Performance Models and Search Methods for Optimal FFT Implementations

by

David Sepiashvili

May 1, 2000

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Electrical and Computer Engineering

Electrical and Computer Engineering Department
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213

Advisor: **Professor José M. F. Moura**
Reader: **Professor David P. Casasent**

This work was supported by DARPA through the ARO grant # DABT639810004

Abstract

This thesis considers systematic methodologies for finding optimized implementations for the fast Fourier transform (FFT). By employing rewrite rules (e.g., the Cooley-Tukey formula), we obtain a divide and conquer procedure (decomposition) that breaks down the initial transform into combinations of different smaller size sub-transforms, which are graphically represented as breakdown trees. Recursive application of the rewrite rules generates a set of algorithms and alternative codes for the FFT computation. The set of "all" possible implementations (within the given set of the rules) results in pairing the possible breakdown trees with the code implementation alternatives.

To evaluate the quality of these implementations, we develop analytical and experimental performance models. Based on these models, we derive methods – dynamic programming, soft decision dynamic programming and exhaustive search – to find the implementation with minimal runtime.

Our test results demonstrate that good algorithms and codes, accurate performance evaluation models, and effective search methods, combined together provide a system framework (library) to derive automatically fast FFT implementations.

Contents

1 Introduction

Efficient signal processing algorithm implementations are of central importance in science and engineering. Fast discrete signal transforms, especially the fast Fourier transform (FFT), are key building blocks. Many of these transforms, including the FFT, are based on a decomposition procedure, which gives rise to a large number of degrees of freedom for the implementation of the transform. The performance models and search methods presented in this thesis use these degrees of freedom to generate automatically very efficient implementations for these transforms.

Motivation and Related Work

Discrete-time signal processing (SP) plays and will continue to play an important role in today's science and technology. SP applications often require real-time signal processing using sophisticated algorithms usually based on discrete SP transforms, [24].

Much research has been done in optimizing these algorithms. A review assessment of these efforts is given in [13]. Most of this work is concerned with **minimizing the number of floating-point operations (flops)** required by an algorithm, since these operations were a bottleneck in older computers. For example, there is a large amount of research done in the area of minimizing the number of floating-point operations required to compute the discrete Fourier transform (DFT), [6].

The fast Fourier transform (FFT) is a remarkable example of a computationally efficient algorithm, first introduced in modern times by Cooley and Tukey, [3]. The standard way of computing the 2-power FFT, presented in many standard textbooks, is the radix-2 FFT, [6]. Radix-4 and radix-8 algorithms lead usually to faster FFT versions. Mixed radix algorithms have also been used, [19, 20, 23, 29]. It is well understood that, in general, radix-4 and radix-8 algorithms give about 20-30% improvement over the radix-2 FFT.

Another fast method for computing the DFT is the prime factor algorithm (PFA) which

uses an index map developed by Thomas and Good, [14]. Prime factorization is slow when n is large, but the DFT for small cases, such as $n = 2, 3, 4, 5, 7, 8, 11, 13, 16$, can be made fast using the Winograd algorithm, [8, 16, 17, 18]. H. W. Johnson and C. S. Burrus, [15], developed a method to use dynamic programming to design optimal FFT programs by reducing the number of flops as well as data transfers. This approach designs custom algorithms for particular computer architectures.

Efficient programs have been developed to implement the split-radix FFT algorithm, [19, 20, 21, 22, 23]. General length algorithms also exist, [29, 30, 31]. For certain signal classes H. Guo and C. S. Burrus, [32, 33], introduce a new transform that uses the characteristics of the signal being transformed and combines the discrete wavelet transform (DWT) with the DFT. This transform is an approximate FFT whose number of multiplications is linear with the FFT size.

Minimizing the number of floating-point operations is of less significance with today's technology. The interaction of algorithms with the **memory hierarchy and the processor pipelines** is a source of major bottlenecks, [1, 10]. Compilers for general-purpose languages cannot efficiently tackle the problem of optimizing these interactions, as they have little knowledge about what exactly the algorithms are computing, [34]. Computer architecture researchers are mostly concerned with developing new architectures, optimized for doing a set of predefined tasks, and are much less concerned with creating efficient algorithms on existing platforms. Thus, the problem of developing efficient algorithms with good memory interaction patterns is left to the algorithm developer, who must have a very good understanding of the computer architecture. Development of such algorithm implementations by hand is an error-prone and time consuming task, and, in general, is platform dependent, which makes the code not portable to other computing platforms.

The development of algorithms that are self-adaptable to any existing platform has been an active area of research in the past few years, [1, 10]. Portable packages have been developed for computing the one-dimensional and multidimensional complex DFT. These

algorithms tune their computation automatically for any particular hardware on which they are being used.

A very effective general length FFT system, called **FFTW**, was developed by Frigo and Johnson, [10, 11, 12]. It is faster than most of the existing DFT software packages, including FATPACK, [35], and the code from Numerical Recipes, [8]. FFTW is restricted to the Fourier transform and does not try to formulate optimization rules and to apply these rules to other SP transforms in order to obtain efficient implementations.

This thesis is part of the SPIRAL project, [1], a recent effort to create a self-adapted library of optimized implementations of SP algorithms. It uses a specialized signal processing language, SPL, an extension of TPL, [2], to formulate signal processing applications in a high-level mathematical language, and utilizes optimization rules to automatically generate implementations that are efficient in the given computational platform. SPIRAL is described in the following section.

SPIRAL

SPIRAL (Signal Processing Algorithms Implementation Research for Adaptive Libraries) is an interdisciplinary project between the areas of signal processing, computational mathematics, and computer science.

SPIRAL's goal is to develop an optimized portable library of SP algorithms suitable for numerous applications. It aims to generate automatically highly optimized code for signal processing algorithms for many platforms and a large class of applications. Its approach is to use a specialized SP language and a code generator with a feedback loop, which allows the systematic exploring of all possible choices of formula and code implementation and chooses the best combination.

SPIRAL's approach recognizes that algorithms for many SP problems can be described using mathematical formulas. This allows for easy generation of a large number of mathematically equivalent algorithms (formulas) which, however, have different computational

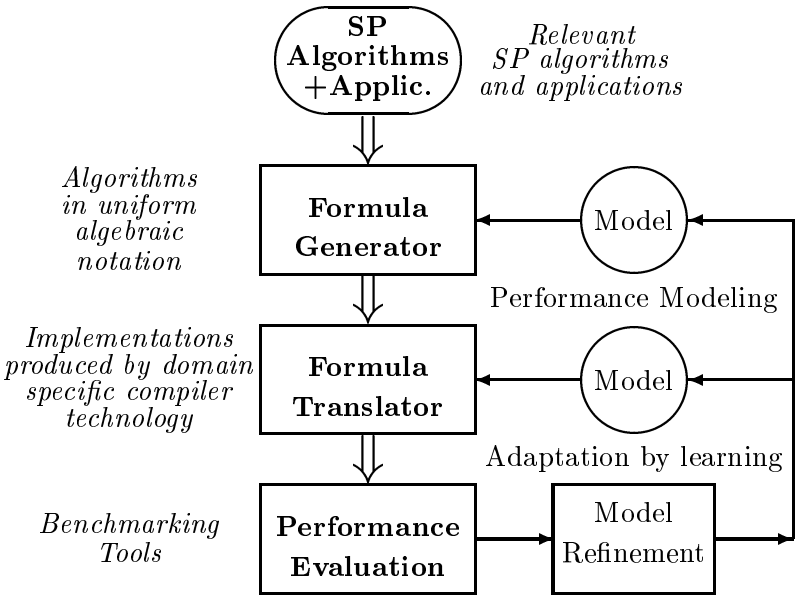


Figure 1: SPIRAL Modules

performance in different computing environments. In Figure 1, we show the architecture of SPIRAL.

The Formula Generator block generates a large number of mathematically equivalent algorithms. A second degree of freedom is provided by the Formula Translator block that creates automatically for each formula a code implementation. To determine the "optimal" implementation, we envision searching over all possible formulas and all possible implementations. To avoid such exhaustive search, SPIRAL develops a learning mechanism in a feedback loop. This learning module combines predictive models with machine learning algorithms, and determines what formula and what implementation should be tested out in the self-adaptation mode of the library. The actual benchmark results of running a chosen formula and its implementation are used by the learning module to update SPIRAL's predictive models.

Our thesis is within the SPIRAL framework, and our results are a step towards the ultimate goal of automatically determining optimized implementations for fast discrete signal transforms on large classes of existing computational platforms. The next paragraph focuses on our contributions.

Research Goal

The main goal of this thesis is to develop methodologies for finding fast implementations of signal processing discrete transforms within the framework of the 2-power point fast Fourier transform (FFT), [6]. The primary reasons for choosing this transform are its wide use in many practical applications and the fact that, as it has been studied extensively and there has been a large amount of research done in this area, it provides us with very well optimized packages to compare our results against.

