypothesis for $n+1-s$ and s, we compute the number of multiplications required as:

$$2^s \cdot M(n+1-s) + 2^{n+1-s} \cdot M(s) + W_{n1}(2^{n+1}, 2^s), \tag{1.19}$$

where $W_{n1}(2^{k+1}, 2^s)$ is the number of twiddle factors in $\mathbf{T}_{2^s}^{2^{k+1}}$ that are *not* equal to one, given in Equation (1.9). Substituting Equation (1.9) in Equation (1.19), we compute the number of multiplications required as:

$$\begin{aligned}
& 2^s \left((n+1-s-2)2^{n+1-s-1} + 1\right) + 2^{n+1-s}\left((s-2)2^{s-1}+1\right) + \\
& 2^{n+1} - 2^s - 2^{n+1-s} + 1 \\
=\; & (n-s-1) \cdot 2^n + 2^s + (s-2) \cdot 2^n + 2^{n+1-s} + \\
& 2^{n+1} - 2^s - 2^{n+1-s} + 1 \\
=\; & (n-3) \cdot 2^n + 2^{n+1} + 1 \\
=\; & (n-3) \cdot 2^n + 2 \cdot 2^n + 1 \\
=\; & (n-1) \cdot 2^n + 1 \\
=\; & (n+1-2) \cdot 2^{n+1-1} + 1 \\
=\; & M(n+1),
\end{aligned}$$

which completes the proof.

1.3 Research Goals

The goals of this research are to answer several questions about the effect of Cooley-Tukey decompositions on the performance of computer programs for computing FFTs.

Chapter 2 of this report addresses the following questions:

1. What guidelines should be followed to create an efficient computer program for performing FFTs?

2. How does the run time of an efficient FFT program vary over all possible Cooley-Tukey decompositions for a given size FFT?

3. What is the best way to search the large space of Cooley-Tukey decompositions for efficient decompositions?

4. What topological features are common to the most efficient decompositions?

Chapter 3 addresses the following questions:

1. Why does the successful FFT program FFTW [5] consider only right-expanded decompositions?

2. Can the performance of FFTW be improved by expanding the program to consider arbitrary decompositions?

Chapter 4 addresses the following questions:

1. How does the performance of an FFT program that computes FFTs in-place using explicit permutations compare with the performance of FFTW, which computes FFTs out-of-place and does not perform explicit permutations?

2. What effect does transposing the Cooley-Tukey algorithm have on the performance of an FFT implementation?

3. How does the performance of an iterative implementation of the Cooley-Tukey algorithm compare with the performance of a fully-recursive implementation of the algorithm?

4. What decompositions are optimal for iterative and recursive implementations of a computer program for computing in-place FFTs?

Finally, Chapter 5 reviews the principal conclusions from Chapters 2 through 4 and discusses future directions for the research discussed in this report.

Chapter 2

Searching for Efficient Cooley-Tukey Decompositions

2.1 Introduction

In the previous chapter the Cooley-Tukey algorithm for the FFT was introduced and briefly explained. It was noted that the number of distinct decompositions for an FFT of size 2^k increases exponentially with k, and that all decompositions for a given size FFT have equivalent arithmetic complexity. It was also noted that the different decompositions have different data access patterns, and that the variation in data access patterns should translate to a variation in performance among FFTs computed using the different decompositions.

In this chapter, the exact impact of the different data access patterns on computer program runtimes is studied by writing a program to compute FFTs corresponding to arbitrary decompositions and timing the program as it computes FFTs using all of the decompositions. In addition, a basic dynamic programming strategy for drastically reducing the size of the search space of decompositions is applied to find efficient decompositions. The concept of dynamic programming is introduced in [2], and an explanation of dynamic programming applied to FFTs is provided in [6]. Two more sophisticated versions of dynamic programming are introduced in this chapter and are compared to the basic dynamic programming. Finally, the features of the optimal decompositions for FFTs of size up to 2^{20} are investigated.

2.2 Methodology

2.2.1 An Original FFT Program

In order to investigate various Cooley-Tukey decompositions for the FFT, an original FFT program was written. The program was implemented in both Fortran and C++, in order to ensure that results were not tied to one particular programming language. The Fortran implementation is called FFT-Fortran and the C++ implementation is called FFT-C. The program borrows ideas from a successful FFT program called *FFTW* [5]. However, the program has an important distinction from FFTW in that it can compute FFTs corresponding to arbitrary decompositions while FFTW can compute FFTs corresponding to right-expanded decompositions only. The FFT program borrows the following ideas from FFTW:

1. Use highly optimized code modules for the base cases of the recursion in the Cooley-Tukey algorithm.
2. Create a plan structure that allows each decomposition to be represented by a unique *plan*.
3. Precompute the twiddle factors that will be needed for a decomposition and store them with the plan.
4. Create an executor program that computes an FFT according to a plan without performing explicit permutations.

The Small FFT Code Modules

Highly optimized code modules were written to compute FFTs of sizes 2^1, 2^2, 2^3, and 2^4. The code modules were used for the base cases of the recursion in the Cooley-Tukey algorithm. The code modules are discussed in detail in [8] and are based on algorithms given in [7].

The Plan Structure

The FFT program uses a plan structure that allows each decomposition to be represented by a unique *plan*. A plan is simply a linked list of nodes that contains information needed for different steps in an FFT computation. A diagram of a single plan node is shown in Figure 2.1.

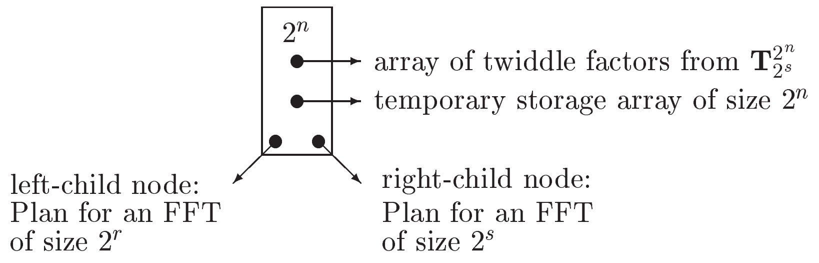

2^n
array of twiddle factors from $\mathbf{T}_{2^s}^{2^n}$
temporary storage array of size 2^n
left-child node: Plan for an FFT of size 2^r
right-child node: Plan for an FFT of size 2^s

Figure 2.1: A Single Plan Node for an FFT of Size 2^n

As shown in Figure 2.1, each plan node contains the size of the FFT at a particular step in the decomposition and four pointers. The first pointer points to an array of twiddle factors that are the elements from the diagonal of the twiddle matrix $\mathbf{T}_{2^s}^{2^n}$. The twiddle factors are computed as a plan is being created and are stored with the plan, so that they do not have to be computed when an FFT is being computed. The second pointer in a plan node points to an array that is used to store intermediate results in an FFT computation. This array is necessary because at certain steps in the FFT computation, FFTs must be computed at an input stride that is different from the output stride. The third and fourth pointers in a plan node point to other plan nodes. These plan nodes specify how the children of the current node are to be decomposed. Figure 2.2 shows a decomposition for an FFT of size 2^5, and Figure 2.3 shows the plan that would be created to represent that decomposition. As shown in Figure 2.3, the pointers in all of the leaf nodes do not point to anything. The reason the pointers do not point to anything is that the only information that is needed at the base cases of the recursion is the size of the FFT that is to be computed.

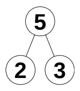


Figure 2.2: A Simple Decomposition

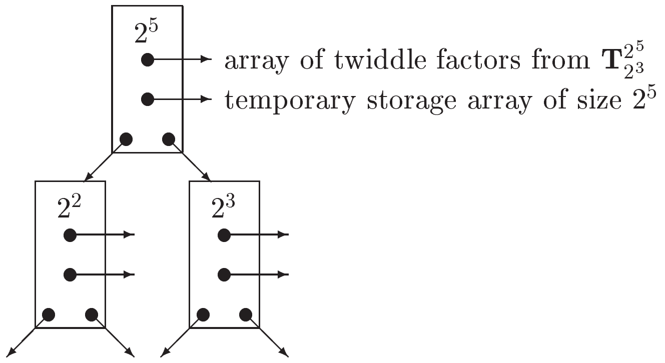

2^5
array of twiddle factors from $\mathbf{T}_{2^3}^{2^5}$
temporary storage array of size 2^5
2^2
2^3

Figure 2.3: Plan Corresponding to the Decomposition in Figure 2.2

The Executor Subroutine

The FFT program uses a subroutine called the *executor* to compute an FFT corresponding to a plan. An important feature of the executor subroutine that is borrowed from FFTW is that the stride permutations in the FFT computation are not performed explicitly. Instead, the stride permutations are implemented by computing the FFTs for the subproblems at various input and output strides. That is, to compute

$$\mathbf{y} \leftarrow (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x}, \tag{2.1}$$

the elements in vector $\mathbf{x}$ are not first completely permuted by $\mathbf{L}_R^{RS}$ before the $\mathbf{F}_S$'s are applied. Instead, the entire operation in Equation (2.1) is performed by computing m $\mathbf{F}_S$'s in which the input is the elements of $\mathbf{x}$ taken at stride R and the output is stored in y at stride 1.

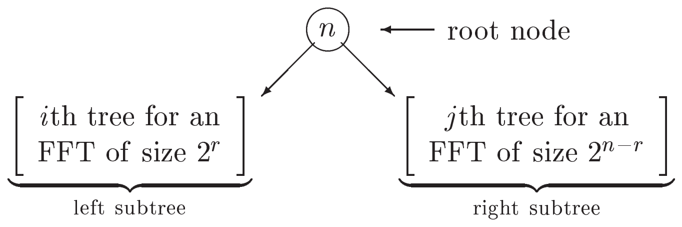
The executor computes an FFT by evaluating each of the nodes in a plan and making computations based on information contained in the nodes. The executor begins by evaluating the root node, and recursively visits all of the nodes in the plan. When the executor encounters an *internal* node, that is, a node that is not a leaf, the executor computes

$$\begin{aligned} \mathbf{t} &\leftarrow \mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} && (2.2) \\ \mathbf{y} &\leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{t}, && (2.3) \end{aligned}$$

where $\mathbf{x}$ is the array containing the input for the transform, $\mathbf{y}$ is the array for the output of the transform, and $\mathbf{t}$ is the temporary storage array associated with the current node. When the executor routine encounters a leaf node, it simply makes a call to the small FFT code module that computes an FFT of the size indicated by the node.

2.2.2 Exhaustive Search

In order to compute FFTs using all possible decompositions, it is necessary to first generate the decompositions and save them to a file. The decompositions are generated using the following algorithm. To generate all decompositions for an FFT of size 2^n, it is necessary to first generate all possible decompositions for FFTs of size 2^1 through 2^{n-1}. For an FFT of size 2^1, there is only one "decomposition", since the problem cannot be further decomposed. For an FFT of size 2^n, $n > 1$, a decomposition tree is generated for every possible combination of one of the decomposition trees for an FFT of size 2^r with one of the decomposition trees for an FFT of size 2^{n-r}, $r = 1, \ldots, n-1$, as shown in Figure 2.4. A parameter n_L


n
root node
ith tree for an FFT of size 2^r
jth tree for an FFT of size 2^{n-r}
left subtree
right subtree

$$\begin{array}{lcl} i & = & 1, \ldots, T(r) \\ j & = & 1, \ldots, T(n-r) \\ r & = & 1, \ldots, n-1 \\ T(r) & = & \text{number of decompositions} \\ & & \text{for an FFT of size } 2^r \end{array}$$

Figure 2.4: Algorithm for Generating All Decompositions for an FFT of Size 2^n, $n > n_L$

must be used to specify the largest allowable size for a leaf in any decomposition tree. n_L is always set to equal the size of the largest FFT for which a small code module is available. When the decompositions for an FFT of size 2^n are being generated, an extra decomposition must be generated if $n \leq n_L$. This decomposition corresponds to the case where the problem is not further decomposed and contains only one node—the root node.

2.2.3 Dynamic Programming Search

The dynamic programming search is performed by filling a list of length n_{max} with the optimal *top-level* decompositions for FFTs of sizes 2^1 through $2^{n_{max}}$. After the DP search is complete, the n-th entry in the list contains the optimal top-level decomposition for an FFT of size 2^n. By necessity, the list is filled from smallest FFT sizes to largest. To determine the optimal top-level decomposition for an FFT of size 2^n, all possible top-level decompositions for that size FFT are timed. If no small DFT code module exists for an FFT of size 2^n, then there are $n-1$ decompositions to time in order to find the optimal top-level decomposition for that size FFT; otherwise there are n decompositions to time.

2.2.4 N-Best Dynamic Programming Search

The n-best DP search is performed by filling a $n_{max} \times b_{max}$ table with the b best top-level decompositions for FFTs of sizes 2^1 through $2^{n_{max}}$. When the n-best DP search is complete, the entry at row n, column b of the table contains the b-th best top-level decomposition for an FFT of size 2^n. The table is filled one row at a time, from the smallest size FFT to the largest. To determine the b best top-level decompositions for an FFT of size 2^n, all top-level decompositions must be timed.

2.2.5 Stride Sensitive Dynamic Programming Search

The stride sensitive DP search is performed by filling n_{max} tables of size $n_{max} \times n_{max}$ with the best top-level decompositions for FFTs of sizes 2^1 through $2^{n_{max}}$ performed at input strides of 2^0 through $2^{n_{max}-1}$ and output strides of 2^0 through $2^{n_{max}-1}$. When the stride sensitive DP search is complete, the entry at row i, column j of the n-th table contains the optimal decomposition for an FFT of size 2^n performed at input stride 2^i and output stride 2^j.

Once the tables have been filled completely, one has all of the information needed to perform the optimal decompositions at various input and output strides. However, in order to use this information, it is necessary to know the correct input and output strides for each subproblem in a decomposition. In order to see how the input and output strides for each subproblem in a decompsition are computed, consider the simple decomposition shown in Figure 2.5. This decomposition could be a subproblem in a larger decomposition. Assume

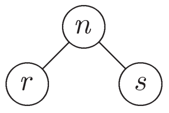


Figure 2.5: A Single Decomposition

that for the FFT corresponding to node N, the input stride is P_{in} and the output stride is P_{out}. The input and output strides for the child nodes in Figure 2.5 are listed in Table 2.1.

	right-child node	left-child node
input stride:	$2^r \cdot P_{in}$	2^r
output stride:	1	$2^r \cdot P_{out}$

Table 2.1: Input and Output Strides for the Child Nodes in Figure 2.5

As shown in the table, the input stride accumulates mulitplicatively with each deeper right-child in the recursion, while the output stride accumulates multiplicatively with each deeper left-child in the recursion. The table also shows that there is no accumulation of strides for the output of the right-child computation, since the output of this computation is stored in the temporary storage array that is dedicated for node n.

2.2.6 Measuring Runtimes

All runtimes are measured using a software timer explained in [8]. The timer uses the *clock()* function in the C library to make the timings. The timer repeats each routine it measures until at least one second has expired before making a single timing. The reason for making the minimum experiment duration one second is that the precision of the *clock()* function is 0.01 second, and a reasonably acceptable value for the maximum relative error in each timing is 1%. Each time the timing routine is called, it makes several timings of a routine to be measured, and returns the average of all the timings that were left after any outlying times were removed. The standard deviation of timing estimates was typically about 5%.

2.3 Results

Figures 2.6 and 2.7 show the runtimes for all decompositions using FFT-Fortran and FFT-C for FFT sizes of 2^5 through 2^{10}. The runtimes are sorted from shortest to longest. As shown in the figures, the runtimes for FFT-Fortran are fairly evenly distributed between the minimum and maximum, while for FFT-C the runtimes are slightly more concentrated near the maximum.