In line with this goal, we formulate the following objectives:

1. To analyze FFT algorithms derived from the Cooley-Tukey formula, [3], and to create their efficient codes, supporting the arbitrary breakdown of the initial n-point FFT.

2. To develop analytical and experimental performance models for evaluating any n-point FFT implementation (combination of code and breakdown procedure), and determine the range of values of n where these models are applicable.

3. Based on the performance models that we develop, derive search methods over the set of possible Cooley-Tukey breakdown procedures, and find the best runtime implementation for large class of uniprocessor computational platforms.

Thesis Overview

In this **Introduction** we discussed the motivation and stated the goals of our research, dwelled on the major contributions of this thesis to the project SPIRAL, and gave a brief review assessment of related work.

In **Chapter 2** we provide the reader with relevant background information and terminology used throughout the thesis. We also outline the methodology necessary for the understanding of the subsequent chapters.

Chapter 3 considers different algorithms that are mathematically equivalent when computing the DFT. The chapter also addresses the process of finding the best possible code for a given algorithm.

In **Chapter 4** we design a benchmark strategy for runtime performance evaluation of different implementations.

In **Chapter 5** we present different analytical performance predictive models that are based on platform dependent parameters.

Chapter 6 introduces the notion of a search space of FFT implementations, and discusses search methods – several versions based on dynamic programming and exhaustive search – that are based on our performance models. These methods search over the space for the optimal implementation.

In **Chapter 7** we present our test results and analyze different algorithm codes. We evaluate our analytical performance models, define the range of optimal applications for dynamic programming, and compare our results against existing packages.

Conclusions and future work are discussed in **Chapter 8**.

2 Background Information and Methodology

This chapter provides the reader with background information necessary for understanding the material presented here. It also introduces the terminology used throughout the thesis, and gives a brief account of our methodology.

2.1 Specialized SP Programming Language (SPL)

Motivation Many signal processing applications face a quandary: they require real-time processing, while their processing algorithms are computationally heavy. Many such applications are implemented on special purpose hardware with code written in general-purpose low-level languages, such as assembly or C, in order to obtain very efficient implementations. Consequently, code becomes very difficult to write, and is machine dependent. When writing SP algorithms in general purpose programming languages, programmers cannot rely on a generic compiler to do the job of optimizing their code. The generic compiler is limited by the syntax of that language, and cannot explore all the inherent properties of SP applications that allow their fast computation. Thus, creation of an abstract programming language specifically designed for SP algorithms is desirable. It allows programmers to design and implement the SP algorithms at a high level, avoiding low-level implementation details, and to make their code portable. SPIRAL's SPL is an example of such a language, [2].

SPIRAL's SPL In the framework of SPIRAL, [1], (see Chapter 1), classes of fast SP algorithms are represented as mathematical expressions (**formulas**) using general mathematical constructs. For these purposes, a specifically designed high-level SP programming language, SPL, is being developed. Formulas then can be rewritten into mathematically equivalent ones by applying mathematical properties (**rewrite rules**) of these constructs. This way, SPIRAL systematically generates mathematically equivalent formulas, which can be translated into programs and compared in terms of their performance in order to find

the best one.

Basic constructs of the SPL language are:

$\bullet$	Matrix Multiply	I_n	Identity Matrix of size n
$\otimes$	Tensor Product	P_n	Permutation Matrix
$\oplus$	Direct Sum	D_n	Diagonal Matrix
		F_n	SP Transform Matrix

Examples of fast SP algorithms that can be represented in this notation are: Discrete Fourier Transform (DFT), Discrete Zak Transform (DZT), Discrete Cosine Transform (DCT), Walsh-Hadamard Transform (WHT), just to mention a few, [1, 2].

2.2 Tensor Products, Direct Sums, and Permutations

In the sequel, we will often have the occasion of indexing matrices. Sometimes the indexing is just a counting index, e.g., A_i, $i = 1, \ldots, n$, but often it will indicate the dimensions of the matrices (if square), e.g., A_r (matrix A of size $r \times r$). We hope the context will make clear what the index stands for.

Tensor Product If A and B are $m_1 \times m_2$ and $n_1 \times n_2$ matrices, respectively, then their tensor or Kronecker product is defined, [4], as

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & a_{1,2} \cdot B & \ldots & a_{1,m_2} \cdot B \\ a_{2,1} \cdot B & a_{2,2} \cdot B & \ldots & a_{2,m_2} \cdot B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m_1,1} \cdot B & a_{m_1,2} \cdot B & \ldots & a_{m_1,m_2} \cdot B \end{pmatrix},$$

i.e., A determines the coarse structure, and B defines the fine structure of $A \otimes B$.

Tensor Product Properties Standard properties of the tensor product, [2, 4], include the following:

1. DISTRIBUTIVE $(A + B) \otimes C = A \otimes C + B \otimes C$
2. ASSOCIATIVE $(A \otimes B) \otimes C = A \otimes (B \otimes C)$
3. SCALAR MULTIPLICATION $\alpha(A \otimes B) = (\alpha A) \otimes B = A \otimes (\alpha B)$, where α is a scalar.
4. TRANSPOSE $(A \otimes B)^T = B^T \otimes A^T$
5. INVERSE $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$
6. GROUPING $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$
7. EXPANSION $A \otimes B = (A \otimes I)(I \otimes B)$

Direct Sum The direct sum, [7], of n matrices A_i, $i = 1, \ldots, n$, not necessarily of the same dimensions, is defined as the block diagonal matrix

$$\bigoplus_{i=1}^{n} A_i = A_1 \oplus A_2 \oplus \ldots \oplus A_n = \begin{pmatrix} A_1 & & & 0 \\ & A_2 & & \\ & & \ddots & \\ 0 & & & A_n \end{pmatrix}.$$

Stride Permutation Let $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ be a vector of length $n = r \cdot s$. The $n \times n$ matrix is a permutation of stride s, written as L^n_s, if it permutes the elements of $\mathbf{x}$ as

$$\begin{aligned} i &\mapsto i \cdot s \bmod (n-1), \ i < n-1, \\ n-1 &\mapsto n-1. \end{aligned}$$

For example, if $\mathbf{x} = (x_0, x_1, x_2, x_3, x_4, x_5)$, then $L^6_3 \cdot \mathbf{x} = (x_0, x_3, x_1, x_4, x_2, x_5)$.

Stride Permutation Properties

1. $A_r \otimes B_s = L^{rs}_r (B_s \otimes A_r) L^{rs}_s$

2. $L^{rs}_s \cdot L^{rs}_r = I$

3. $L^{rst}_s \cdot L^{rst}_t = L^{rst}_{st}$

Bit-reversal Permutation Let $n = 2^k$. The $n \times n$ matrix is a bit-reversal permutation, written as B_n, if it is given by

$$B_{2^k} = \prod_{i=1}^{k} (I_{2^{k-i}} \otimes L^{2^i}_2).$$

Suppose $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$. Bit-reversed permutation of the elements of $\mathbf{x}$, $B_n \cdot \mathbf{x}$, can be described in a neat way via the binary representation of the indices. We take the indices of the elements of $\mathbf{x}$, and write them in binary form with leading 0's so that all indices have exactly k digits. Then, we reverse the order of the digits, and convert back to decimal. For example, if

$$\begin{aligned} \mathbf{x} &= (x_{000b}, x_{001b}, x_{010b}, x_{011b}, x_{100b}, x_{101b}, x_{110b}, x_{111b}), \text{ then} \\ B_8 \cdot \mathbf{x} &= (x_{000b}, x_{100b}, x_{010b}, x_{110b}, x_{001b}, x_{101b}, x_{011b}, x_{111b}). \end{aligned}$$

2.3 Discrete Fourier Transform (DFT)

Suppose $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ is a finite-duration discrete time signal, n samples long. We define the discrete Fourier transform (DFT), [6], to be the signal $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})^T$ also n samples long given by the formula

$$y_l = \sum_{i=0}^{n-1} x_i \cdot e^{-j2\pi \cdot il/n}, \qquad i = 0, 1, \ldots, n-1.$$

The DFT can also be written in matrix notation. We define $w_n = e^{-j2\pi/n}$. Then the DFT matrix F_n will be of size $n \times n$, and can be written as

$$F_n = (w_{i,j})_{i,j=0,1,\ldots,n-1}, \quad \text{where} \quad w_{i,j} = w_n^{i \cdot j}.$$

Due to the periodicity of the complex exponents $e^{-j2\pi \cdot il/n}$, only n entries of the DFT matrix have distinct values. The entries are the n-th roots of unity, and are the solutions to the equation $t^n - 1 = 0$ over the field of complex numbers. In particular, w_n is itself a primitive root of unity. The n-th roots of unity can be generated by raising a primitive root of unity to an appropriate power.

The periodicity property can be written as

$$w_n^i = w_n^{i \bmod n}.$$

The DFT in matrix notation becomes

$$\mathbf{y} = F_n \cdot \mathbf{x}.$$

2.4 Fast Fourier Transform (FFT)

An efficient class of algorithms for computing the DFT was discovered by Cooley and Tookey (1965), [3], and has come to be known as the fast Fourier transform (FFT).

Cooley-Tukey Formula Suppose $n = r \cdot s$. Then the Cooley-Tukey algorithm for computing the DFT in the tensor product notation is given by

$$F_n = (F_r \otimes I_s) \cdot T_s^n \cdot (I_r \otimes F_s) \cdot L_r^n, \tag{1}$$

where F_n is the $n \times n$ DFT matrix, L_r^n is a stride permutation matrix, and T_s^n is a diagonal matrix with certain powers of a primitive root of unity w_n on its diagonal.

T_s^n is called the **twiddle matrix** and has the following structure:

$$T_s^n = \bigoplus_{i=0}^{r-1} \mathrm{diag}(w_n^0, w_n^1, \ldots, w_n^{s-1})^i,$$

i.e., it is the direct sum of diagonal matrices, whose diagonal elements are n-th roots of unity in a certain order.

Relevant Properties (Formula Rewrite Rules) Applying properties of the tensor product and combining some of the properties above, we obtain the formula rewrite rules, [1, 2].

1. $(F_r \otimes I_s) = L^n_r \cdot (I_s \otimes F_r) \cdot L^n_s$
2. $L^n_s \cdot L^n_r = I_n$
3. $L^n_s \cdot T^n_s = T^n_r \cdot L^n_s$
4. $I_r \otimes F_s = \bigoplus_{i=0}^{r-1} F_s$
5. $T^n_s = \bigoplus_{i=0}^{r-1} \left(\bigoplus_{l=0}^{s-1} w_n^{il} \right)$

Asymptotic Performance of FFT and DFT The brute force computation of the DFT requires the multiplication of the $n \times n$ DFT matrix by a vector of size n. Asymptotically this direct multiplication takes order of n^2 floating point operations (flops), which we denote by $\Theta(n^2)$.

The FFT algorithm, given by Equation (1), is recursive. It takes a problem of size $n = r \cdot s$ and represents it in terms of smaller problems of sizes r and s. The FFT algorithm with the least number of multiplications has as many factors as possible, i.e., it is the 2-power FFT ($n = 2^k$). The 2-power FFT is the most commonly used FFT and requires $O(n \cdot \log_2 n)$ flops.

2-power FFT Because of its popularity and asymptotic superior performance, we will concentrate only on cases when $n = 2^k$. For such cases the Cooley-Tukey formula given by Equation (1) can be rewritten as follows:

$$F_{2^k} = (F_{2^q} \otimes I_{2^{k-q}}) \cdot T^{2^k}_{2^{k-q}} \cdot (I_{2^q} \otimes F_{2^{k-q}}) \cdot L^{2^k}_{2^q}, \quad \text{where} \quad 2^k = 2^q \cdot 2^{k-q}. \tag{2}$$

The factorization of problem size 2^k into problem sizes 2^q and 2^{k-q} is completely determined by the values of k and q.

2.5 Breakdown Trees

General Problem Formulation Like the FFT, many other signal-processing transforms have a recursive structure, which allows deriving fast algorithms for their computation. On each step of a recursion, there is the degree of freedom to recurse further, or to stop the recursion, and compute the elementary transforms from the definition (recursion initial conditions). A strategy for choosing these recursion parameters is referred to as a **breakdown strategy**. Each breakdown strategy represents a certain fast algorithm for computing the respective signal transform.

From a mathematical point of view, all breakdown strategies represent the same transform. However, the implementation of different breakdown strategies will show a significant difference in runtime **performance**. Our problem is to find the best possible breakdown strategy, i.e., the one that will optimize the runtime performance.

The parameters defining the optimal breakdown strategy are platform and application dependent, and cannot be set a priori. Thus, the performance of a breakdown strategy is a function of the platform on which the transform is to be used. A natural way of representing a breakdown strategy is a tree, which we will call a **breakdown tree**. Note that, in general, breakdown trees will not be binary. The set of all possible breakdown trees forms the breakdown tree **search space**. Then the problem of choosing the best breakdown strategy is equivalent to identifying the best breakdown tree in the search space.

2-power FFT The Cooley-Tukey formula, given by Equation (2), reveals the recursive nature of the DFT. Thus, everything we described in the previous paragraph applies to the DFT. In the case of the DFT, the degree of freedom at each step of the recursion that is available is the choice of q given k ($q \in \mathrm{Z} : 0 < q < k$).

The choices that we make at each step of the recursion can be represented as a breakdown tree. Figure 2 shows a few examples of such trees. We start with a DFT of size 2^7. Each node q in the tree represents the computation of a 2^q-point DFT. The root node

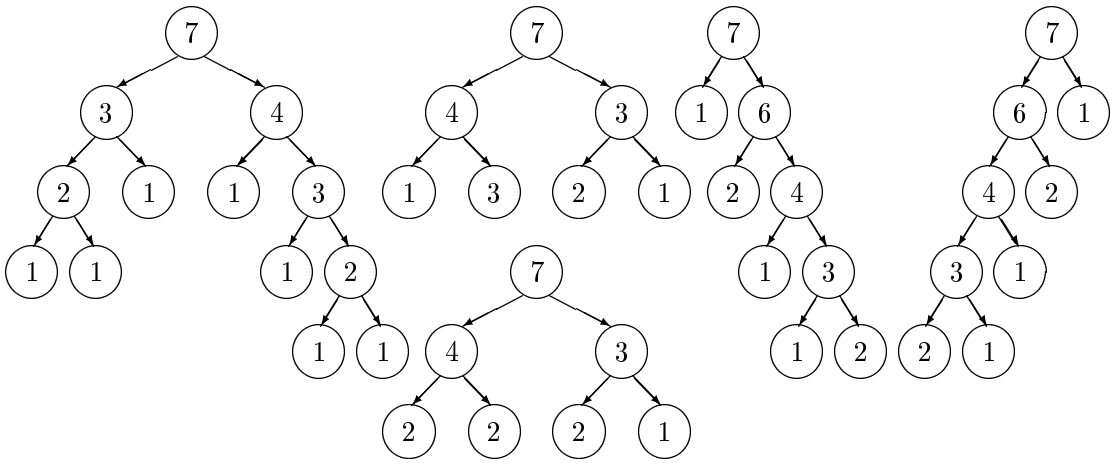


Figure 2: Examples of breakdown trees for an FFT of size 2^7

is labeled by the value $k = \log_2(n) = 7$. The two children of this node are labeled by a decomposition of k into two positive integers. For example, in the first tree of Figure 2, $7 = 3+4$. The left child is labeled with $q = 3$, and the right child is labeled with $q = 4$. The tree breakdown continues recursively. The recursion stops when we choose $q = 0$, which means a computation using the DFT definition. The third tree in Figure 2 is a special case, as it breaks down only to the right. Its left node is always a leaf, while its right node breakdown continues recursively. We call it a **right-most breakdown tree**. Similarly we define a **left-most breakdown tree**, shown last in Figure 2. Another interesting case is a **balanced breakdown tree**. We call a tree to be balanced, if any left node differs at most by 1 from the corresponding right node. Such a tree is shown below the second tree in Figure 2. Note that the second tree is not balanced, as $4 = 1 + 3$.

2.6 Methodology of the Thesis

The SPIRAL library aims at deriving systematic methodologies to create optimized signal processing algorithm implementations that self-adapt to the computational platform. Developing these efficient discrete signal transform algorithms is a challenge. Their compu-

tation needs to be tuned automatically to the unknown a priori parameters of the particular hardware. Determining an optimal implementation by exhaustive search is not feasible due to the extremely large number of possible alternatives. To solve these issues new concepts and methods are required.

This thesis is within the framework of SPIRAL, but is only one of the components of SPIRAL. For one thing, we focus on a single SP discrete transform, namely, the DFT. Secondly, we do not automate the code generation. We perform the operations assigned to the Formula Generation and Formula Translation blocks in Figure 1 manually, while in SPIRAL they will be automated. Finally, we do not consider the learning mechanism with the feedback loop that combines predictive models with machine learning algorithms and determines what formulas and what implementations should be tested out in the self-adaptation mode of the library. Our objectives are much narrower and modest, as described in Chapter 1.

We take a system approach to the problem. We do not choose the optimal code and the optimal breakdown tree independently of each other. Rather, we combine all possible code alternatives with all possible allowed breakdown trees, and then, employing performance models and search strategies that we develop, we choose their best combination. By combining good algorithms and good codes with accurate performance evaluation models and effective search methods, we obtain efficient FFT implementations.