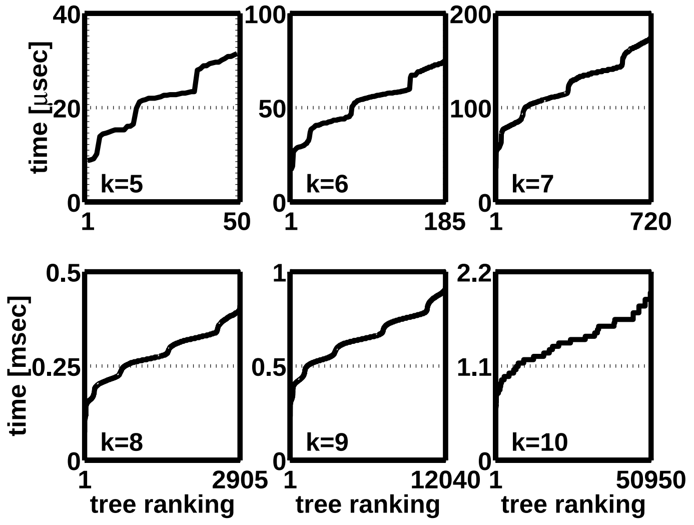


Figure 2.6: Sorted Runtimes for All Decompositions Using FFT-Fortran

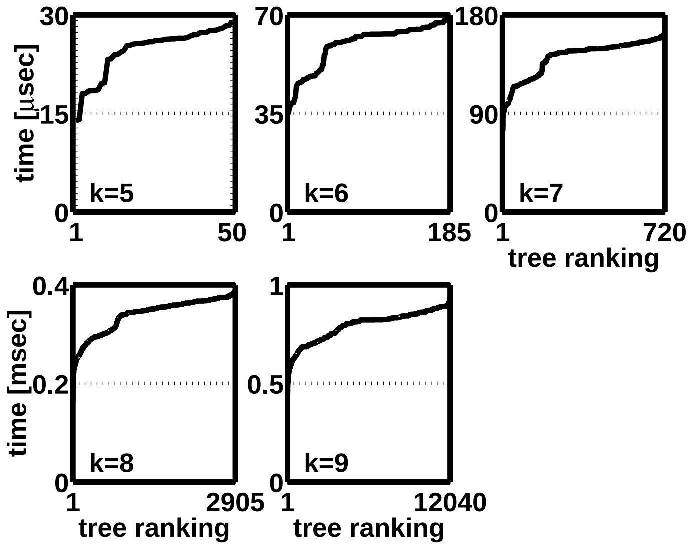


Figure 2.7: Sorted Runtimes for All Decompositions Using FFT-C

These findings are underscored in Figures 2.8 and 2.9, which contain histograms for all of the runtimes. As shown in Figure 2.9, the histograms for the runtimes of FFT-C all contain a large spike that is located near the maximum runtime. In contrast, Figure 2.8 shows that the histograms for the runtimes of FFT-Fortran contain multiple spikes, and the spikes are not all concentrated close to the maximum runtime.

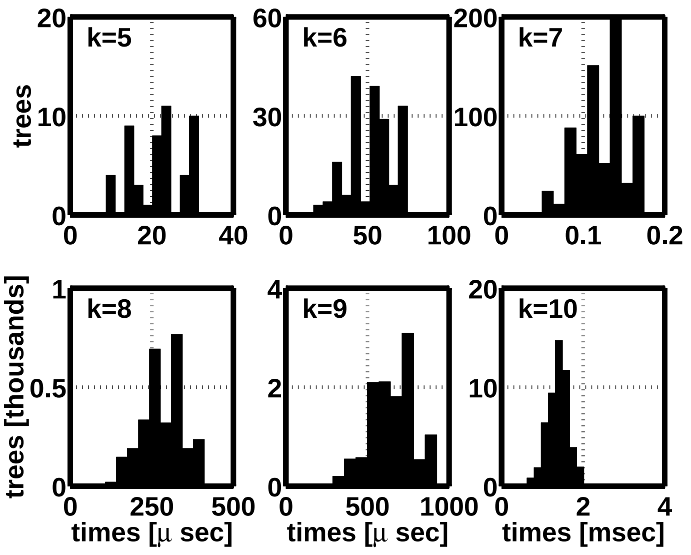


Figure 2.8: Histogram of the Runtimes for All Decompositions Using FFT-Fortran

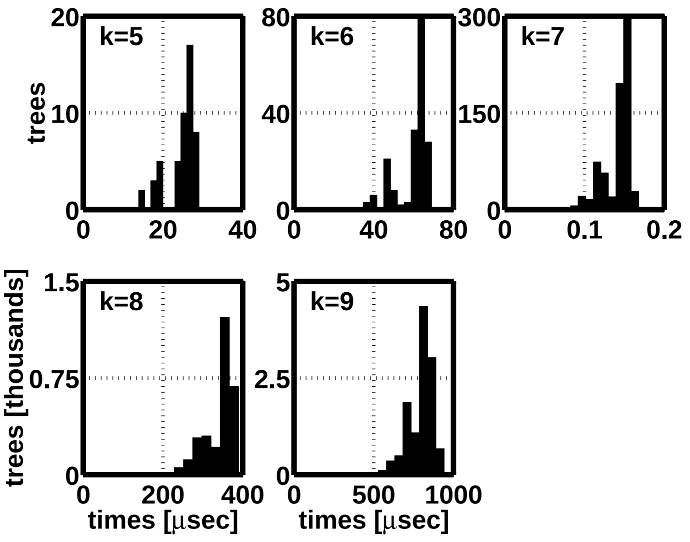


Figure 2.9: Histogram of the Runtimes for All Decompositions Using FFT-C

Figure 2.10 shows the ratios of the mean, median, and maximum runtimes to the minimum runtime for the experiments done with FFT-Fortran and FFT-C. This figure shows the average performance advantage of the optimal decomposition over a random decomposition. As shown in the right plot in the figure, all of the mean and median runtimes found using FFT-C lie close to the maximum runtime, a point that was emphasized by the histograms of Figure 2.9. This means that, on average, a random decomposition gives close to the worst possible performance for FFT-C. Another important point about the right plot in Figure 2.10 is that all of the median and mean runtimes lie close to about twice the minimum runtime. This means that FFT-C can compute an FFT of size 2^5 to 2^{10} on average *twice* as fast with the optimal decomposition as with a random decompositon.

The ratio of mean runtime to minimum runtime is even greater for FFT-Fortran, as shown in the left plot in Figurr 2.10. In this plot the mean and median runtimes are about three times the minimum runtime. This means that FFT-Fortran can compute an FFT of size 2^5 to 2^{10} on average *three times* as fast with the optimal decomposition as with a random decompositon. The findings presented in Figure 2.10 justify putting effort into finding efficient decompositions, especially since these findings are for relatively small size FFTs, in which data access patterns and cache misses are not as much of an issue.

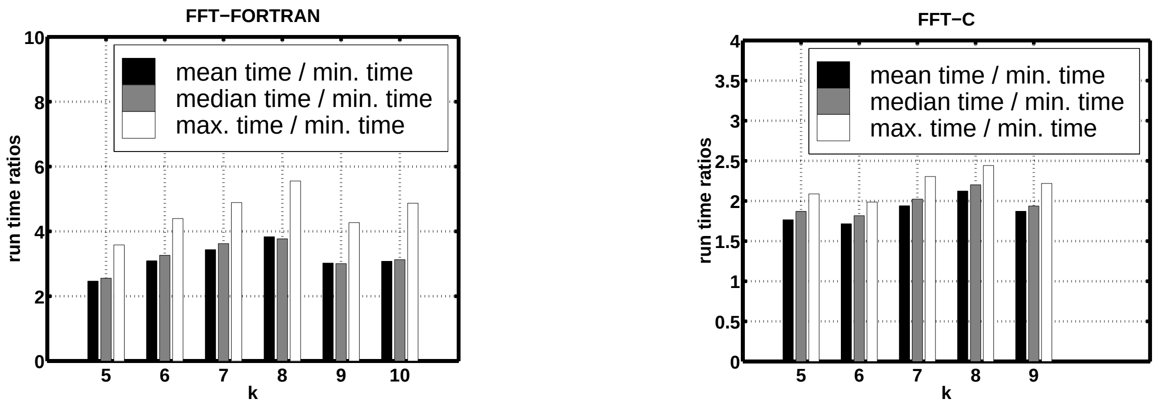


Figure 2.10: Mean, Median, and Maximum Runtime Relative to the Minimum Runtime

Figure 2.11 compares the runtimes of the optimal decompositions found using both an exhaustive search and a dynamic programming (DP) search. As shown in the figure, within the precision of the timer, which is about 5%, the performance of the two search strategies is equivalent for FFTs of size 2^5 through 2^{10}.

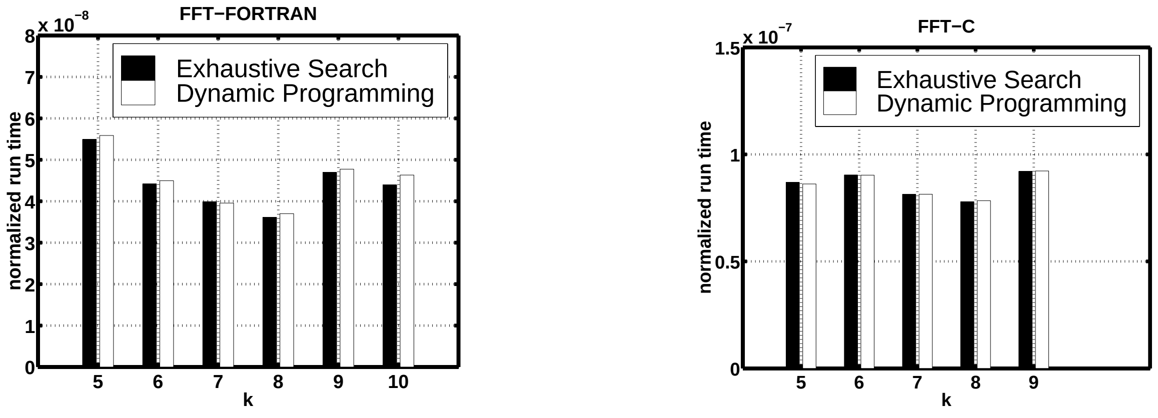


Figure 2.11: Dynamic Programming (DP) Search vs. Exhaustive Search

Figure 2.12 compares the performance of basic DP with n-best DP and stride sensitive DP. As shown in the figure, n-best DP and stride sensitive DP provide only slightly better performance than basic DP.

Figures 2.13 and 2.14 show the DP optimal decompositions found using FFT-Fortran and FFT-C, respectively. As shown in the figures, for FFTs larger than 2^{15}, the trees are mostly balanced. For smaller FFTs however, there is no clear preference for left-expanded decompositions or right-expanded decompositions. Another trend in Figures 2.13 and 2.14

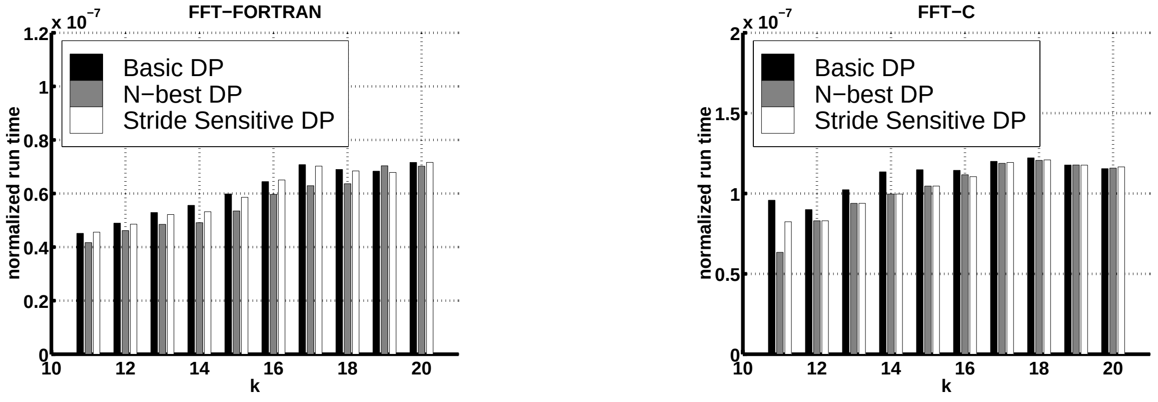


Figure 2.12: DP Search Strategy Comparisons

is that the leaf nodes consist mostly of DFTs of size 2^4, with a few DFTs of size 2^3 and almost zero DFTs of size 2^1 or 2^2.

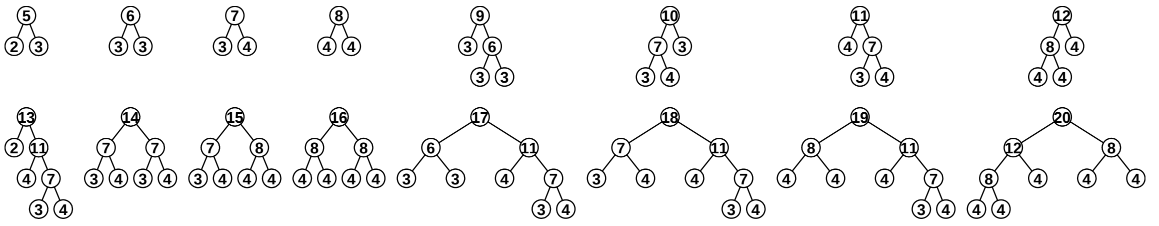


Figure 2.13: DP Optimal Decompositions Using FFT-Fortran

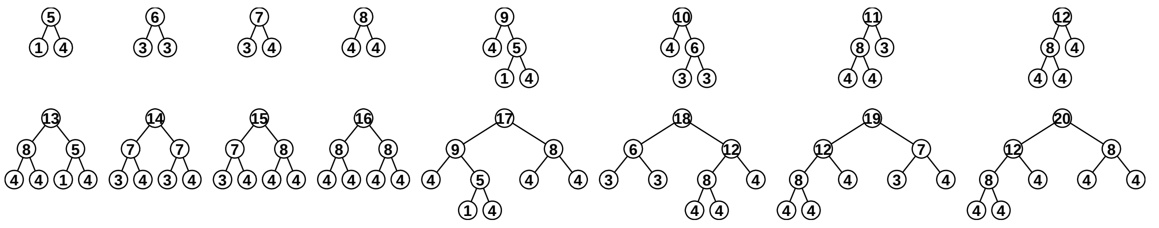


Figure 2.14: DP Optimal Decompositions Using FFT-C

Figures 2.15 and 2.16 show the five best decompositions for FFTs of size 2^6 through 2^{10}. The decompositions are found using the n-best dynamic programming search. The very best decomposition is shown on the left side of each figure, and the fifth-best decomposition is shown on the right side of each figure. As shown in Figure 2.15, the majority of the best decompositions using FFT-Fortran are right-expanded. However, as shown in Figure 2.16, the best decompositions using FFT-C are almost equally divided between right-expanded and left-expanded decompositions.

Figures 2.17 and 2.18 show the five best decompositions for FFTs of size 2^{11} through 2^{15}. As shown in the figures, more of the best decompositions are somewhat balanced for these FFT sizes than for the smaller size FFTs in Figures 2.15 and 2.16.

Figure 2.19 and 2.20 show the best five decompositions for FFTs of size 2^{16} through 2^{20}. As shown in the figures, most of the best decompositions for these size FFTs are somewhat

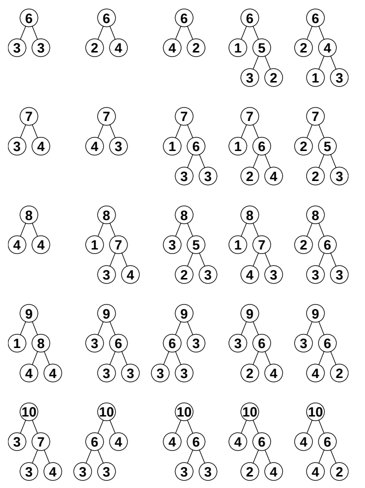


Figure 2.15: Five Best Decompositions Using FFT-Fortran, $n = 6, \ldots, 10$

balanced.

Figure 2.21 compares the runtimes of the DP optimal decompositions for FFT-Fortran and FFT-C. As shown in the figure, FFT-Fortran is often almost 50% faster than FFT-C. Since both programs use the same algorithm, the reason for the difference in performance may be that, for this type of program, the Fortran compiler, DIGITAL Visual Fortran 6.0, produces more optimized assembly code than the C++ compiler, Microsoft Visual C++ 6.0.