Figure 3 discusses the functional block diagram of our system framework. At the top we have the formula for computing the FFT $\mathbf{y} = \mathbf{F_n} \cdot \mathbf{x}$. This is the Cooley-Tukey formula. By applying the rewrite rules to it, given in Section 2.1, we generate many algorithms for the FFT. Even though the Cooley-Tukey formula itself can also be viewed as a rewrite rule, in our system framework we decided to use it as a starting point for computing the FFT, as we are not considering any other major formulas. Also, algorithms in reality are nothing else than mathematical formulas, consisting of basic constructs of the SPL language (see Section 2.1). These algorithms support the possible breakdown trees, as presented in

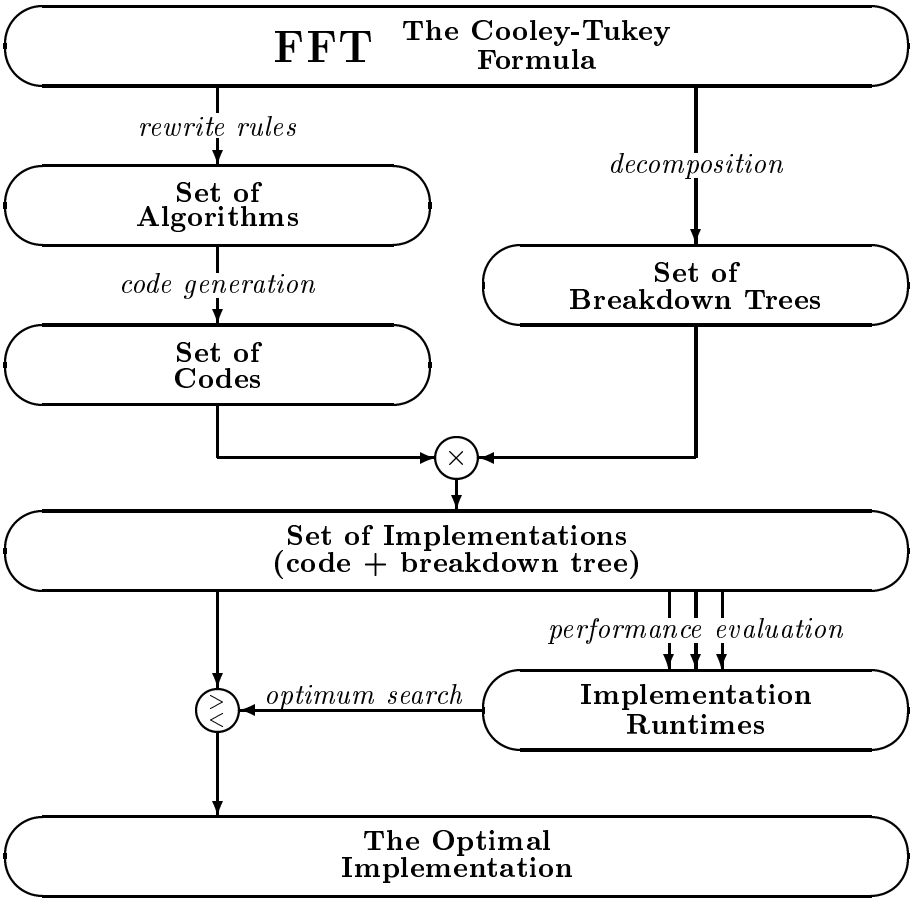


Figure 3: Functional block-diagram of the optimal FFT implementation

Section 2.5. Then, we generate possible alternative codes for these algorithms. These codes depend on the computational platform. This gives us a set of all possible codes, which support arbitrary breakdown trees for the FFT computation. This corresponds to the first two blocks on the left in Figure 3.

The right top block of Figure 3 (a set of all possible breakdown trees) is generated from the Cooley-Tukey formula by a divide and conquer procedure (decomposition). The transform is presented by breaking it down into smaller size sub-transforms, and combining the solutions to these sub-transforms to form the actual solution. This defines a breakdown tree. By varying the combination of different size sub-transforms, we get the set of all breakdown trees.

Once we get the set of all breakdown trees, and the set of all codes, we form pairs of codes and breakdown trees, which form the set of all possible implementations. The reason for doing it lies in the fact that the optimal code cannot be determined uniquely for any breakdown tree, i.e., the optimal code depends on the breakdown tree it is used with.

The next step is to choose an optimal implementation from a set of all possible implementations. This is achieved in two steps. First, we assign the runtime estimate to each implementation employing the performance evaluation models. Secondly, we find the implementation that has the minimal runtime by applying the optimum search methods to a set of implementation runtimes. This produces the optimal implementation.

3 FFT Algorithms and Optimized Codes

3.1 Introduction

The goal of this chapter is to derive several FFT algorithms based on the Cooley-Tukey formula, and to generate their efficient codes. These codes will support any arbitrary breakdown trees of the initial n-point FFT, as opposed to implementations, which are defined as a pair of a code with a breakdown tree (see Section 2.6 for details).

Methodology We derive the code in two steps, as shown in Figure 4. First, we take a recursive **formula**, in our case, the Cooley-Tukey formula given by Equation (1) in Section 2.4. By applying to it the rewrite rules, we get mathematically equivalent formulas that we call a recursive **algorithm**. We implement the algorithm in a given programming language and obtain the **code**. Note that for a given recursive formula there are many algorithms (e.g., in-place algorithm, algorithm with temporary storage, etc.), and, for each algorithm, there are many codes (e.g., recursive, iterative, and other mixed methods).

$$\text{FORMULA} \rightarrow \text{ALGORITHM} \rightarrow \text{CODE}$$

Figure 4: Derivation of the code

To create efficient codes for these algorithms, we choose an appropriate programming language, develop highly optimized codes for the small size DFTs, and optimize the twiddle factor computation and access.

Description The chapter is organized in the following way. In Section 3.2, we introduce a pseudo-code notation that is used throughout the chapter. Then, in Section 3.3, we derive three major types of algorithms for the FFT with completely different characteristics: FFT-BR; FFT-RT; FFT-TS. We could consider many other algorithms, but the ones we study will share their characteristics. Further, in Section 3.4, we write the code for the

algorithms derived in Section 3.3 by choosing the programming language and developing highly optimized code for small size DFT's. In this section we also optimize the twiddle factor computation and access. Finally, in Section 3.5, we summarize the process of deriving efficient codes, and present conclusions.

3.2 Pseudo-code Notation

Throughout the chapter, we will use a pseudo-code notation whenever we want to introduce a code example. The pseudo-code uses a calling convention shown in Figure 5.

`fft()` is the subroutine that computes FFT's of arbitrary size. In parenthesis we specify its input arguments. The variable `n` is the size of the FFT to be computed. By `x:xs` and `y:ys` we define memory addresses for the values of the input and output vectors. This subroutine is recursive, which means that it is called from within its body. When we want to access the `i`-th element of the array, we will write `y[i]`. The symbol `w_n` is a primitive root of unity, as explained in Chapter 2. When we write `(a div b)` and `(a mod b)`, we mean that we compute the integer part of the division of a by b, and its remainder, respectively. The subroutine call `bitrev(i,n)` does generalized bit-reversal of the index i modulo n. When $n = 2^k$, a generalized bit-reversal is equivalent to binary bit-reversal. It writes an index i in the binary form, reverses the order of the bits from right-to-left to left-to-right, and converts it back to the decimal form. Additional information on the bit-reversal is given in Section 2.2.

```
fft(n,x:xs,y:ys) {
   // n is the FFT size
   // x is the start of an input vector
   // y is the start of an output vector
   // xs is the stride to step through elements of x
   // ys is the stride to step through elements of y
   ...
}
```

Figure 5: Pseudo-code notation

3.3 Cooley-Tukey: Alternative FFT Algorithms

The Cooley-Tukey formula given by Equation (1) in Section 2.4 and combined with the definition of the DFT $y = F_n \cdot x$ can be written as

$$y = (F_r \otimes I_s) \cdot T^n_s \cdot (I_r \otimes F_s) \cdot L^n_r \cdot x \quad \text{where} \quad n = r \cdot s \tag{3}$$

The matrix F_n is the $n \times n$ DFT matrix, L^n_r is a stride permutation matrix of stride r, and T^n_s is the twiddle factor matrix. It is a diagonal matrix with certain powers of a primitive root of unity w_n on its diagonal.

By applying the formula rewrite rules presented in Section 2.4 to Equation (3), we can generate many equivalent recursive algorithms. Below we present three major algorithms with completely different characteristics. We note that, although the algorithms are recursive, their implementation need not be recursive. The same algorithm may have recursive and iterative implementations.

When we implement algorithms derived from Equation (3), we will not do matrix multiplications by definition. Matrix notation is only a convenient way for representing different algorithms. For example, writing

$$L^n_r \cdot x,$$

does not mean matrix-vector multiplication, but rather accessing data elements in x at stride r instead of the conventional stride 1. In our pseudo-code notation this will be written as

```
x:r.
```

Thus, the notation

$$(L^n_r \cdot y) = y$$

means that we read elements of y at stride 1, and write them back to y at stride r.

We now present these different algorithms.

3.3.1 Recursive In-place Bit-Reversed Algorithm (FFT-BR)

By applying the rewrite rules from Sections 2.2 and 2.4, we transform Equation (3) into

$$y = \left(L_r^n\right) \cdot \left(I_s \otimes F_r\right) \cdot \left(T_r^n\right) \cdot \left(L_s^n \cdot (I_r \otimes F_s) \cdot L_r^n\right) \cdot x.$$

This recursive formula, split into four recursive steps, is given below.

Algorithm 1 (FFT-BR-O) *In-place bit-reversed output algorithm*

1. $(L_r^n \cdot y) = (I_r \otimes F_s) \cdot (L_r^n \cdot x)$
2. $y = T_r^n \cdot y$
3. $y = (I_s \otimes F_r) \cdot y$
4. $(L_r^n \cdot y) = y$ *(done outside of the recursion)*

These four steps are done for each recursive step, i.e., on each internal node of the breakdown tree. If the last step is pulled out of the recursion, we need to apply to y a special permutation called bit-reversal, described in more detail in Section 2.2.

The first three steps of this algorithm are **in-place**, i.e., the output signal is stored in the same place of the input signal. The nice property of this algorithm is that if the DFT is used to compute the convolution of two signals, then, instead of using an actual DFT, we can use a DFT with elements in the frequency domain stored in the bit-reversed order by computing the steps 1, 2, 3 only, since, when the inverse bit-reversed transform is taken, we get the convolution in the proper order, thus eliminating the need for step 4 at all.

Figure 6 shows the pseudo-code for this algorithm. In line 2 we check if the DFT of size n is computed from the definition. The decision is made based on the fixed breakdown tree we use in the context of this computation. If the decision is positive, we compute the DFT of size n from the definition in line 3 by calling the corresponding small code module subroutine, and then we exit the instance of this subroutine by jumping to line 12.

Pseudo-code 1 (FFT-BR-O) *Recursive implementation of the first three steps*

```
 1   fft(n, x : xs, y : ys) {
 2      if is_leaf_node(n)
 3         dft_small_module(n, x : xs, y : ys);
 4      else
 5         r=left_node(n);   s=right_node(n);
 6         for (ri=0;ri<r;ri++)
 7            fft(s, x + xs*ri : xs*r, y + ys*ri : ys*r);
 8         for (i=0;i<n;i++)
 9            y[ys*i] = y[ys*i] * w_n^(bitrev((i div r),s)*(i mod r));
10         for (si=0;si<s;si++)
11            fft(r, y + ys*si*r : ys*1, y + ys*si*r : ys*1);
12   }
```

Figure 6: Pseudo-code for the FFT-BR-O algorithm

Otherwise, we apply the Cooley-Tukey formula by breaking the DFT of size n into DFTs of smaller sizes r and s, where $n = r \cdot s$. This is done is lines 5-11. In line 5 we decide how to break the value of n into a product of two values. This decision is based on the breakdown tree we use. Then in lines 6-7 we call recursively the subroutine, given in line 1, r times for computing the DFT of size s. These two lines correspond to step 1 of the FFT-BR-O algorithm, presented above. In lines 8-9 we perform the multiplication by twiddle factors, which corresponds to step 2 of the algorithm. Finally, in lines 10-11 we call recursively the subroutine, given in line 1, s times for computing the DFT of size r. This corresponds to step 3 of the algorithm. Step 4 is not shown in this pseudo-code, as it is done outside of the recursion.

Another flavor of this algorithm is a recursive in-place bit-reversed input algorithm, given by

$$y = \left(L^n_r \cdot (I_s \otimes F_r) \cdot L^n_s\right) \cdot \left(T^n_s\right) \cdot \left(I_r \otimes F_s\right) \cdot \left(L^n_r\right) \cdot x.$$

This recursive formula, split into four recursive steps, is given below.

Algorithm 2 (FFT-BR-I) *In-place bit-reversed input algorithm*

1. $y = (L^n_r \cdot x)$ *(done outside of the recursion)*

2. $y = (I_r \otimes F_s) \cdot y$

3. $y = T^n_s \cdot y$

4. $(L^n_s \cdot y) = (I_s \otimes F_r) \cdot (L^n_s \cdot y)$

Figure 7 shows the pseudo-code for this algorithm. It is very similar to the pseudo-code given in Figure 6, explained above. The main difference is in the strides at which we are accessing the data (see lines 7,9 ,11).

Pseudo-code 2 (FFT-BR-I) *Recursive implementation of the first three steps*

```
 1   fft(n, x : xs, y : ys) {
 2      if is_leaf_node(n)
 3         dft_small_module(n, x : xs, y : ys);
 4      else
 5         r=left_node(n);   s=right_node(n);
 6         for (ri=0;ri<r;ri++)
 7            fft(s, x + xs*ri*s : xs*1, y + ys*ri*s : ys*1);
 8         for (i=0;i<n;i++)
 9            y[ys*i] = y[ys*i] * w_n^(bitrev((i div s),r)*(i mod s));
10         for (si=0;si<s;si++)
11            fft(r, y + ys*si : ys*s, y + ys*si : ys*s);
12   }
```

Figure 7: Pseudo-code for the FFT-BR-I algorithm

3.3.2 Recursive Algorithm for Right-Most Trees (FFT-RT)

In the previous subsection we derived two similar algorithms for computing the FFT. These algorithms required an explicit bit-reversal, but were in-place, with no additional temporary storage being required. In this subsection we derive another algorithm, with

completely different characteristics. It is out-of-place, but requires no explicit bit-reversal. It also works without any temporary storage.

By parenthesizing the Equation (3) differently, we get

$$y = \left(L_r^n \cdot (I_s \otimes F_r) \cdot L_s^n\right) \cdot \left(T_s^n\right) \cdot \left((I_r \otimes F_s) \cdot L_r^n\right) \cdot x,$$

which defines the following new algorithm.

Algorithm 3 (FFT-RT-3) *Out-of-place algorithm for right-most trees*

1. $y = (I_r \otimes F_s) \cdot (L_r^n \cdot x)$
2. $y = T_s^n \cdot y$
3. $(L_s^n \cdot y) = (I_s \otimes F_r) \cdot (L_s^n \cdot y)$

To support arbitrary trees this algorithm requires allocating additional temporary storage. As we see, step 3 is done in-place on an array y. So, if we wanted to break down recursively the computation of F_r, our algorithm would have to be in-place. However, step 1 cannot be done in-place without a temporary array, as it is accessing input and output arrays at different strides. Thus, F_r has to be the initial step of the recursion, i.e., has to be computed from the definition, without further breakdown. This limitation means that the only breakdown trees we can consider with this algorithm are right-most breakdown trees, defined in Section 2.5, Figure 2. This is the main limitation of this algorithm, since we cannot apply it to other breakdown trees. However, as we will see from the experimental results in Chapter 7, this algorithm has a very good runtime performance. In the DFTs we ran on a Pentium II machine, it outperforms all other algorithms.

The pseudo-code for this algorithm is given in Figure 8. Line 9 is very different from line 9 in Figure 6. This line does the twiddle factor multiplication. In this algorithm we do not need to do the bit-reversal on one of the indexes for computing the offset into the array of twiddle factors.

Pseudo-code 3 (FFT-RT-3) *Recursive implementation*

```
 1   fft(n, x : xs, y : ys) {
 2      if is_leaf_node(n)
 3         dft_small_module(n, x : xs, y : ys);
 4      else
 5         r=left_node(n);   s=right_node(n);
 6         for (ri=0;ri<r;ri++)
 7            fft(s, x + xs*ri : xs*r, y + ys*ri*s : ys*1);
 8         for (i=0;i<n;i++)
 9            y[ys*i] = y[ys*i] * w_n^((i div s)*(i mod s));
10         for (si=0;si<s;si++)
11            fft(r, y + ys*si : ys*s, y + ys*si : ys*s);
12   }
```

Figure 8: Pseudo-code for the FFT-RT-3 algorithm

3.3.3 Recursive Algorithm with Temporary Storage (FFT-TS)

In the previous Subsection 3.3.2, we derived an algorithm that is out-of-place, does not require an explicit bit-reversal, and works without any additional temporary storage (assuming that we are not required to do the computation in-place). But it supports only a limited set of breakdown trees, namely, only right-most trees. In this section we present a different algorithm that supports arbitrary breakdown trees. This algorithm is in-place, but has additional temporary storage requirements.