2.4 Conclusions

In this chapter the following findings were presented:

1. The median and mean runtimes of all decompositions are about twice the minimum runtime for FFT-C and about three times the minimum runtime for FFT-Fortran.

2. The dynamic programming search performs as well as an exhaustive search at finding efficient decompositions for FFTs of size 2^5 through 2^{10}.

3. Two more sophisticated versions of dynamic programming, n-best DP and stride sensitive DP, were introduced, but neither performs significantly better than basic DP at finding fast decompositions.

4. The DP optimal decompositions for FFTs of sizes larger than 2^{14} are mostly balanced. For FFTs of smaller sizes the optimal decompositions are balanced, left-expanded, or right-expanded about equally as often.

5. Most of the leaf nodes of the optimal decompositions consist of DFTs of size 2^4, while a smaller amount consist of DFTs of size 2^3, and almost zero consist of DFTs of size 2^1 or 2^2.

6. FFT-Fortran is about 50% faster than FFT-C. The reason for the difference may be that the Fortran compiler produces more optimized assembly code than the C++ compiler for this type of program.

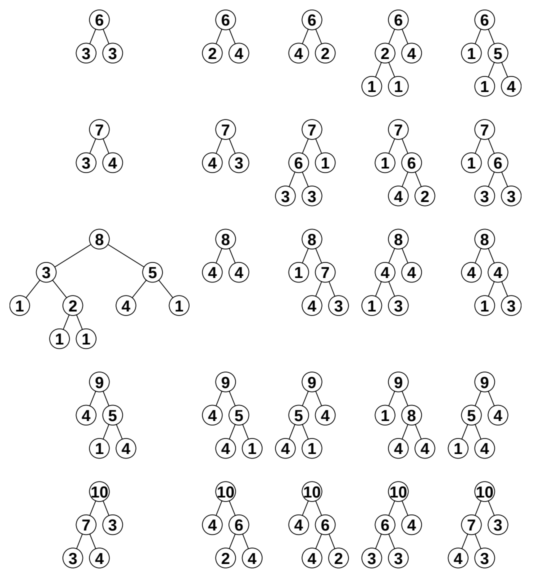


Figure 2.16: Five Best Decompositions Using FFT-C, $n = 6, \ldots, 10$

In the next chapter a successful FFT program called FFTW is used as a tool to conduct further experiments. Although the results from this chapter suggest that the optimal decompositions for larger size FFTs are balanced, FFTW was designed to compute FFTs using right-expanded decompositions only. Therefore, modifications were made to FFTW to allow it to compute FFTs using arbitrary decompositions.

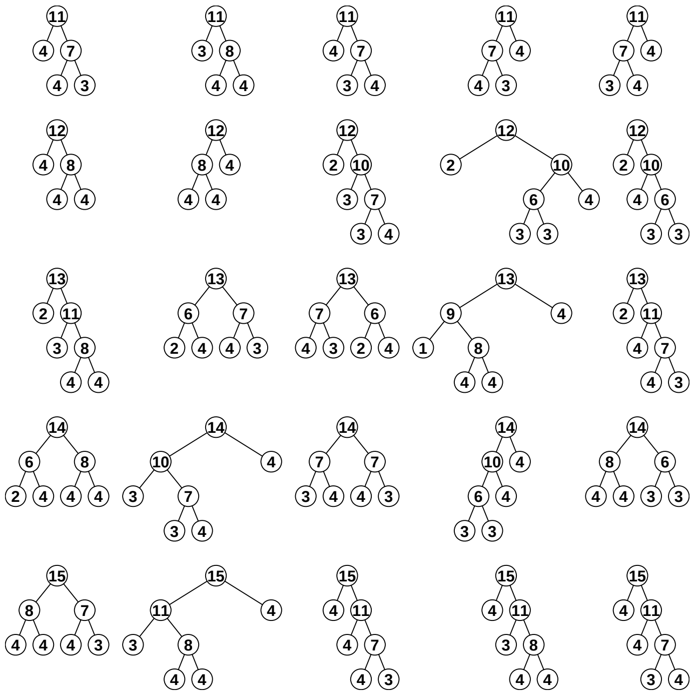


Figure 2.17: Five Best Decompositions Using FFT-Fortran, $n = 11, \ldots, 15$

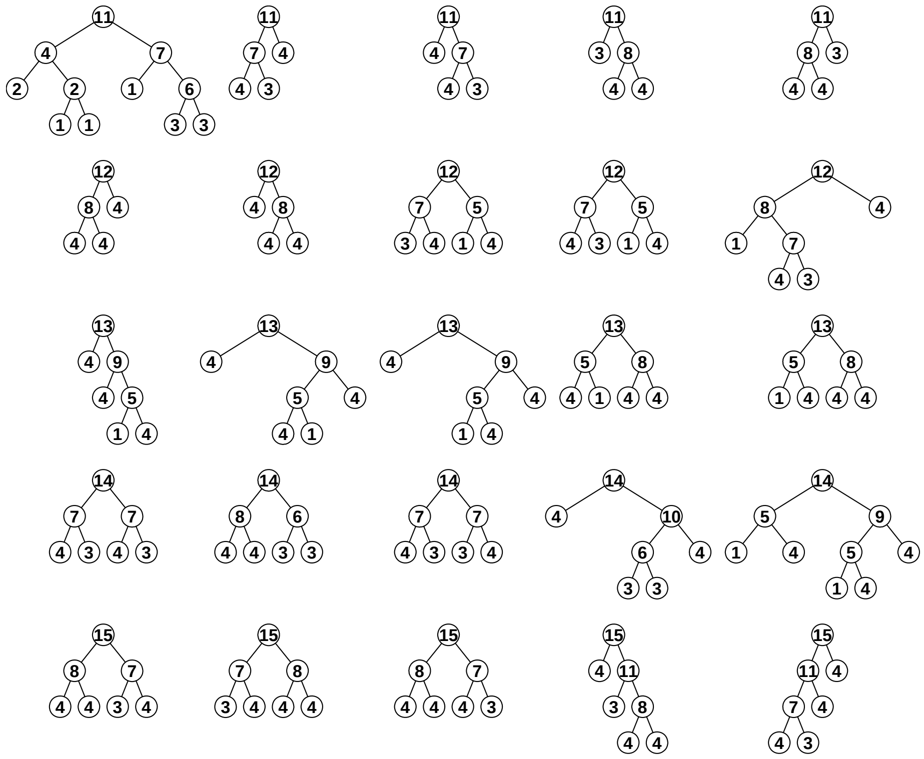


Figure 2.18: Five Best Decompositions Using FFT-C, $n = 11, \ldots, 15$

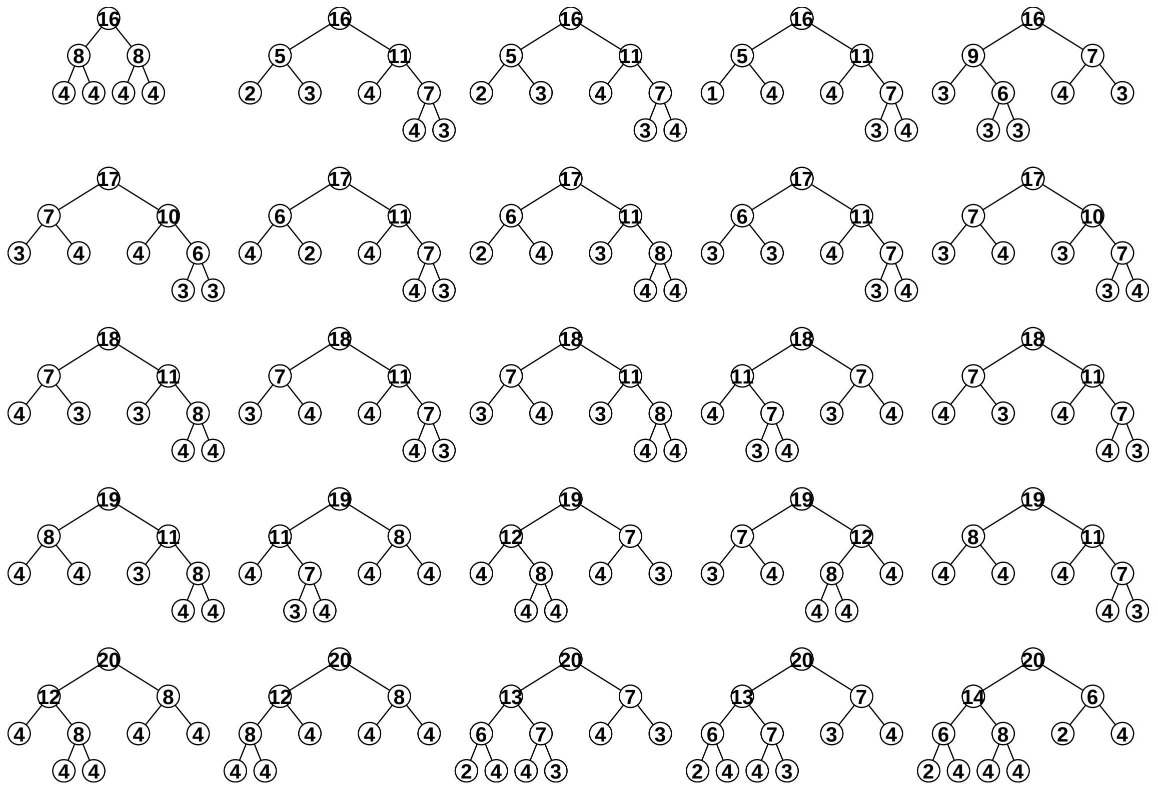


Figure 2.19: Five Best Decompositions Using FFT-Fortran, $n = 16, \ldots, 20$

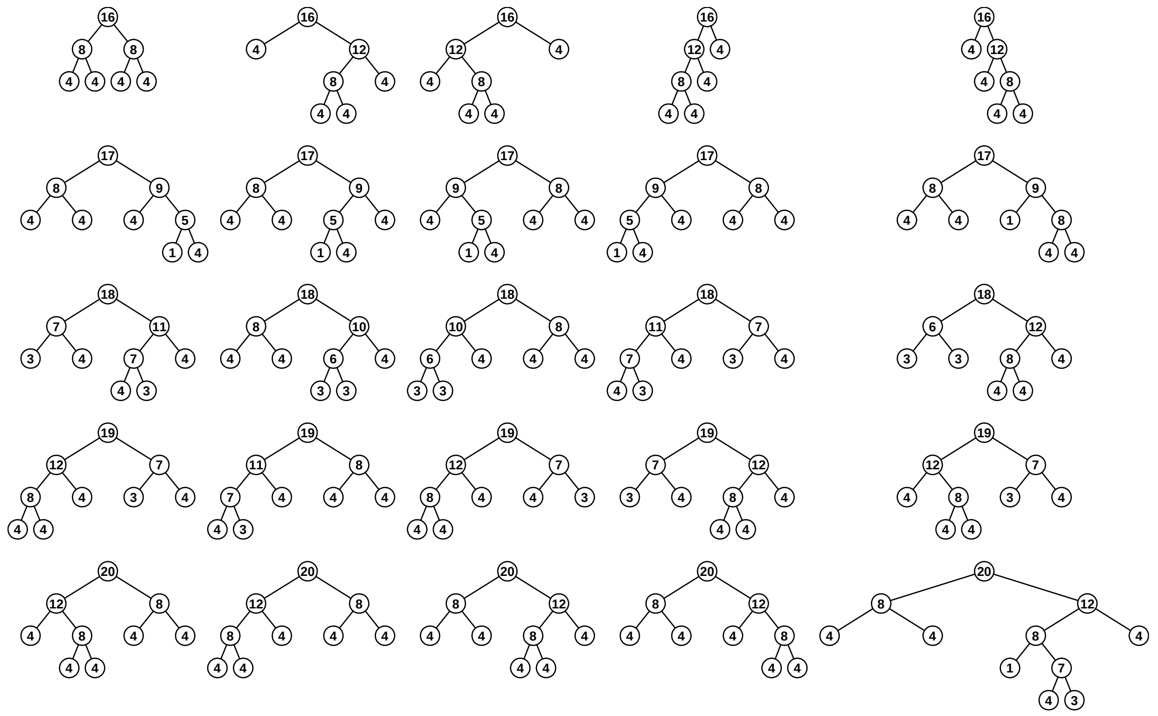


Figure 2.20: Five Best Decompositions Using FFT-C, $n = 16, \ldots, 20$

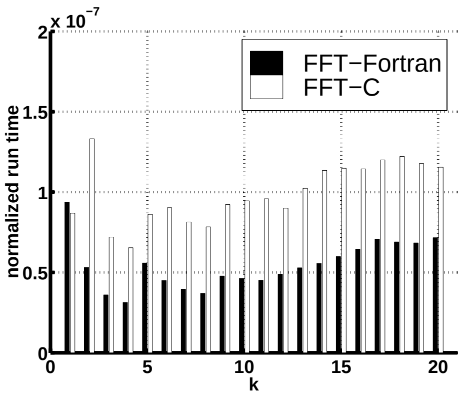


Figure 2.21: FFT-Fortran vs. FFT-C

Chapter 3

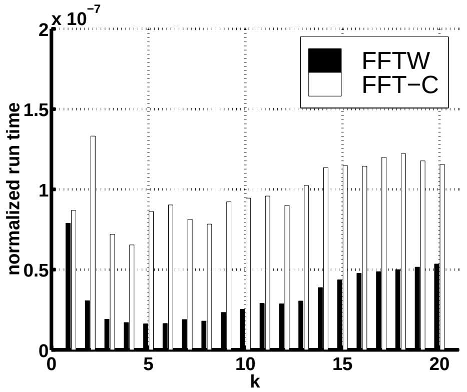
Experiments with FFTW

3.1 Introduction

3.1.1 Motivation and Goals

In Chapter 2, computer programs for computing FFTs using an algorithm similar to the one used by FFTW were introduced. Although the programs were designed to be as fast as possible, they are still slower than FFTW, as shown in Figure 3.1. There are two reasons why FFTW is faster than the programs introduced in Chapter 2. The first reason is that FFTW has more optimized code for the base cases of the recursion, as will be explained later in this chapter. The second reason is that FFTW does not use any arrays for temporary storage other than a single input and output array when it computes an FFT.

Given that FFTW is clearly faster than the programs discussed in Chapter 2, a surprising feature of FFTW is that it computes FFTs using right-expanded decompositions only. This feature is surprising in light of the experiments in Chapter 2, which suggest that the optimal decompositions tend to be balanced for FFTs larger than 2^{13}. Therefore, in order to determine if the performance of FFTW is inhibited by the fact that it computes FFTs using right-expanded decompositions only, we modified FFTW to allow it to compute FFTs corresponding to arbitrary decompositions.



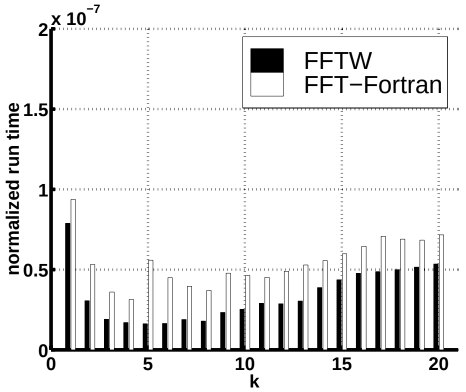


Figure 3.1: Left: FFTW vs. FFT-C, Right: FFTW vs. FFT-Fortran

The goals of the experiments discussed in this chapter are the following:

1. Determine if the performance of FFTW can be improved if it is expanded so that it can compute FFTs using arbitrary decompositions instead of only right-expanded

decompositions.

2. Determine if the stride sensitive or n-best dynamic programming searches are more effective than basic DP at finding fast decompositions for FFTW.

3. If FFTW is not faster for balanced trees than right-expanded trees, determine why not.

3.1.2 Introduction to FFTW

FFTW is an FFT program that is generally faster than all other publicly available DFT software and is at least competitive with most proprietary codes [5]. Three key features of FFTW that will be explained next are the *codelets*, the *plan structure*, and the *executor*.

Codelets

The most powerful feature of FFTW is the *codelets*, which are machine-generated blocks of C code that compute FFTs of various sizes. The codelets are generated by a Meta Language (ML) program that is explained in [4]. Each codelet computes an FFT of a specific size. For convenience, a codelet that computes an FFT of size N is referred to as a "size N codelet" in this report. The largest codelets that are distributed with FFTW are of size 64. Larger size codelets can be generated with the FFTW codelet generator FFTW, but they are inefficient because the code is very large and takes up a significant amount of space in the cache.

There are two types of codelets—*twiddle codelets* and *no-twiddle codelets*. The twiddle codelets compute in-place FFTs while the no-twiddle codelets compute out-of-place FFTs. The twiddle codelets get their name because they also perform multiplication by twiddle factors, unlike the no-twiddle codelets.

A no-twiddle codelet of size N computes the following:

$$\mathbf{y} \leftarrow \mathbf{F}_N \mathbf{x}, \tag{3.1}$$

where $\mathbf{x}$ corresponds to an input array and $\mathbf{y}$ corresponds to an output array. Both arrays are of length M, where $M \geq N$. An important feature of the no-twiddle codelets is that they can compute the FFT of N elements of $\mathbf{x}$ collected at stride P_1 and store the result in $\mathbf{y}$ at stride P_2. That is, the no-twiddle codelets can perform FFTs with an input stride that is different from the output stride.

A twiddle codelet of size R computes the following:

$$\mathbf{y} \leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}\mathbf{y}, \tag{3.2}$$

where $\mathbf{y}$ is the array for both the input and output of the computation, which is performed in-place. The length of $\mathbf{y}$ is M, where $M \geq RS$. The twiddle factors for $\mathbf{T}_S^{RS}$ are pre-computed and stored in an array that is accessed when the computation in Equation (3.2) is performed. The computation in Equation (3.2) is performed in S steps. In step i, $i = 1, \ldots, S$, the following computations are performed:

1. Read R elements of $\mathbf{y}$ at stride S, starting at location i.

2. Multiply those elements of $\mathbf{y}$ by the appropriate R twiddle factors.

3. Store the result back in $\mathbf{y}$ in the same locations that the elements of $\mathbf{y}$ were read from.

4. Compute the in-place FFT of size R on those same elements of $\mathbf{y}$.

An important aspect of the above procedure is that the multiplication by twiddle factors is interleaved with the computation of the $\mathbf{F}_R$'s. This practice is more efficient than simply multiplying all of $\mathbf{y}$ by the twiddle factors and then doing the S FFTs on $\mathbf{y}$, because it allows the entire computation in Equation (3.2) to be performed with a single pass through $\mathbf{y}$ instead of two passes. If $\mathbf{y}$ is large enough, then each time all of the elements of $\mathbf{y}$ are accessed, there could be many cache misses. Therefore, it is important to use procedures that minimize the number of passes through the data.

The Plan Structure

FFTW represents each possible right-expanded decomposition with a *plan*. A plan is simply a list of nodes that contain pointers to codelets. Figure 3.2 illustrates the two types of plan nodes in FFTW—*twiddle nodes* and *no-twiddle nodes*. As shown in the figure, each twiddle node contains pointers to an array of twiddle factors, a twiddle codelet, and another plan node called the right-child node. There is no pointer to a left-child node, for two reasons. The first reason is that only right-expanded decompositions are considered by FFTW. The second reason is that a twiddle node contains all of the information needed to perform the computations associated with a parent node and a left-child node.

Each no-twiddle node contains only a pointer to a no-twiddle codelet. No-twiddle nodes do not contain pointers to other nodes, because there is only one no-twiddle node per plan, and it is always the last node in the plan.

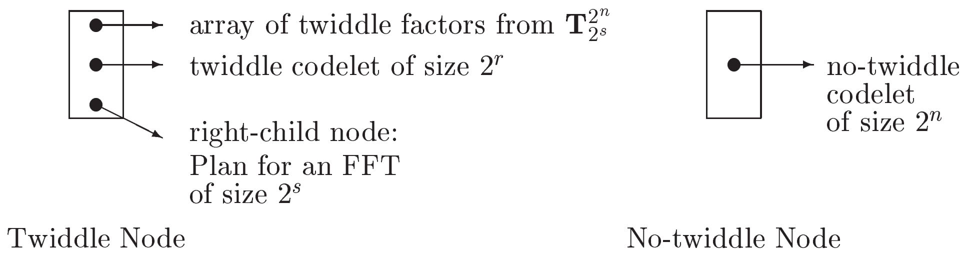
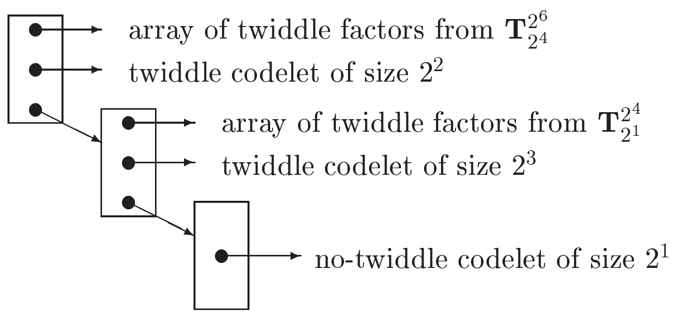

array of twiddle factors from $\mathbf{T}_{2^s}^{2^n}$
twiddle codelet of size 2^r
right-child node: Plan for an FFT of size 2^s
no-twiddle codelet of size 2^n
Twiddle Node
No-twiddle Node

Figure 3.2: The Two Types of Nodes in an FFTW Plan

An example of complete plan is shown in Figure 3.3, which corresponds to the Cooley-Tukey decomposition shown in Figure 3.4. The procedure that FFTW uses to compute an FFT according to a plan is explained next.

The Executor

The *executor* is a recursive C subroutine that computes an FFT corresponding to a plan. Like the executor program discussed in Chapter 2, the FFTW executor computes an FFT by evaluating each of the nodes in a plan and making computations based on information contained in the nodes. The executor begins the FFT computation by evaluating the root node, and recursively evaluates all of the nodes in the plan. The executor performs the computations for each node at a particular input stride and output stride. That means that when the executor evaluates a node with input stride P_{in} and output stride s_{out}, the input for the computations performed for that node is accessed at stride s_{in} and the output is stored at stride P_{out}.



array of twiddle factors from $\mathbf{T}_{2^4}^{2^6}$
twiddle codelet of size 2^2
array of twiddle factors from $\mathbf{T}_{2^1}^{2^4}$
twiddle codelet of size 2^3
no-twiddle codelet of size 2^1

Figure 3.3: The FFTW Plan for the Decomposition in Figure 3.4

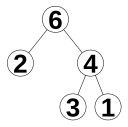


Figure 3.4: A Right-Expanded Decomposition

When the executor encounters a node of type *twiddle*, it performs the following computations:

$$\mathbf{y} \leftarrow (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} \tag{3.3}$$

$$\mathbf{y} \leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}\mathbf{y}, \tag{3.4}$$

where $\mathbf{x}$ is the input array and $\mathbf{y}$ is the output array. Like the executor discussed in Chapter 2, the FFTW executor does not explicitly permute the elements of $\mathbf{x}$ to perform the computation in Equation (3.3). Instead, the executor performs the computation in R steps, doing the following tasks on the i-th step, $i = 1, \ldots, R$:

1. Read S elements from $\mathbf{x}$ at stride $R \cdot P_{in}$, starting at location i.

2. Compute the S-point FFT of those elements from $\mathbf{x}$.

3. Store the output of the FFT in $\mathbf{y}$ at stride P_{out}, starting at location $(i-1) \cdot S \cdot P_{out} + 1$.

The executor applies the $\mathbf{F}_S$ in Equation (3.3) by recursively calling itself with the right-child node of the node currently being evaluated. The computation in Equation (3.4) is performed with a single call to the twiddle node associated with the current node. The twiddle factors for $\mathbf{T}_S^{RS}$ are accessed from the twiddle factor array that is associated with the current node.

When the executor encounters a node of type *no-twiddle*, it performs the following computation:

$$y \leftarrow \mathbf{F}_N x, \tag{3.5}$$

where $\mathbf{x}$ and $\mathbf{y}$ are again the input and output arrays. The FFT is performed at the appropriate input and output strides. The executor performs the computation in Equation (3.5) with a single call to the no-twiddle codelet that is associated with the current node.

3.2 Methodology

3.2.1 Modifications to FFTW

New Node Type

In order to allow FFTW to compute FFTs using arbitrary decompositions instead of only right-expanded decompositions, a new type of plan node was added to the plan structure. The name for the new node is "left-child-is-not-leaf"—abbreviated LCINL—since the left child of the node is never a leaf node; that is, the left child is always further decomposed. An LCINL node is illustrated in Figure 3.5. As shown in the figure, an LCINL node contains pointers to an array of twiddle factors, an array for temporary storage, a left-child node, and a right-child node. The temporary storage array is used to hold intermediate results in the FFT computation. An example of a complete plan that contains an LCINL node is shown in Figure 3.6.

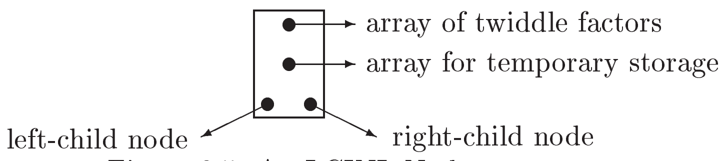


Figure 3.5: An LCINL Node

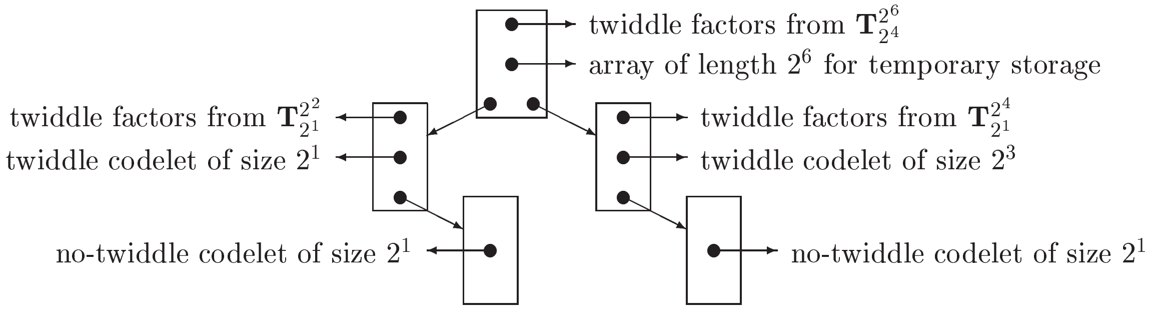

twiddle factors from $\mathbf{T}_{2^4}^{2^6}$
array of length 2^6 for temporary storage
twiddle factors from $\mathbf{T}_{2^1}^{2^2}$
twiddle codelet of size 2^1
twiddle factors from $\mathbf{T}_{2^1}^{2^4}$
twiddle codelet of size 2^3
no-twiddle codelet of size 2^1
no-twiddle codelet of size 2^1

Figure 3.6: The Plan for the Decomposition in Figure 3.7

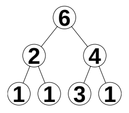


Figure 3.7: A Balanced Decomposition

Modification to the Executor

The executor was modified so that it would be able to evaluate nodes of type LCINL. This modification does not affect the way the executor evaluates nodes of type twiddle or no-twiddle.

When the modified executor encounters a node of type LCINL with input stride P_{in} and output stride P_{out}, it computes the following:

$$\mathbf{t} \leftarrow \mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} \tag{3.6}$$

$$\mathbf{y} \leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{t}, \tag{3.7}$$

where $\mathbf{x}$ and $\mathbf{y}$ are input and output arrays, and $\mathbf{t}$ is the temporary storage array associated with the node currently being evaluated. The computation in Equation (3.6) is performed in R iterations. On the i-th iteration, $i = 1, \ldots, R$, the following tasks are performed:

1. Read S elements from $\mathbf{x}$ at stride $R \cdot P_{in}$, starting at location $(i-1) \cdot P_{in} + 1$.
2. Compute the FFT of those elements of $\mathbf{x}$.
3. Multiply the result of the FFT by the appropriate S twiddle factors from the twiddle factor array associated with the current node.
4. Store the result of the twiddle factor multiplications in $\mathbf{t}$ at stride 1, starting at location $i \cdot S$.

The executor subroutine applies the $\mathbf{F}_S$ in Equation (3.6) by recursively calling itself to evaluate the right-child node of the current node. The right-child node is evaluated at input stride $R \cdot P_{in}$ and output stride 1.

The computation in Equation (3.7) is performed in n iterations. On the i-th iteration, $i = 1, \ldots, S$, the following tasks are performed:

1. Read R elements from $\mathbf{t}$ at stride S, starting at location i.
2. Compute the FFT of those R elements from $\mathbf{t}$.
3. Store the output of the FFT in $\mathbf{y}$ at stride $S \cdot P_{out}$, starting at location $(i-1) \cdot P_{out} + 1$.

The executor routine applies the $\mathbf{F}_R$ in Equation (3.7) by recursively calling itself to evaluate the left-child node of the current node. The left-child node is evaluated at input stride S and output stride $S \cdot P_{out}$.

3.2.2 Modification to the Stride Sensitive DP Search Strategy

The stride sensitive approach used with FFTW is slightly different than the stride sensitive search discussed in Chapter 2. As before, the idea of the stride sensitive DP search is to determine the optimal decomposition for a particular size FFT at a given input stride and output stride. The complication that arises in the case of the modified version of FFTW is that the input and output strides of the left and right children of a node depend on whether the node is of type twiddle or LCINL. This is due to the fact that the executor makes different computations for a twiddle node than it does for an LCINL node.

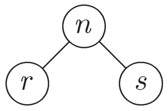


Figure 3.8: A Single Decomposition

Consider the simple decomposition shown in Figure 3.8. This decomposition could appear anywhere in a larger decomposition. Assume that the input stride for node n is P_{in} and the input stride is P_{out}. Table 3.1 lists the input and output strides for the child nodes of node n when the node is of type twiddle. Similarly, Table 3.2 lists the input and output

	right-child node	left-child node
input stride:	$2^r \cdot P_{in}$	$2^s \cdot P_{out}$
output stride:	P_{out}	$2^s \cdot P_{out}$

Table 3.1: Input and Output Strides for the Child Nodes of a Twiddle Node

	right-child node	left-child node
input stride:	$2^r \cdot P_{in}$	2^s
output stride:	1	$2^s \cdot P_{out}$

Table 3.2: Input and Output Strides for the Child Nodes of a LCINL Node

strides for the child nodes of node n when the node is of type LCINL. Comparing the two tables, one sees that twiddle nodes and LCINL nodes differ in the output stride for the right-child node and the input stride for the left-child node. The reason for the difference in output strides for right-child nodes is that the output of the right-child node computation is stored in the output array $\mathbf{y}$ for twiddle nodes, and in the temporary storage array $\mathbf{t}$ for LCINL nodes. Similarly, the reason for the difference in input strides for left-child nodes is that the input for the left-child node computation is the output array $\mathbf{y}$ for twiddle nodes, and the temporary storage array $\mathbf{t}$ for LCINL nodes.

3.2.3 Adding a Penalty for Twiddle Nodes

In order to compensate for the fact that the executor normally has to access three arrays when evaluating an LCINL node and only two when evaluating a twiddle node, a modification was made that forces the executor to access a third array when evaluating twiddle nodes. The modification was made in two steps. The first step was to add pointers to temporary storage arrays in the twiddle nodes. The second step was to modify the executor so that it reads and writes data to a temporary storage array when it evaluates a twiddle node. Specifically, when the modified executor encounters a node of type twiddle, it performs the following computations:

$$\begin{aligned} \mathbf{t} &\leftarrow (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} && (3.8) \\ \mathbf{y} &\leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}\mathbf{t}, && (3.9) \end{aligned}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the input and output arrays, and $\mathbf{t}$ is the temporary storage array associated with the current node. Essentially, this modification forces the executor to store the output of the computation in Equation (3.8) in $\mathbf{t}$ instead of directly in $\mathbf{y}$.