We rewrite Equation (3) in the form

$$y = \left(L^n_r \cdot (I_s \otimes F_r) \cdot L^n_s\right) \cdot \left(T^n_s\right) \cdot \left((I_r \otimes F_s) \cdot L^n_r\right) \cdot x.$$

This is the same formula that we used to derive the algorithm for the right-most trees in Subsection 3.3.2. We now split this recursive formula into three recursive steps that lead to a different algorithm.

Algorithm 4 (FFT-TS) *In-place algorithm with extra temporary storage requirements*

1. $t = (I_r \otimes F_s) \cdot (L^n_r \cdot x)$
2. $y = T^n_s \cdot t$
3. $(L^n_s \cdot y) = (I_s \otimes F_r) \cdot (L^n_s \cdot y)$

Lemma 1 *For supporting any arbitrary breakdown tree, the amount of temporary storage required by the algorithm is asymptotically* $2 \cdot n$.

Proof: Each step of the recursion requires a temporary storage of size n. Since the algorithm has to support any arbitrary breakdown tree, we calculate the minimal amount of storage required by this algorithm in the worst-case scenario. This happens when we split n as $n = \frac{n}{2} \cdot 2$ at each step of the recursion. Then, as we recurse, the total temporary storage size is $n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \leq 2 \cdot n$. As n grows, this sum converges to $2 \cdot n$. Thus the minimal amount of storage we need to support any breakdown tree is $2 \cdot n$. □

Since the FFT-TS algorithm requires temporary storage, it needs to be allocated at some point in time. A trivial implementation allocates temporary storage on the fly, i.e., allocates it at each step of the recursion. This is not efficient, as calls to subroutines for allocating memory are slow. A better approach is to allocate the memory for temporary storage once, before the computation begins. In this case, we need an algorithm for finding at each step of the recursion the address space of the temporary array not in use by the other steps of the recursion. The proof to the lemma suggests the following algorithm. At each step of the recursion we offset by $2 \cdot n$ elements in the temporary array, and use n elements starting from this offset. In this way we are guaranteed that different steps of the recursion will not be overwriting each other's results. For such algorithms the temporary storage requirement will be exactly $2 \cdot n$. This is done on line 6 of the pseudo-code for this

algorithm, shown in Figure 9. Other lines of the pseudo-code are very similar to previously explained pseudo-codes, so they are not explained here.

Pseudo-code 4 (FFT-TS) *Recursive implementation*

```
1   fft(n, x : xs, y : ys) {
2      if is_leaf_node(n)
3         dft_small_module(n, x : xs, y : ys);
4      else
5         r=left_node(n);   s=right_node(n);
6         t=temparray + 2*n;   // locate empty space in temparray
7         for (ri=0;ri<r;ri++)
8            fft(s, x + xs*ri : xs*r, t + ri*s : 1);
9         for (i=0;i<n;i++)
10            t[i] = t[i] * w_n^((i div s)*(i mod s));
11         for (si=0;si<s;si++)
12            fft(r, y + ys*si : ys*s, t + si : s);
13  }
```

Figure 9: Pseudo-code for the FFT-TS algorithm

An efficient implementation of this algorithm will only use the temporary storage exactly needed, no more, no less. For example, when the breakdown tree is the right-most tree, the algorithm will not use any additional temporary storage. In the future, we will consider only such efficient implementations. Thus, for right-most trees, this algorithm will be equivalent to the FFT-RT algorithm of Subsection 3.3.2.

Summary

In this section we derived three major types of algorithms for the FFT with completely different characteristics: FFT-BR; FFT-RT; FFT-TS. It is possible to write many other algorithms, but they will share the characteristics of the three algorithms we presented.

The bit-reversed algorithm (FFT-BR) is done in-place, requiring no additional temporary storage. It produces output in bit-reversed order (see Section 2.2 for the definition of

bit-reversal permutation), or it consumes an input in bit-reversed order. This may not be a problem. If we need an algorithm that computes an actual FFT, we will have to complete explicitly the bit-reversal. This is a time-consuming process on a general-purpose hardware platform that does not support bit-reversed memory addressing.

The right-most tree algorithm (FFT-RT) cannot be done in-place. It requires the input and the output to have different memory addresses. It is limited to right-most trees only (see Chapter 2 for the definition). This algorithm has the major advantage of not requiring temporary storage.

The in-place algorithm with extra temporary storage (FFT-TS) can be done in-place, and supports arbitrary breakdown trees. Its disadvantage is that it requires a temporary storage of size $2 \cdot n$.

3.4 Algorithm Realization - Optimizing Codes

In the framework of SPIRAL, [1], the generation of optimized DFT codes will be done automatically, by a special purpose compiler. Such a compiler does not exist at the present time. To be able to perform meaningful experiments, we wrote highly optimized DFT codes by hand. We describe them in this section.

The code of each algorithm consists of six subroutines. The first is an interface subroutine `fft(n,x,y)`. This interface subroutine is called by the application when it needs to compute a DFT. Its parameters are the size of the DFT and the starting locations for the input and the output signals (arrays). It uses the standard C calling convention, i.e., passed parameters are pushed on the program stack by the caller in reverse order, and popped from it when the control returns to it.

The next subroutine `fft_rec(n,x:xs,y:ys)` is recursive, i.e., it calls itself. This subroutine is called by the interface subroutine. Parameters to this subroutine are passed via registers to reduce the number of accesses to the stack. Thus, it utilizes non-standard calling convention via registers. We choose a non-standard calling convention in order to speed-up

small size DFT's. When we are computing large size DFT's by the Cooley-Tukey breakdown method, we go recursively down to small sizes and compute them many times. Even incremental improvements for small sizes have a major impact on the overall performance. These results are derived in a more systematic way in Chapter 5. The initial conditions (leaves in the breakdown tree) for our recursive algorithms have to be DFT's computed from the definition, and are called **small-code-modules**. As discussed in Chapter 5, only a limited set of different sizes need to be supported for initial conditions. We implement small-code-modules for sizes $n = (2, 4, 8, 16)$. Thus, we have four additional subroutines: `dft_2(x,y)`, `dft_4(x,y)`, `dft_8(x,y)`, `dft_16(x,y)`. For the reasons discussed above, these subroutines pass parameters via registers as well.

The above-mentioned six subroutines are written in assembly language. Below we discuss the reasons for choosing this programming language.

3.4.1 Why Assembly Language?

We choose to program in assembly for the Intel Pentium architecture. The primary reason for choosing an assembly language over other programming languages is our ability to gain a deep insight on the DFT computation, so that we can create good analytical predictive performance models by modeling the execution of the DFT implementation. Our performance models will need parameters that characterize the code, so that they can be fine-tuned to it. Examples of such feedback parameters are the number of floating point instructions performed, the number of floating-point loads and stores, and the number of temporary variables used.

The second reason for choosing assembly language, as a programming language, is that we avoid many quirks associated with using a general-purpose compiler, while getting more freedom in carrying optimizations.

Complex Data Types vs. Real Data Types in High-Level Languages We implemented three types of algorithms described in Section 3.3 in C++ and Fortran programming languages to see how well compilers for these high-level programming languages are able to optimize the computation.

Since we are computing the DFT over the field of complex numbers, the very first step is to define a complex data type and common arithmetic operations for it. In C++ we wrote a generic complex class. Fortran supports complex numbers natively. We hoped that the compilers were smart enough to replace complex additions and multiplications by a combination of the corresponding real operations before performing optimizations. In reality, the compilers decided not to inline functions that implement complex operations. They decided to do a call to complex arithmetic functions, using standard-calling conventions, i.e., pushing arguments on the stack, and calling a subroutine to perform the operation. As a result, code written using complex data types turns out to be much slower than the same code manually rewritten with all variables and arithmetic operations being real valued.

The lesson learned is that we should not rely on general-purpose compilers to perform optimizations even if they are thought to be trivial. This also illustrates that with high-level general-purpose languages it is almost impossible to write a very abstract and easy to understand code while achieving a good runtime performance at the same time.

3.4.2 Small-Code-Modules: Highly-Optimized Code for Small DFTs

Optimization Goals Small-code-modules are subroutines that compute the DFT and are written using straight-line code, i.e., without any loops or subroutine calls. Input parameters for small-code-modules are pointers to input and output arrays (memory addresses for first elements) and strides for accessing elements from these arrays. For the algorithms presented in the previous section, we need two types of small-code-modules. Small-code-modules of the first type are in-place, i.e., they store the results of the computation in-place of its input data. As input parameters they only need to take one pointer and one stride.

Small-code-modules of the second type are out-of-place. The output stride is always 1. Support for both input and output strides increases the total number of instructions needed, because for access with variable stride extra instructions are needed to compute an index into arrays. Constant strides do not require additional overhead. Having this in mind, to create the best possible small-code-modules, rather than creating a single generic small-code-module that meets both requirements, we create two different small-code-modules, one for each type.