3.2.4 Removing a Penalty for LCINL Nodes

In another attempt to compensate for the fact that the executor normally accesses three arrays when evaluating an LCINL node and only two when evaluating a twiddle node, a modification was made that allows the executor to access only two arrays when evaluating LCINL nodes. This modification was made independently of the previous modification, in which a penalty was added to the executor for twiddle nodes. This modification causes FFTW to no longer compute FFTs correctly, so it was added only to see how the runtimes would be affected. This modification was made in two steps. The first step was to remove the pointers to temporary storage arrays from the LCINL nodes. The second step was to modify the executor so that when it encounters a node of type LCINL, it computes

$$\begin{aligned} \mathbf{x} &\leftarrow \mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} && (3.10) \\ \mathbf{y} &\leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{x}, && (3.11) \end{aligned}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the input and output arrays. Essentially, this modification allows the executor to store the result of the computation in Equation (3.10) back in $\mathbf{x}$ instead of in a third array.

3.2.5 Removing All Twiddle Factor Multiplications

In order to compensate for the fact that the executor performs twiddle factor multiplications outside of a codelet when evaluating a LCINL node, and inside of a codelet when evaluating a twiddle node, the following modification was made: The expanded version of FFTW was modified so that none of the twiddle factor multiplications are performed when an FFT is computed. The modification was added independently of the previous two modifications—the addition of a penalty for twiddle nodes and the removal of a penalty for LCINL nodes. Like the previous modification, this modification causes FFTW to compute FFTs incorrectly, so it was added only to see how the runtimes would be affected. This modification was implemented by replacing the twiddle codelets with modified versions of the no-twiddle codelets. The modified no-twiddle codelets compute:

$$\mathbf{y} \leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{y} \qquad (3.12)$$

in contrast to the regular twiddle codelets, which compute

$$\mathbf{y} \leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}\mathbf{y}. \qquad (3.13)$$

The modification allows these no-twiddle codelets to perform the same computations as twiddle codelets, with the exception that the no-twiddle codelets do not perform the twiddle factor multiplications.

In addition to replacing the twiddle codelets, the executor was also modified so that when it encounters a node of type LCINL, it computes:

$$\begin{aligned} \mathbf{t} &\leftarrow (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} && (3.14) \\ \mathbf{y} &\leftarrow \mathbf{L}_R^{RS}(\mathbf{I}_S \otimes \mathbf{F}_R)\mathbf{L}_S^{RS}\mathbf{t}, && (3.15) \end{aligned}$$

instead of:

$$\begin{aligned} \mathbf{t} &\leftarrow \mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}\mathbf{x} && (3.16) \\ \mathbf{y} &\leftarrow \mathbf{L}_R^{RS}(\mathbf{I}_S \otimes \mathbf{F}_R)\mathbf{L}_S^{RS}\mathbf{t}. && (3.17) \end{aligned}$$

As can be seen by comparing Equations (3.16) and (3.14), the twiddle factor multiplications that are normally performed when LCINL nodes are evaluated are no longer performed.

3.3 Results

Figure 3.9 shows the dynamic programming (DP) optimal decompositions found using the modified version of FFTW that can compute FFTs using arbitrary decompositions. As

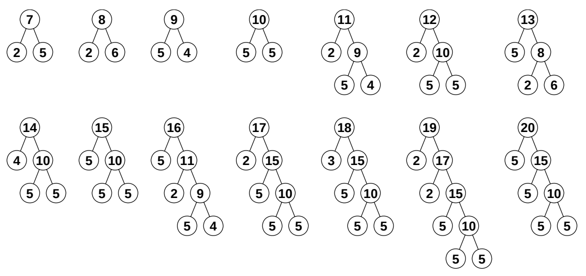


Figure 3.9: DP Optimal Decompositions for the Expanded Version of FFTW

shown in the figure, the optimal decompositions that were found are all right-expanded. This result means that none of the optimal plans contain an LCINL node.

Figures 3.10 through 3.12 show the five best decompositions for FFTs of size 2^6 through 2^{20}. The very best decompositions are shown on the left of each figure, and the fifth-best decompositions are shown on the right of each figure. As shown in the figures, the best five trees are all right-expanded for FFT sizes of 2^6 through 2^{20}.

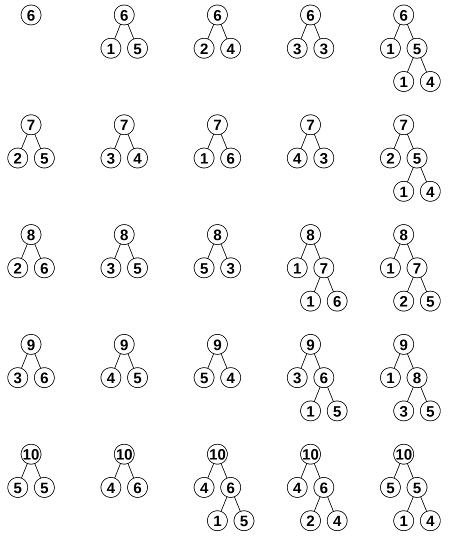


Figure 3.10: Best Five Decompositions for FFTs of Size 2^6 through 2^{10}

Figure 3.13 compares the runtimes of the optimal decompositions found using three types of dynamic programming—basic, n-best, and stride sensitive. As shown in the figure, within the precision of the timer, which is about 5%, the three search strategies perform equally well.

Figure 3.14 shows the DP optimal decompositions found when a penalty is added to the executor for evaluating twiddle nodes. As shown in the figure, with the penalty added, the optimal decompositions for FFTs of size of 2^{19} and 2^{20} are not right-expanded, although the optimal decompositions for all smaller size FFTs are right-expanded. Figure 3.15 shows the surprising result that adding the penalty for twiddle nodes to the executor does not significantly increase the runtimes of the optimal decompositions. This result suggests that accessing the extra temporary storage arrays may not be very costly.

Figure 3.16 shows the DP optimal decompositions found when the penalty paid by the executor for evaluating LCINL nodes is removed. As shown in the figure, right-expanded

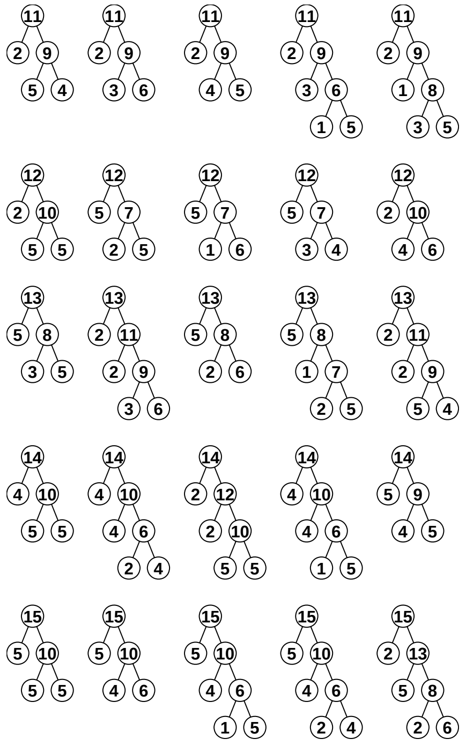


Figure 3.11: Best Five Decompositions for FFTs of Size 2^{11} through 2^{15}

decompositions are still optimal without the penalty. This result suggests that the reason decompositions with LCINL nodes lose to right-expanded decompositions is not because the executor must access a temporary storage array when evaluating a LCINL node.

Figure 3.17 shows the DP optimal decompositions when all twiddle factor multiplications are removed from the FFT computation. Under these conditions, the optimal decompositions are still right-expanded. This result suggests that the fact that twiddle factor multiplications must be performed outside of a codelet when the executor evaluates an LCINL node is not the reason why decompositions with LCINL nodes lose to right-expanded decompositions.

Figure 3.18 shows the DP optimal decompositions when the data access penalty is added for twiddle nodes and all twiddle factor multiplications are removed. As shown in the figure, under these conditions many of the decompositions are somewhat balanced, and only one is right-expanded. This result suggests that balanced decompositions are inferior to right-expanded decompositions due to a *combination* of two factors: twiddle factors multiplications that are performed outside of codelets for LCINL nodes, and the cost associated with accessing extra arrays for LCINL nodes.

3.4 Conclusions

The following conclusions can be drawn from the results presented in this chapter:

1. Even if FFTW is modified so that it can compute FFTs according to arbitrary decompositions instead of only right-expanded decompositions, the right-expanded decompositions are still optimal.

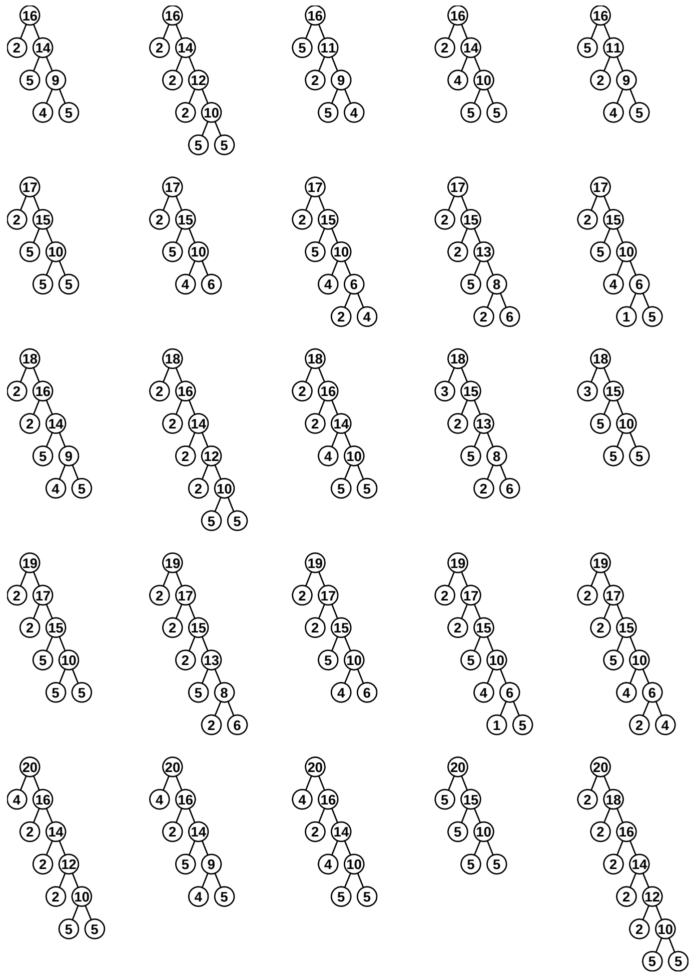


Figure 3.12: Best Five Decompositions for FFTs of Size 2^{16} through 2^{20}

2. N-best DP and stride sensitive DP do not offer an advantage over basic DP when searching for a fast decomposition for FFTW.
3. Neither the cost of accessing extra arrays for LCINL nodes nor the cost of performing twiddle factor multiplications outside of codelets by itself is the reason that the right-expanded trees are optimal. Instead, it is a combination of these two factors that causes non-right-expanded decompositions to be inferior to right-expanded decompositions.

In the previous chapter it was shown that the optimal Cooley-Tukey decompositions for FFTs larger than 2^{13} tend to be somewhat balanced. In this chapter it was shown that the optimal decompositions are right-expanded, but the exact reason why is not clear. It was concluded that the reason was due at least in part to the fact that non-right-expanded decompositions require extra arrays to be accessed and twiddle factor multiplications to be performed outside of codelets. However, it is not clear whether or not there is some other reason why right-expanded decompositions are the fastest. Therefore, in the next chapter, a powerful FFT program that does not access extra arrays for non-right-expanded decompositions or use codelets to perform twiddle factor multiplications is investigated.

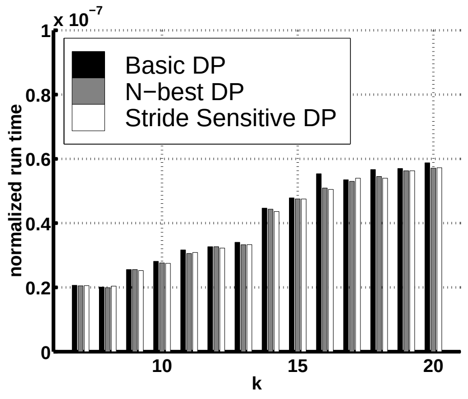


Figure 3.13: Comparison of Dynamic Programming Search Strategies

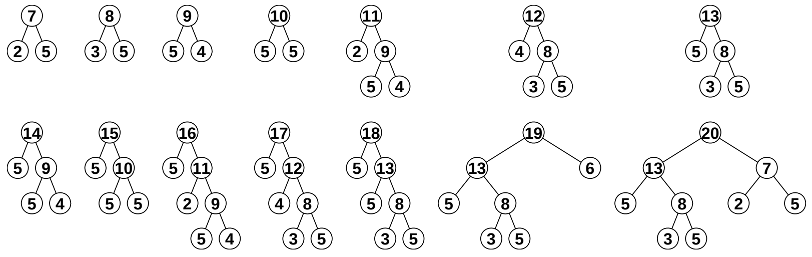


Figure 3.14: DP Optimal Decompositions with Penalty Added for Twiddle Nodes

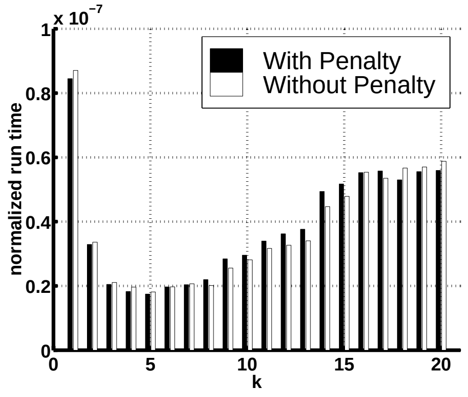


Figure 3.15: Best Runtimes With and Without the Twiddle Node Penalty

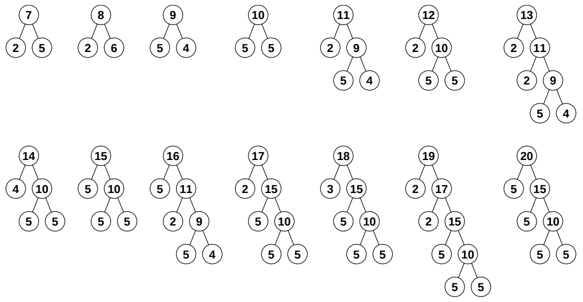


Figure 3.16: DP Optimal Decompositions with Penalty for LCINL Nodes Removed

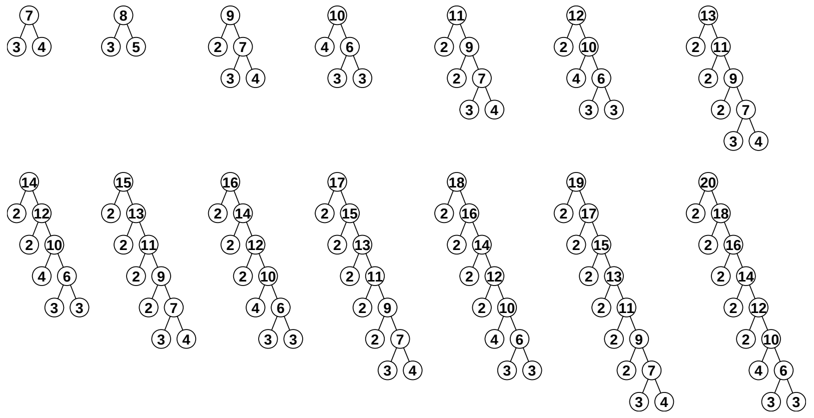


Figure 3.17: DP Optimal Decompositions with All Twiddle Factor Multiplications Removed

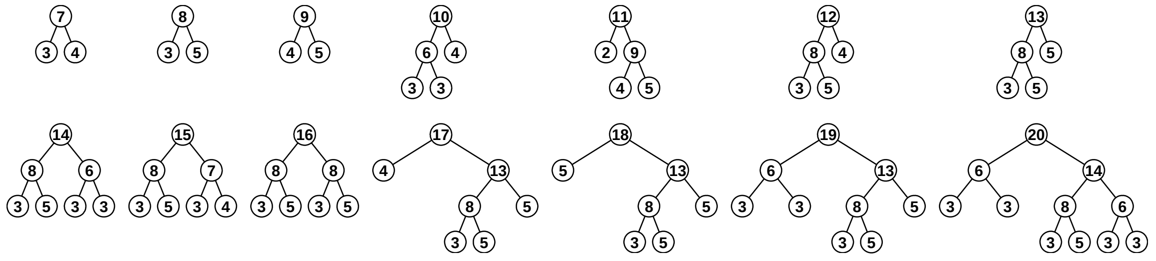


Figure 3.18: Penalty for Twiddle Nodes and All Twiddle Factor Multiplications Removed

Chapter 4

Experiments with Egner's FFT Program

4.1 Introduction

4.1.1 Motivation and Goals

The results of experiments conducted with a modified version of FFTW in Chapter 3 suggest that right-expanded decompositions are optimal, in contradiction to results presented in Chapter 2 that suggest that balanced decompositions are optimal. However, since it was determined that the FFTW tool suffers a penalty for non-right-expanded decompositions, it was decided that another program would be needed to investigate the decompositions without bias. A program developed by Sebastian Egner and investigated in this chapter is highly appropriate for the task, since it suffers no penalty for non-right-expanded decompositions. However, as shown in Figure 4.1, Egner's program is significantly slower than FFTW, which benefits from highly-optimized, machine-generated code for the base cases of the recursion and the fact that it computes FFTs without performing explicit permutations. Therefore, it was decided that Egner's program would first have to be significantly optimized so that it would be competitive with FFTW before it could be used as a reliable tool for finding fast decompositions.

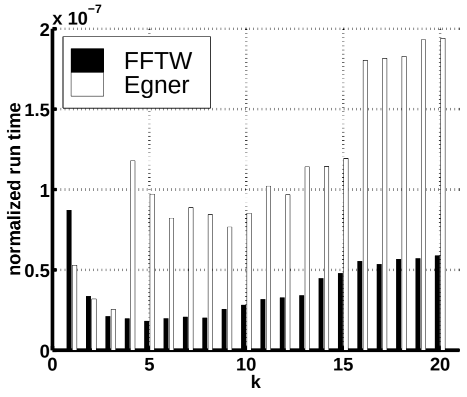


Figure 4.1: FFTW vs. Egner's FFT Program

The goals of the research discussed in this chapter are the following:

1. Implement various modifications to Egner's program to make it more competitive

with FFTW.

2. Investigate the topology of the optimal decompositions using improved versions of Egner's program.
3. Determine which aspects of Egner's program and FFTW are most important for creating an efficient FFT implementation.

4.1.2 Introduction to Egner's Program

Egner's program has many features that make it exceptionally powerful. The main features are:

1. Generic arithmetics over an arbitrary base field.
2. Various algorithms for computing the FFT other than Cooley-Tukey, including Rader, Good-Thomas, Bluestein, and Q-power.
3. Efficient methods for non-power-of-two FFT sizes
4. Efficient methods for large prime FFT sizes.
5. A syntactical interface that makes experiments easy. The interface allows decompositions to be expressed in words so that they are highly readable.
6. Reentrant design that allows several applications to use the package at a single time.

Five important elements of Egner's program that are discussed next are the small-FFT code modules, the plan structure, the iterative nature, the permutation arrays, and the executor subroutine.

Small-FFT Code Modules

Like FFTW, Egner's program used highly optimized code modules to compute the small FFTs at the base cases of the recursion in the Cooley-Tukey algorithm. Egner's program includes code modules for computing FFTs of size 2^1 through 2^3. The algorithms for the code modules were adapted from the Nussbaumer text [7]. Each small-FFT code module computes

$$\mathbf{x} \leftarrow (\mathbf{I}_C \otimes \mathbf{F}_N \otimes \mathbf{I}_D)\mathbf{x}, \tag{4.1}$$

where $\mathbf{x}$ is the array containing the data to be transformed, and C and D are integer arguments to the code module. The purpose of the arguments C and D is explained below.

The Plan Structure

Like FFTW, Egner's program uses a plan structure to represent Cooley-Tukey decompositions. There are two types of nodes in Egner's plan structure—*Cooley-Tukey* nodes and *small-FFT* nodes. The two types of plan nodes are illustrated in Figure 4.2. As shown in the figure, Cooley-Tukey nodes contain pointers to an array of twiddle factors, a permutation array, a left-child node, and a right-child node. The concept of a permutation array is explained below. The small-FFT nodes, also shown in Figure 4.2, contain only a pointer to a code module that computes a small FFT.

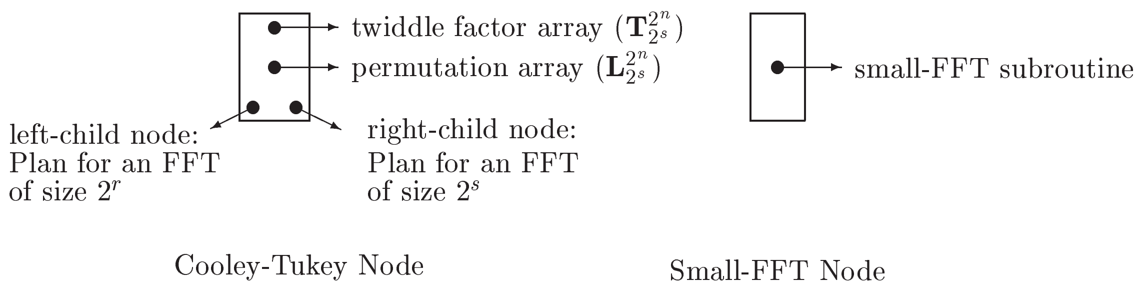

twiddle factor array ($\mathbf{T}_{2^s}^{2^n}$)
permutation array ($\mathbf{L}_{2^s}^{2^n}$)
left-child node: Plan for an FFT of size 2^r
right-child node: Plan for an FFT of size 2^s
small-FFT subroutine
Cooley-Tukey Node
Small-FFT Node

Figure 4.2: The Two Types of Plan Nodes in Egner's Program

Unlike the FFT programs discussed in Chapters 2 and 3, Egner's program computes FFTs using the transpose of the tensor product formulation of the Cooley-Tukey (CT) algorithm shown below:

$$\mathbf{F}_{RS} = (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{L}_R^{RS}. \tag{4.2}$$

The transpose of the algorithm in Equation (4.2) must compute the same result as the untransposed algorithm since the DFT matrix $\mathbf{F}_{mn}$ is symmetric. Egner's algorithm is derived as follows: Using the rule

$$(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}, \tag{4.3}$$

the transpose of $\mathbf{F}_{mn}$ can be written as

$$\mathbf{F}_{RS}^{\mathrm{T}} = \left(\mathbf{L}_R^{RS}\right)^{\mathrm{T}}(\mathbf{I}_R \otimes \mathbf{F}_S)^{\mathrm{T}}\left(\mathbf{T}_S^{RS}\right)^{\mathrm{T}}(\mathbf{F}_R \otimes \mathbf{I}_S)^{\mathrm{T}}. \tag{4.4}$$

Next, using the rule

$$(\mathbf{A} \otimes \mathbf{B})^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}} \otimes \mathbf{B}^{\mathrm{T}}, \tag{4.5}$$

Equation (4.4) can be written as

$$\mathbf{F}_{RS} = \left(\mathbf{L}_R^{RS}\right)^{\mathrm{T}}(\mathbf{I}_R^{\mathrm{T}} \otimes \mathbf{F}_S^{\mathrm{T}})\left(\mathbf{T}_S^{RS}\right)^{\mathrm{T}}(\mathbf{F}_R^{\mathrm{T}} \otimes \mathbf{I}_S^{\mathrm{T}}). \tag{4.6}$$

Finally, using the fact that matrices $\mathbf{I}_R$, $\mathbf{F}_S$, and $\mathbf{T}_S^{RS}$ are all symmetric, and the fact that

$$\left(\mathbf{L}_R^{RS}\right)^{\mathrm{T}} = \mathbf{L}_S^{RS}, \tag{4.7}$$

Equation (4.6) can be written as

$$\mathbf{F}_{RS} = \mathbf{L}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{T}_S^{RS}(\mathbf{F}_R \otimes \mathbf{I}_S). \tag{4.8}$$

Using the two plan nodes shown in Figure 4.2, one can create any decomposition of the form shown in Equation (4.8). An example of a complete plan for Egner's program is shown in Figure 4.3. The plan corresponds to the decomposition shown in Figure 4.4.

Iterative Nature of the Program

An important difference between Egner's program and FFTW is that Egner's program is not quite as recursive as FFTW. In fact, Egner's program has an iterative nature that may hinder performance for data locality reasons that are explained below. The iterative

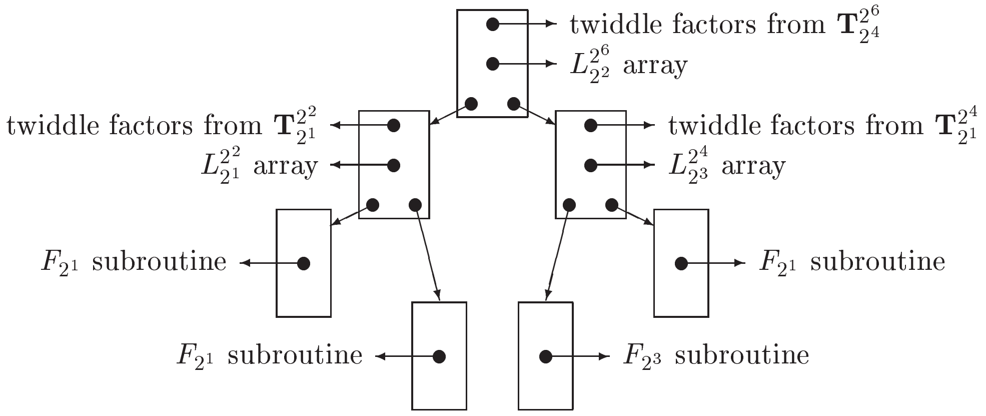


Figure 4.3: Plan Corresponding to the Decomposition in Figure 4.4

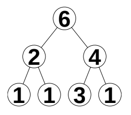


Figure 4.4: A Cooley-Tukey Decomposition for an FFT of Size 2^{15}

nature of Egner's program can be explained with the following example. Consider the decomposition shown in Figure 4.5. An FFT computation according to this decomposition can be written in tensor product notation as:

$$\mathbf{y} = (\mathbf{F}_{2^2} \otimes \mathbf{I}_{2^4})\mathbf{T}^{2^6}_{2^4}\left(\mathbf{I}_{2^2} \otimes \left[(\mathbf{F}_{2^1} \otimes \mathbf{I}_{2^3})\mathbf{T}^{2^4}_{2^3}(\mathbf{I}_{2^1} \otimes \mathbf{F}_{2^3})\mathbf{L}^{2^4}_{2^1}\right]\right)\mathbf{L}^{2^6}_{2^2}\mathbf{x}. \tag{4.9}$$

FFTW would compute the FFT by applying the operations in about the order that they are shown in Equation (4.9), with the exception that FFTW would not perform explicit permutations. In contrast, Egner's program uses a procedure that differs significantly from the order of operations shown in Equation (4.9). First of all, as explained above, Egner's program uses the transpose of the regular algorithm. In this case, the transpose of the algorithm is given by:

$$\mathbf{y} = \mathbf{L}^{2^6}_{2^4}\left(\mathbf{I}_{2^2} \otimes \left[\mathbf{L}^{2^4}_{2^3}(\mathbf{I}_{2^1} \otimes \mathbf{F}_{2^3})\mathbf{T}^{2^4}_{2^3}(\mathbf{F}_{2^1} \otimes \mathbf{I}_{2^3})\right]\right)\mathbf{T}^{2^6}_{2^4}(\mathbf{F}_{2^2} \otimes \mathbf{I}_{2^4})\mathbf{x}. \tag{4.10}$$

Secondly, Egner's program does not compute the FFT in the order specified by the parentheses in Equation (4.10). Using properties of the tensor product introduced in Chapter 1, Equation (4.10) can be expanded as:

$$\mathbf{y} = \mathbf{L}^{2^6}_{2^4}(\mathbf{I}_{2^2} \otimes \mathbf{L}^{2^4}_{2^3})(\mathbf{I}_{2^3} \otimes \mathbf{F}_{2^3})(\mathbf{I}_{2^2} \otimes \mathbf{T}^{2^4}_{2^3})(\mathbf{I}_{2^2} \otimes \mathbf{F}_{2^1} \otimes \mathbf{I}_{2^3})\mathbf{T}^{2^6}_{2^4}(\mathbf{F}_{2^2} \otimes \mathbf{I}_{2^4})\mathbf{x}. \tag{4.11}$$

Egner's program performs the operations from right to left exactly in the order that they are shown in Equation (4.11). Comparing Equations (4.9) and (4.11), one sees that while FFTW alternates between performing $\mathbf{F}_{2^1}$'s and $\mathbf{F}_{2^3}$'s, Egner's program first computes all of the $\mathbf{F}_{2^1}$'s and then computes all of the $\mathbf{F}_{2^3}$'s afterwards. The advantage of the approach used by FFTW is that the $\mathbf{F}_{2^1}$ and the $\mathbf{F}_{2^3}$ are applied to the same section of data at roughly the same time. In contrast, Egner's program first applies $\mathbf{F}_{2^1}$'s to all 64 elements of the data, and then applies $\mathbf{F}_{2^3}$'s to all 64 elements of the data. Essentially, FFTW uses the data more efficiently with respect to the cache than Egner's program. On the other hand,

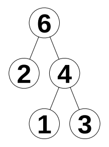


Figure 4.5: A Cooley-Tukey Decomposition for an FFT of Size 2^6

since Egner's program uses one code module for computing a small DFT at a time, instead of alternating between code modules as FFTW does, Egner's program uses the instructions more efficiently than FFTW with respect to the cache.

Execution of a Plan

Like FFTW, Egner's program also uses an executor program to compute FFTs corresponding to the decomposition in a plan. Egner's executor program begins evaluating a plan at the root node, and computes an FFT by evaluating all of the nodes in the plan using a depth-first search traversal.

When Egner's executor program encounters a node of type Cooley-Tukey, it computes the following:

$$\mathbf{x} \leftarrow (\mathbf{I}_C \otimes \mathbf{F}_R \otimes \mathbf{I}_{DS})\mathbf{x}, \tag{4.12}$$
$$\mathbf{x} \leftarrow (\mathbf{I}_C \otimes \mathbf{T}_S^{RS} \otimes \mathbf{I}_D)\mathbf{x}, \tag{4.13}$$
$$\mathbf{x} \leftarrow (\mathbf{I}_{CR} \otimes \mathbf{F}_S \otimes \mathbf{I}_D)\mathbf{x}, \tag{4.14}$$
$$\mathbf{x} \leftarrow (\mathbf{I}_C \otimes \mathbf{L}_S^{RS} \otimes \mathbf{I}_D)\mathbf{x}, \tag{4.15}$$

where $\mathbf{x}$ is the array that contains the data to be transformed. The parameters C and D are both set to 1 for the root node, and accumulate with deeper levels of recursion. Specifically, C increases with each deeper left child node that is evaluated, while D increases with each deeper right child node that is evaluated. The executor performs the computation in Equation (4.12) by recursively calling itself with the left-child node of the current node. Similarly, the executor performs the computation in Equation (4.14) by recursively calling itself with the right-child node of the current node. The permutation $\mathbf{L}_S^{RS}$ is performed according to the permutation array associated with the current node. The procedure for storing a permutation in an array and then performing the permutation according to the array are explained in the next section.

When Egner's program encounters a small-FFT node, it performs the computation shown in Equation (4.1). The computation is performed with a single call to the small code module associated with the plan node.