Having in mind the architecture of modern computers, we prioritize our optimization goals as follows:

1. Minimize the total number of temporary variables
2. Reduce the data cache misses by changing the order in which the elements are accessed
3. Minimize the total number of instructions and the total number of loads/stores

This choice of goals is based on the series of experiments we performed. The experiments show that the penalty for having extra temporary variables is very large. Hence, our first optimization is to minimize the total number of temporary variables.

The next optimization is to change the order in which elements are accessed. This optimization does not have any effect on our ability to perform other optimizations that is why we do it next.

By minimizing the total number of instructions and putting them in good order we achieve a better performance too. Minimizing the number of loads/stores reduces the code size, as op-codes become shorter. However, the trade-off leads to an increased number of instructions. Our experiments show that it is worth reducing the number of loads/stores even if the total number of instructions is increased.

Below we apply this optimization to the butterfly computation. Before that, in order to present the reader with the necessary background information, we briefly explain the floating-point unit of the Intel Pentium processor.

Intel Pentium Floating-Point Unit (FPU) Floating point values on the Intel's Pentium architecture can be either 32-bits (single real), 64-bits (double real), or 80-bits (extended real) wide, [9].

The FPU data registers consist of 8 80-bit registers. When real values are loaded from memory into the FPU data registers, the values are automatically converted into the 80-bit format. When computation results are subsequently transferred back into memory, the results can be left in the 80-bit format or converted back into the 64-bit or 32-bit formats.

The FPU instructions treat the 8 FPU data registers as a register stack. All addressing of the data registers is relative to the register on the top of the stack. For the FPU, a load operation is equivalent to a push and a store operation is equivalent to a pop. If we want to access the i-th element from the top of the stack, we write ST(i). When the FPU runs out of empty registers, a trappable exception occurs.

The FPU arithmetic instructions can only get their data from the FPU registers, thus explicit loads and stores are necessary. Instructions of interest can be grouped into two

Instruction	Args.	Description
FLD	mem	Load and push value on top of the stack
FSTP	mem	Store and pop value from top of the stack
FST	mem	Store top of the stack without poping
FXCH	ST(0),ST(i)	exchange values of registers
FADD/FADDP	ST(0),ST(i)	add real (/and pop)
FSUB/FSUBP	ST(0),ST(i)	subtract real (/and pop)
FSUBR/FSUBRP	ST(0),ST(i)	reverse subtract real (/and pop)
FMUL/FMULP	ST(0),ST(i)	multiply real (/and pop)
FCHS		Change sign

Table 1: Data transfer and arithmetic instructions for the Intel Pentium's FPU

categories: data transfer instructions and arithmetic instructions, as shown in Table 1. All arithmetic instructions operate on at most two FPU registers, one of which has to be the top of the stack ST(0). The first register plays the role of both the first operand of an instruction, and the destination for the result. For example, the command FSUB ST(0),ST(i) in Table 1 represents the subtraction ST(0)=ST(0)−ST(i).

Now we proceed with optimizing the butterfly computation for the Intel Pentium II floating-point unit.

Butterfly Computation When computing the DFT, after performing all possible arithmetic optimizations, each value produced at some stage is consumed at most twice, at consecutive stages. Most of the time, the process of consuming values occurs in pairs. For example, we take a pair of values, and compute their sum and their difference. When we do this, we consume each value in our pair twice. Since they are not used again, they can be discarded. The process of computing the sum and the difference of the two numbers, and discarding them afterwards, is called an **in-place butterfly computation**.

An in-place butterfly computation is the most common operation performed in our small-code-modules. That is why we want to find the best possible way of implementing it. A conventional way requires duplicating one of the values on the FPU stack (see previous paragraph for the background information on the syntax of the Intel's assembly instructions and the floating-point unit), and then computing the sum and the difference. Since the values that we just computed are reused again for the same type of computation, we cannot free the wasted register, unless we do a few extra operations. Our proposed method along with two conventional methods is displayed in Table 2.

The code in the left column of Table 2 wasted two FPU registers out of the eight available in the Pentium architecture. It is very hard to recover wasted registers because they are organized as a stack, and only the top of the stack can be accessed. Either we need many additional instructions (fxch and ffree) for freeing these registers, or, what is more likely to

Original:	Optimized:	Proposed:
fld A	fld A	fld A
fsub B	fsub B	fld B
fld A	fld A	fsub ST(1),ST(0)
fadd B	fadd B	fadd ST(0),ST(0)
		fadd ST(0),ST(1)
fld ST(1)	fld ST(1)	fsub ST(1),ST(0)
fsub ST(0),ST(1)	fsub ST(0),ST(1)	fadd ST(0),ST(0)
fld ST(2)	fxch ST(2)	fadd ST(0),ST(1)
fadd ST(0),ST(2)	faddp ST(1),ST(0)	
wasted 2 reg.	requires 1 free reg.	the best

Table 2: Examples of in-place butterfly computations

happen in the case of compilers, additional temporary storage will be needed to complete the program. The second column in Table 2 shows a somewhat optimized implementation, which again performs the same computation. The disadvantage of this method is that it requires the temporary use of a third register. When all eight registers are in use, this is not possible without additional temporary storage. Our proposed method, displayed in the right column of Table 2, does not require any extra registers, and does not waste any registers either. Another advantage of our method is that it requires less memory address computation at variable strides (computing addresses for A and B). Also it requires less loads from memory, and, as mentioned in the optimization goals, is faster.

Temporary Storage and Stack Alignment Issues As a recap, on the Intel's Pentium family of processors, double-precision data is 64-bits wide, while the integer data is 32-bits wide. This implies that the stack will be 32-bit aligned, but not necessarily, 64-bit aligned.

small-code-module	F_2	F_4	F_8	F_{16}
Out-of-place	0	0	0	10
In-place	0	0	8	26

Table 3: Amount of temporary storage required (64-bit floating-point values)

32-bit alignment means that in hexadecimal the least significant digit of a memory address should be either 0 or 8, while 64-bit alignment requires it to be 0. While the processor will work with 64-bit floating-point loads and stores for memory addresses that are 32-bit aligned, to achieve good performance, memory addresses should be 64-bit aligned. Thus, when allocating temporary storage on the stack, special attention is to be paid to aligning the stack to 64-bit addresses. This can be achieved by storing a 32-bit integer on the stack, if it is not already 64-bit aligned.

In Table 3 we summarize the total amount of temporary storage required to complete the computation of the DFT small-code-modules. This information will be useful for comprehending the results we obtain in the next paragraph.

Test Results We implemented small-code-modules for sizes $n = 2, 4, 8, 16$, and tested their performance at different strides. The results are presented in Figure 10. This figure shows the performance of our small-code-modules. For comparison, Figure 11 shows the performance of the codelets from the FFTW package, [10]. We compare against the FFTW package since this package is acknowledged to be one of the best FFT implementations available.

We tested our small-code-modules and the FFTW codelets at 2-power strides. These are the strides S at which these modules will be used when computing 2-power FFT's. The horizontal axis of both figures shows $\log_2$(stride). The vertical axis is the runtime performance, properly scaled, which allows comparing performance of code modules for

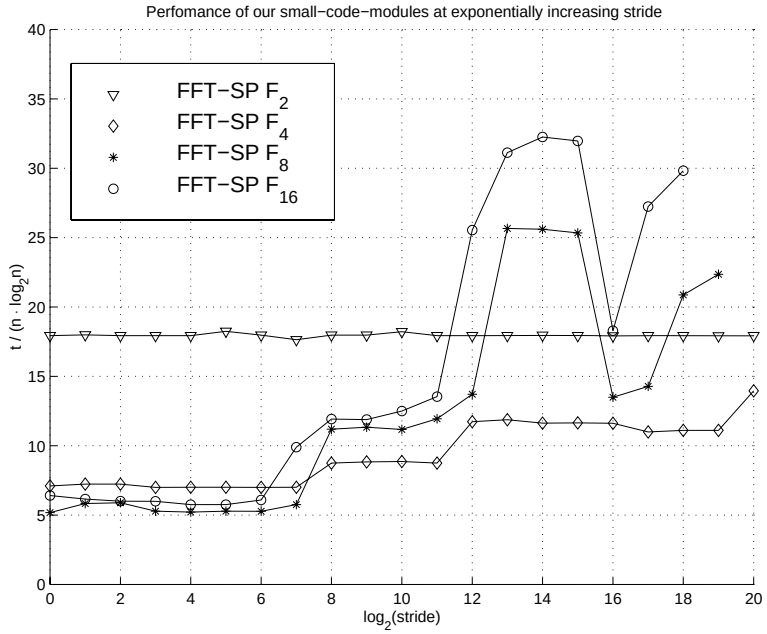


Figure 10: Performance of our small-code-modules at exponentially increasing stride