Permutations

An important feature of Egner's program is that it performs explicit permutations on the data, unlike FFTW. This practice allows Egner's program to perform FFTs completely in-place, and is the reason that Egner's program does not penalize for decompositions that are not completely right-expanded as FFTW does. Each permutation is stored in *cycle notation* as an array of signed integers. The *cycle* $(x_1, x_2, \ldots, x_N)$ corresponds to a permutation of a vector of length $M > N$ in which element x_i is moved to position x_{i+1}, $i = 1, \ldots, N-1$, and element x_N is moved to position x_1. Figure 4.6 shows both of the cycles needed to represent the permutation $\mathbf{L}_2^8\mathbf{x}$. Figure 4.7 illustrates the steps that Egner's program would take to

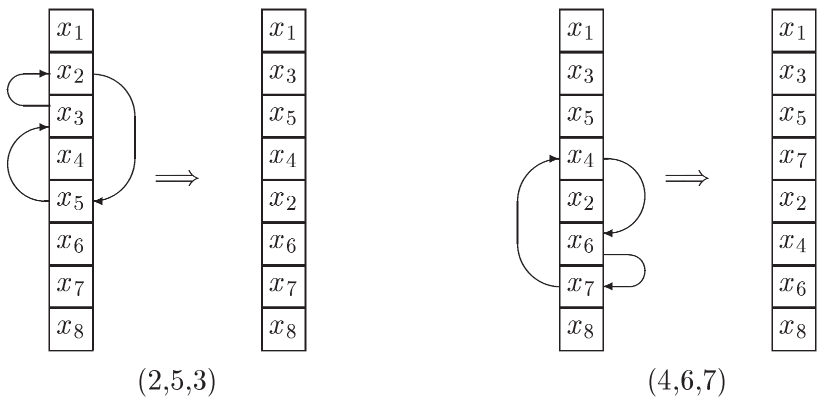


Figure 4.6: Cycles for the Permutation $\mathbf{L}_2^8\mathbf{x}$

implement the permutation corresponding to the cycle (2,3,5). A single permutation usually

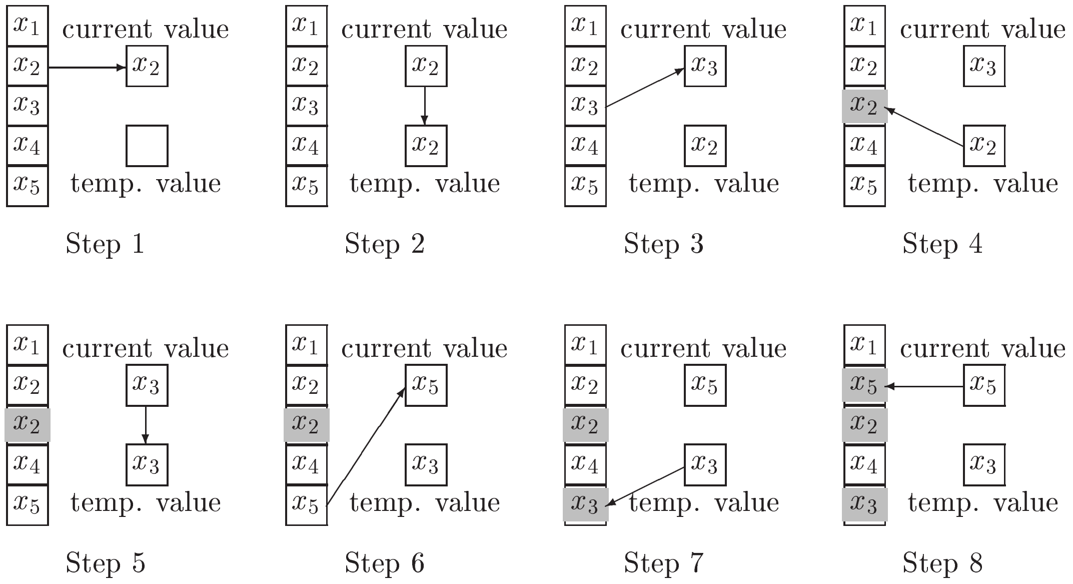


Figure 4.7: Implementation of the Permutation Corresponding to the Cycle (2,3,5)

cannot be represented with one cycle only. Therefore, in order to store multiple cycles in a single array, Egner's program uses a negative integer to indicate the end of each cycle.

4.2 Methodology

4.2.1 Improving the Order of the Twiddle Factors

In Egner's program the twiddle factors are computed and stored in the plan nodes in the following order:

$$w_{i+1} = e^{(-j2\pi/N)i}, \quad i = 0, \ldots, N-1, \tag{4.16}$$

where w_i is the i-th element in the array of twiddle factors. However, when an FFT is computed, the twiddle factors are accessed in a different order, so index computation for the array of twiddle factors must be performed as the FFT is being computed. In order to eliminate the need for index computation, Egner's program was modified so that the twiddle factors are stored in the order that they are accessed. That order, which corresponds to the diagonal of the twiddle factor matrix $\mathbf{T}_m^{mn}$, is the following:

$$w_{i+k\cdot m+1} = e^{(-j2\pi/N)i\cdot k}, \quad i = 0, \ldots, m-1, \quad k = 0, \ldots, n-1. \tag{4.17}$$

4.2.2 Making the Program More Recursive

In order to use the data cache more efficiently, the iterative nature of Egner's program was removed. This was done in two steps. The first step was to modify Egner's code modules for computing small-FFTs so that instead of performing the computation in Equation (4.1), the small-FFT code modules simply compute

$$\mathbf{x} \leftarrow \mathbf{F}_N \mathbf{x} \tag{4.18}$$

at a particular stride. The second step was to modify Egner's executor program. The executor program was modified so that when it encounters a node of type Cooley-Tukey, it computes

$$\begin{aligned} \mathbf{x} &\leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{x}, & (4.19)\\ \mathbf{x} &\leftarrow (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{T}_S^{RS}\mathbf{x}, & (4.20)\\ \mathbf{x} &\leftarrow \mathbf{L}_S^{RS}\mathbf{x}, & (4.21)\end{aligned}$$

where $\mathbf{x}$ is the array that contains the data to be transformed. The executor performs the S $\mathbf{F}_R$'s in Equation (4.19) by calling itself recursively n times to evaluate the left-child node of the current node. Similarly, the executor performs the R $\mathbf{F}_S$'s in Equation (4.20) by calling itself R times recursively to evaluate the right-child node of the current node.

The executor program was also modified so that when it encounters a node of type small-FFT, it performs the computation in Equation (4.18), such that the FFT is performed in-place at a particular stride. The computation is performed with a single call to one of the modified small-FFT code modules.

4.2.3 Adding the No-Twiddle Codelets from FFTW

The next modification made to Egner's program was to replace the small-FFT code modules with no-twiddle codelets from FFTW. This is a straight-forward modification that requires only that the small-FFT nodes in the plans be changed so that they include pointers to FFTW codelets instead of to Egner's small-FFT code modules.

4.2.4 Untransposing the Cooley-Tukey Algorithm

Egner's program was also modified so that it no longer uses the transpose of the Cooley-Tukey algorithm. This change is made by modifying the executor program. The executor program is modified so that when it encounters a node of type Cooley-Tukey, it computes

$$\begin{aligned} \mathbf{x} &\leftarrow \mathbf{L}_R^{RS}\mathbf{x}, & (4.22)\\ \mathbf{x} &\leftarrow \mathbf{T}_S^{RS}(\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{x}, & (4.23)\\ \mathbf{x} &\leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{x}, & (4.24)\end{aligned}$$

where $\mathbf{x}$ is the array that contains the data to be transformed.

4.2.5 Adding the Twiddle Codelets from FFTW

In order to insert the twiddle codelets from FFTW, it is first necessary to untranspose the Cooley-Tukey algorithm, since each twiddle codelet computes

$$\mathbf{x} \leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}\mathbf{x}, \tag{4.25}$$

and the expression $(\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}$ does not appear in the transposed version of the algorithm shown in Equation (4.8). The method used to untranspose the algorithm was explained in the previous section. The next step in inserting the twiddle codelets into Egner's program is to create a new type of plan node. The new type of plan node is called a *twiddle node*, and is illustrated in Figure 4.8. An entire plan that includes two twiddle nodes is shown in

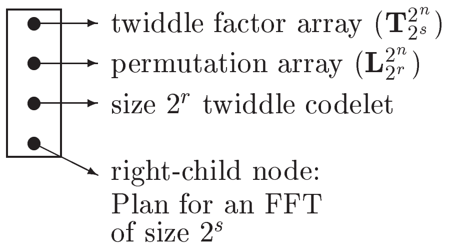

twiddle factor array ($\mathbf{T}_{2^s}^{2^n}$)
permutation array ($\mathbf{L}_{2^r}^{2^n}$)
size 2^r twiddle codelet
right-child node: Plan for an FFT of size 2^s

Figure 4.8: A Twiddle Node

Figure 4.9. The plan corresponds to the decomposition shown in Figure 4.4.

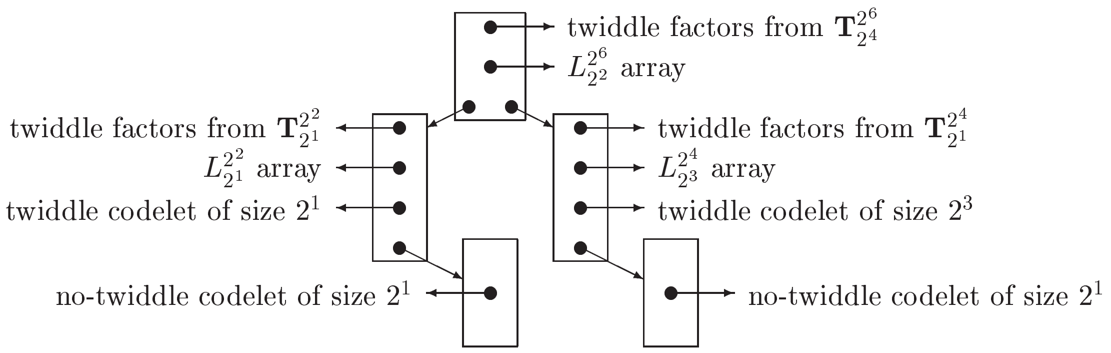

twiddle factors from $\mathbf{T}_{2^4}^{2^6}$
$L_{2^2}^{2^6}$ array
twiddle factors from $\mathbf{T}_{2^1}^{2^2}$
$L_{2^1}^{2^2}$ array
twiddle codelet of size 2^1
twiddle factors from $\mathbf{T}_{2^1}^{2^4}$
$L_{2^3}^{2^4}$ array
twiddle codelet of size 2^3
no-twiddle codelet of size 2^1
no-twiddle codelet of size 2^1

Figure 4.9: Plan with Twiddle Nodes Corresponding to the Decomposition in Figure 4.4

The last step in inserting the twiddle codelets is to modify the executor program so that it can accommodate the twiddle codelets. The executor program is modified so that when it encounters a node of type *twiddle*, it performs the following computations:

$$\begin{aligned} \mathbf{x} &\leftarrow \mathbf{L}_R^{RS}\mathbf{x}, && (4.26)\\ \mathbf{x} &\leftarrow (\mathbf{I}_R \otimes \mathbf{F}_S)\mathbf{x}, && (4.27)\\ \mathbf{x} &\leftarrow (\mathbf{F}_R \otimes \mathbf{I}_S)\mathbf{T}_S^{RS}\mathbf{x}, && (4.28) \end{aligned}$$

where $\mathbf{x}$ is the array containing the data to be transformed. The executor performs the R $\mathbf{F}_S$'s in Equation (4.27) by recursively calling itself R times to evaluate right-child node of the current node. The computation in Equation (4.28) is performed with a single call to the twiddle codelet associated with the current node.

4.3 Results

Figure 4.10 shows the dynamic programming (DP) optimal decompositions for an unmodified version of Egner's program. As shown in the figure, the optimal decompositions tend to be somewhat balanced.

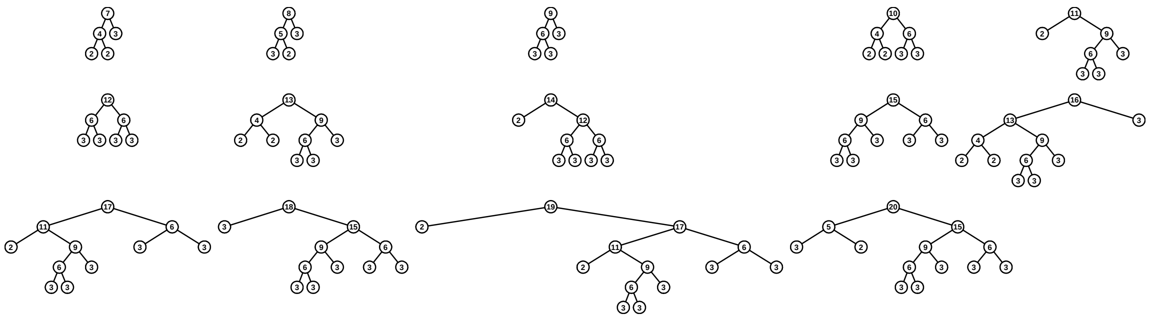


Figure 4.10: DP Optimal Trees Found Using Egner's FFT Program

Figure 4.11 shows the improvement gained when the precomputed twiddle factors are stored in the order in which they are accessed instead of in the order that Egner originally used. The improvement in run-time is as much as 50% for FFTs of size 2^4 through 2^9, but is only about 7% for FFTs of size 2^{15} through 2^{20}.

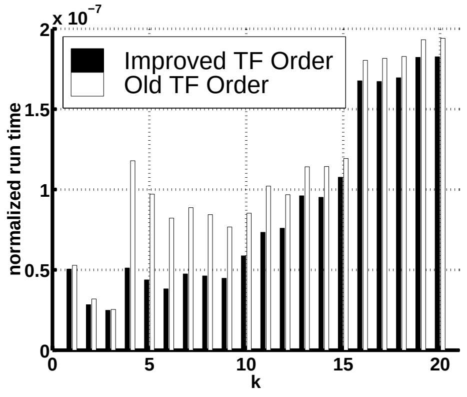


Figure 4.11: Performance Comparison for Two Twiddle Factor Orderings

As shown in Figure 4.12, the DP optimal decompositions when the twiddle factor ordering is improved have a similar topology to the DP optimal decompositions for the original twiddle factor ordering, shown in Figure 4.10.

Figure 4.13 shows the improvement gained by making Egner's program more recursive. As shown in the figure, there is little improvement for FFTs of size 2^1 through 2^{15}, but there is up to 30% improvement for the FFTs of size 2^{16} through 2^{20}. The reason for the improvement is that the more recursive algorithm uses the data cache more efficiently than the iterative algorithm. The reason that there is no improvement for FFT sizes smaller than 2^{16} is that the size of the level-two cache on the computer used for the experiment is 512 kB, or 2^{19} bytes, and the size of each double precision complex number is 2^4 bytes. Therefore, an entire array of up to 2^{15} complex numbers can fit in the level-two cache.

Figures 4.14 and 4.15 show the DP optimal decompositions for the recursive version

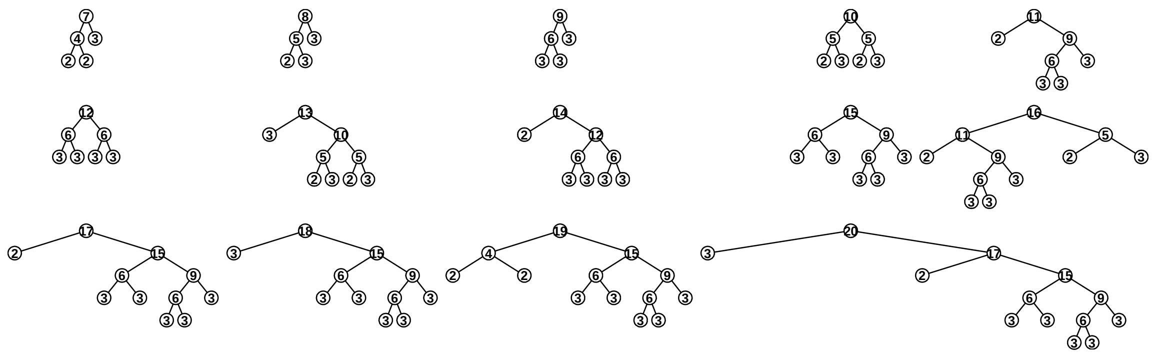


Figure 4.12: DP Optimal Decompositions Using the Improved Twiddle Factor Ordering

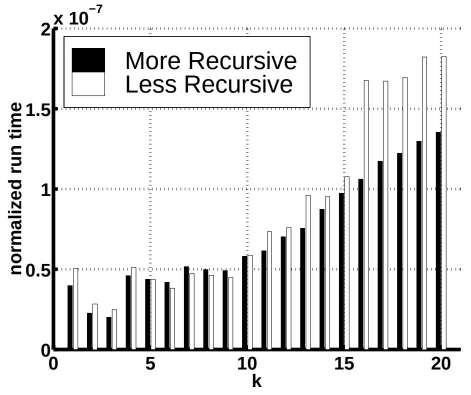


Figure 4.13: The Result of Making Egner's Program More Recursive

of Egner's program. Comparing these figures to Figure 4.12, one immediately sees that the optimal decompositions for the recursive version of the program are much more often right-expanded than the optimal decompositions for the iterative version.

Figure 4.16 shows the improvement gained when the no-twiddle codelets from FFTW are added to Egner's program. As shown in the figure, there is 30-50% improvement for most FFT sizes. The improvement is due to the optimizing techniques used in the creation of the FFTW codelets and explained in [4].

Figure 4.17 shows the DP optimal decompositions for the version of Egner's program with the no-twiddle codelets inserted. As shown in the figure, all of the trees are completely right-expanded.

Figure 4.18 shows a comparison of the run times for the untransposed version of the Cooley-Tukey algorithm with run times for the transposed version of the algorithm. As shown in the figure, within the precision of the timer, which is about 5%, the two versions of the algorithm perform equally well.

Figure 4.19 shows the DP optimal decompositions for the untransposed version of the Cooley-Tukey algorithm. Comparing Figures 4.19 and 4.17 one immediately sees that the optimal decompositions are the same no matter which version of the algorithm is used.

Figure 4.20 shows a comparison of the runtimes of Egner's program with and without the twiddle codelets from FFTW inserted. As shown in the figure, there is almost zero improvement in performance within the precision of the timer. This result suggests that the twiddle factor multiplications performed inside the twiddle codelets are not significantly

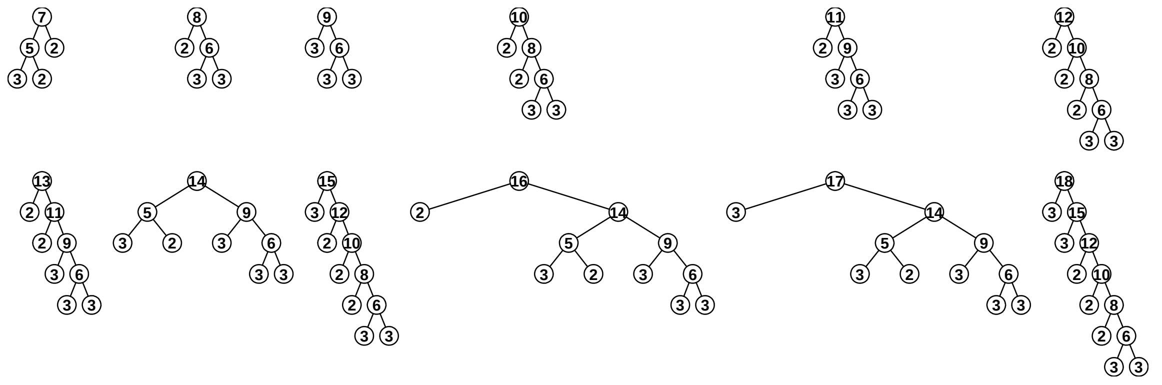


Figure 4.14: DP Optimal Decompositions For the Recursive Implementation, $n = 7, \ldots, 18$

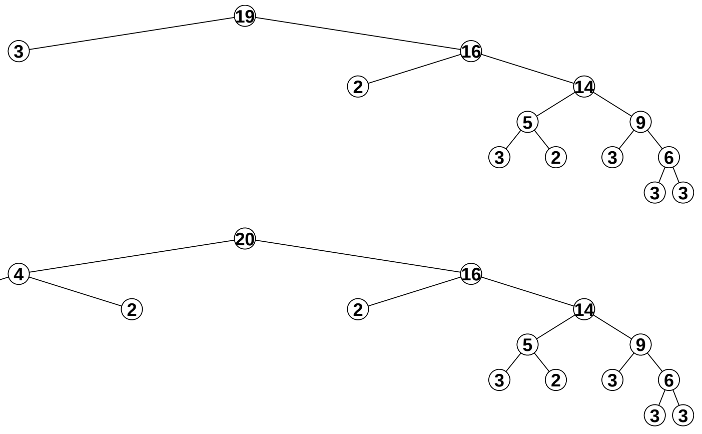


Figure 4.15: DP Optimal Decompositions for the Recursive Implementation, $n = 19, 20$

more efficient than the twiddle factor multiplications performed in a straight-forward manner outside of the codelets.

Figure 4.21 shows the DP optimal decompositions for the version of Egner's program with twiddle codelets from FFTW inserted. Comparing this figure with Figure 4.17, one sees that there is little difference in the topology of the optimal decompositions with and without the twiddle codelets other than that left child nodes of size 2^5 occur more frequently when twiddle codelets are inserted.

Figure 4.22 shows the overall improvement made to Egner's program by implementing the changes discussed above. As shown in the figure the overall improvement is typically about 50% .

Figure 4.23 shows a comparison of Egner's program with all of the modifications to FFTW. As shown in the figure, FFTW is about 25% faster for FFTs larger than 2^{15}. The gap in performance between the two programs decreases for FFTs smaller than 2^{15}.

4.4 Conclusions

The following findings were presented in this chapter:

1. The recursive version of the Cooley-Tukey algorithm is up to 30% faster than the iterative version for FFT sizes that exceed the size of the L2 cache on a pentium

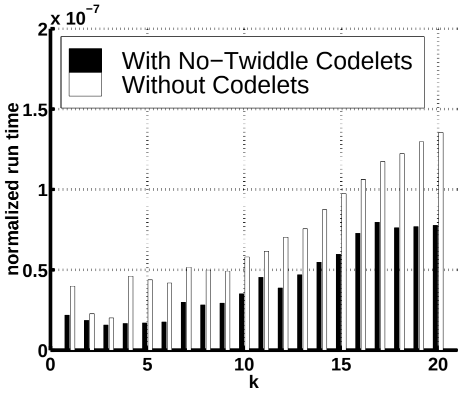


Figure 4.16: Egner's Program With and Without No-Twiddle Codelets Inserted

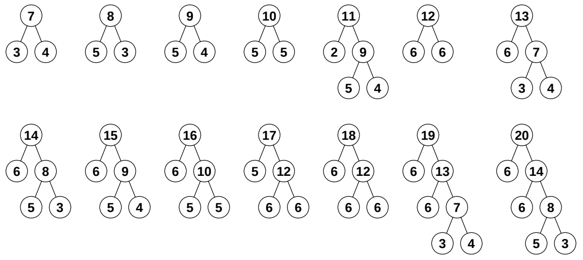


Figure 4.17: DP Optimal Decompositions with No-Twiddle Codelets Inserted

computer.

2. Storing the twiddle factors in the order that they will be accessed instead of the order of increasing powers of an N-th root of unity can result in about a 40% increase in performance for FFTs of size 2^4 through 2^9.

3. The no-twiddle codelets from FFTW result in a 30-50% increase in performance when used instead of the hand-optimized implementations of algorithms from Nussbaumer as the base cases of the recursion in an FFT program. The principal advantage that the no-twiddle codelets have over Egner's code for the small FFTs is that Egner has only code modules for base cases up to size 2^3, while FFTW has codelets for base cases up to size 2^6. The FFTW codelets are machine-generated and feature optimization that would be nearly impossible to implement by hand.

4. The twiddle factor multiplications performed inside the twiddle codelets from FFTW are not significantly more efficient than a straight-forward hand-coded implementation of the operation. In other words, an FFT implementation that uses no-twiddle codelets from FFTW as the base cases of the recursion gains very little in terms of performance if the twiddle codelets are also inserted.

5. Transposing the Cooley-Tukey algorithm results in no change in the performance of the FFT computation.

6. For an iterative implementation of the Cooley-Tukey FFT, somewhat balanced decompositions are optimal.

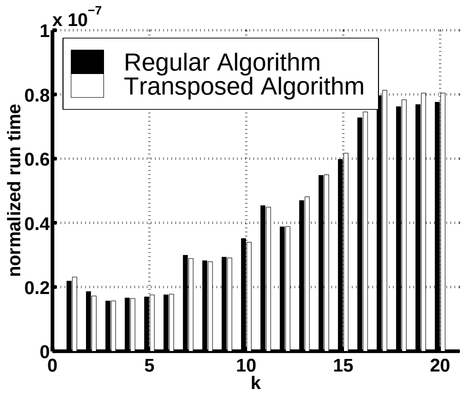


Figure 4.18: The Result of Transposing the Cooley-Tukey Algorithm

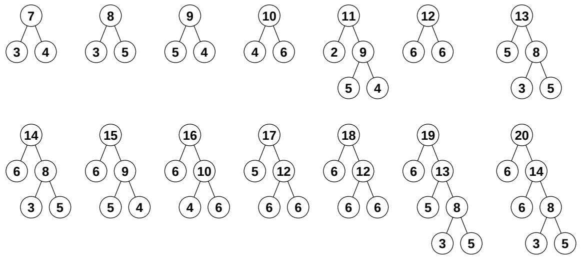


Figure 4.19: DP Optimal Decompositions Using Transposed Cooley-Tukey Algorithm

7. For a recursive implementation of the algorithm, right-expanded decompositions are optimal, even if there is no penalty associated with accessing extra arrays for non-right-expanded decompositions.

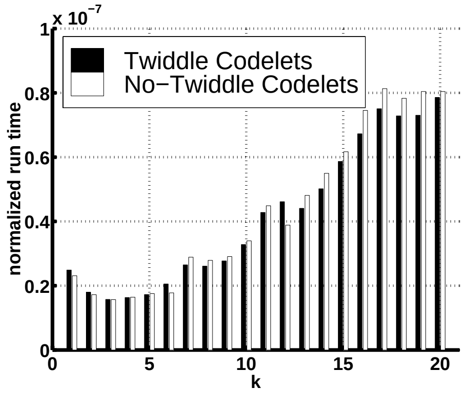


Figure 4.20: Egner's Program With and Without Twiddle Codelets Inserted

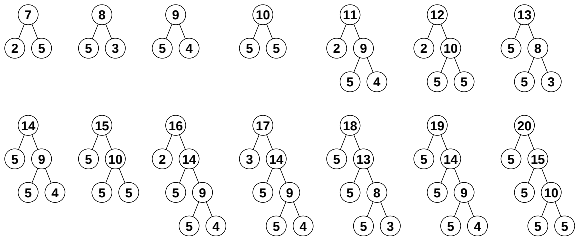


Figure 4.21: DP Optimal Decompositions with Twiddle Codelets Inserted

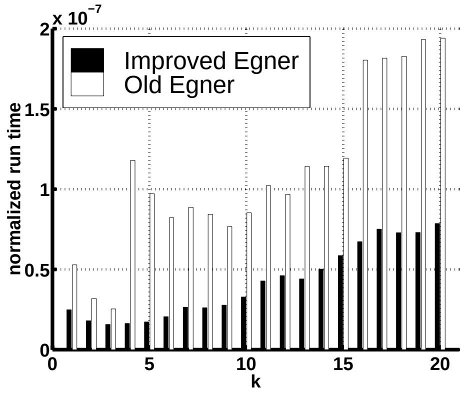


Figure 4.22: The Overall Improvement Made to Egner's Program

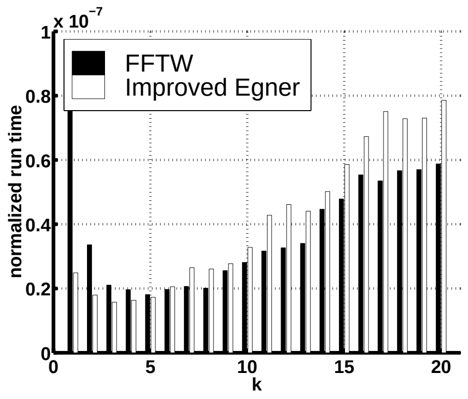


Figure 4.23: FFTW vs. the Improved Version of Egner's Program

Chapter 5

Conclusions and Future Work

In this report we investigated the effect of different Cooley-Tukey decompositions on the performance of FFT programs on a modern computer architecture. The reason for investigating the decompositions was to determine which topological features correspond to the fastest FFT implementations. In Chapter 2, it was shown that for FFTs of relatively small sizes—2^5 through 2^{10}—the runtime for the average decomposition can be up to three times the runtime for the best decomposition. It was also shown that a basic dynamic programming search performs as well as an exhaustive search at finding fast decompositions, at least for FFTs of size 2^5 through 2^{10}. Futhermore, two more sophisticated versions of dynamic programming—n-best DP and stride sensitive DP—were introduced, but neither proved more effective than basic DP at finding fast decompositions. Moreover, it was found using DP that the optimal decompositions for larger FFTs—sizes 2^{15} through 2^{20}—tend to be somewhat balanced.

In Chapter 3, it was shown that if FFTW is modified so that it can use arbitrary decompositions to compute FFTs, it is still no faster than when it can use only right-expanded decompositions. Also, it was shown that basic DP performs as well as n-best DP and stride sensitive DP at finding fast decompositions for FFTW. Furthermore, it was shown that if not for the fact that balanced decompositions require extra arrays to be accessed and twiddle factor multiplications to be performed outside of codelets, the optimal decompositions for the expanded version of FFTW would be balanced. However, under normal conditions, the best decompositions for the expanded version of FFTW are overwhelmingly right-expanded.

In Chapter 4, it was shown that a recursive version of the Cooley-Tukey algorithm may be up to 30% faster than a more iterative version for FFT sizes that exceed the size of the L2 cache on a Pentium computer. Also, it was shown that storing the twiddle factors in the order that they will be accessed instead of the order of increasing powers of an N-th root of unity can result in about a 40% increase in performance for FFT sizes of 2^4 through 2^9. Furthermore, it was shown that transposing the Cooley-Tukey algorithm results in no change in the performance of the FFT computation. Moreover, it was shown that for an iterative, in-place implementation of the Cooley-Tukey algorithm, somewhat balanced decompositions tend to be optimal, whereas for a recursive, in-place implementation of the algorithm, right-expanded decompositions are optimal. Lastly, it was shown that even with all of the optimizations made to Egner's program, it is still not as fast as FFTW. In fact, the unmodified version of FFTW was shown to be faster than all other FFT implementations discussed in this report.

In the future, we will take several steps to extend this research. In particular, we plan to conduct experiments using architectures other than Pentium, including parallel architectures. In addition, we plan to study compilers more closely in order to learn how to write source code that can be compiled into the most efficient assembly code possible. Furthermore, we plan to use tools such as a profiler, a performance counter, and a cache simulator in the design of future programs. Finally, we plan to expand the research to include transforms other than the FFT, such as the discrete cosine transform and wavelet transforms.

Bibliography

[1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematical Computation*, 19:297–301, 1965.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massechusetts, 1997.

[3] Sebastian Egner. *Zur algorithmischen Zerlegungstheorie linearer Transformationen mit Symmetrie*. PhD thesis, UniversitäT Karlsruhe (Technische Hochshule), July 1997.

[4] Matteo Frigo. A fast Fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, May 1999.

[5] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the West. Technical report, MIT Laboratory for Computer Science, September 1997.

[6] Howard W. Johnson and C. Sidney Burrus. The design of optimal DFT algorithms using dynamic programming. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31:378–387, April 1983.

[7] H. J. Nussbaumer. *Fast Fourier Transformation and Convolution Algorithms*. Springer, 2 edition, 1982.

[8] David Sepiashvili. Performance models and search methods for optimal FFT implementations. Master's thesis, Carnegie Mellon University, Pittsburgh, May 2000.

[9] Richard Tolimieri, Myoung An, and Chao Lu. *Algorithms for Discrete Fourier Transform and Convolution*. Springer-Verlag, New York, 2nd edition, 1997.