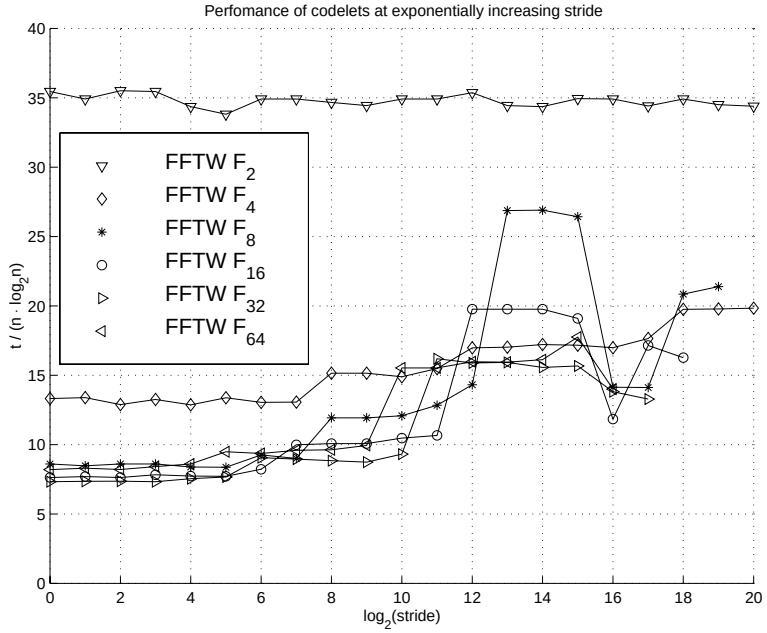


Figure 11: Performance of the FFTW codelets at exponentially increasing stride

different size FFT's. This scale, as shown in Chapter 5, is $\frac{t}{n \cdot \log_2(n)}$. On this scale, best is lower.

From Figure 10, our best small-code-module for small strides up to a stride of 2^7 is the 8-point DFT small-code-module. For large strides, starting from a stride of 2^8, the best small-code-module becomes the 4-point DFT. This can be explained by the fact that the 4-point DFT small-code-module does not require any temporary storage and can perform all computations using 8 FPU registers (the 4-point FFT has 8 real values, and requires only 6 of them to be on the FPU stack in order to do the computation in-place).

From Figure 11, the best FFTW codelet at almost every stride is the 32-point DFT. At strides, where it is not the best, it still approaches the codelets with an optimal performance.

We compare the performances of the best code modules from Figures 10 and 11 on the full stride range. Our 8-point DFT small-code-module for strides 2^0-2^7 combined with our 4-point DFT small-code-module for strides 2^8 and up lie below all sizes of the FFTW codelets at any stride. Thus, our small-code-modules, when used in the optimal implementation, will perform better than the FFTW codelets.

3.4.3 Optimizing the Twiddle Factor Computation and Data Access

Twiddle factors are the elements of the diagonal twiddle matrix T^n_r, which was defined in Section 2.4. Multiplication by the twiddle factors is done at each step of the recursion. For example, in the pseudo-code from Figure 8, the multiplication by twiddle factors is done on lines 8-9.

It is important to realize that for the 2-power FFT the total number of operations is of the same order as the number of operations required to do twiddle factor multiplications, i.e., it is of order $n \cdot \log_2 n$.

Lemma 2 *For the 2-power FFT the number of twiddle factor multiplications is* $O(n \cdot \log_2 n)$.

Proof: Suppose we are computing the n-point FFT. Consider a step of the recursion where

we are computing the n'-point FFT for some $n'|n$. This implies that we do n' multiplications inside of each recursive call of size n'. But there will be $\frac{n}{n'}$ number of calls of this size. Thus, the total number of multiplications done inside of each recursive call of size n' will be n. But there are a total of $1 + \log_2 n$ divisors of n. As a result, the total number of multiplications will be $O(n \cdot \log_2 n)$, which proves the statement. □

The experiments show that the time spent on accessing the twiddle factors and multiplying the data by them takes about 30-40% of the total computation time. For this reason it is very important to optimize their access and computation.

The twiddle factors require the computation of certain values $\cos(\alpha)$ and $\sin(\alpha)$ for obtaining a single complex value. Trigonometric computations are very expensive in terms of performance operations. To achieve good performance, we pre-compute the values of the twiddle factors, and store them in a twiddle factor array. There are many ways of storing these pre-computed values. In this section we focus on identifying the best one.

Figure 12 compares the effect of different ways of storing the twiddle factors on the performance of various size FFT's. On the horizontal axis of the plot we have $\log_2 n$, where n is the size of the FFT. The vertical axis is the relative runtime, which we obtain by dividing the runtimes of FFT's with different strategies by the runtime of the FFT with our preferred strategy.

The plot marked by $\circ$ needs an array of size $2n$ to store twiddle factors when computing an FFT of size n. We store the roots of unity for each $n' \leq n$, starting at the location $2n'$ and storing them in their natural order, i.e., in the position $2n' + i$ we store $(w_{n'})^i$. This is a generic way of storing twiddle factors, as it does not depend on the particular breakdown tree used. We use this storage mechanism in the best version of our code.

In the second approach, for the plot marked by $\triangle$, we presort the roots of unity by the order in which they are located on the diagonals of the twiddle matrices T^n_r. This way of storing the twiddle factors depends on the breakdown tree used. Because it gives only a

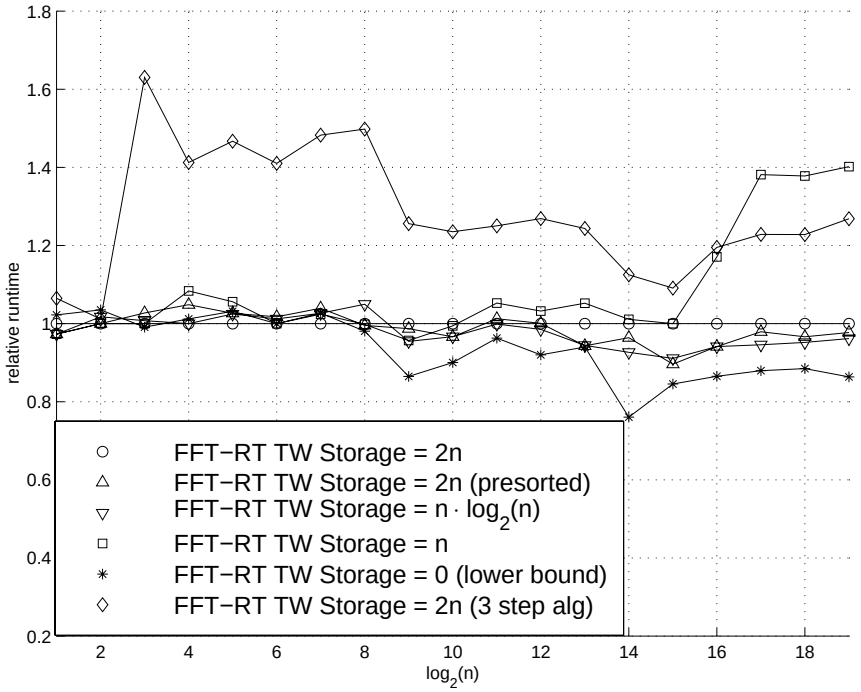


Figure 12: Optimizing the twiddle factor computation and the access performance

small improvement over the preferred strategy, we decided not to use this method.

The third plot, marked with $\bigtriangledown$, is the more natural way of storing twiddle factors. We have a two-dimensional array of size $k \times n$, where $k = \log_2 n$, and on each row of this array we store the roots of unity, corresponding to each value of n in the natural order. This storage mechanism allocates space in memory for $k \cdot n$ storage elements, although only $2n$ elements are actually used. This way of storing twiddle factors gives about 5% improvement for large sizes over the one we chose marked with $\circ$. We decided against this method because it requires $n \cdot \log_2(n)$ memory.

The next method, plotted with $\square$, corresponds to the most compact way of storing the twiddle factors. It stores the roots of unity in the natural order, as explained above, for

the largest size of FFT we compute, and uses them for smaller sizes, by accessing them at stride. Experimentally, we observed that this mechanism of storing twiddle factors is 40% slower than the method we chose, because of the access patterns in strides.

The plot marked with $*$ shows the theoretical minimum for reducing the cost of accessing twiddle factors. This plot does not correspond to computing a correct FFT. Rather it corresponds to multiplying every time by the same value of the twiddle factor. We display this plot to show what the theoretical minimum is, as no twiddle factor storage method can do better than this one. Our twiddle factor storage method is within 10-20% of this theoretical minimum.

The last plot on the figure, marked with $\diamond$, shows a still different access pattern to the twiddle factors during the computation of the FFT. It performs a separate data path through the data and twiddle factors to multiply them. This is the access pattern that we used in our original algorithms, e.g., the FFT-RT-3 algorithm in Figure 8. The plot clearly shows the inferiority of the extra data path. Having this in mind, we rewrote the algorithms to avoid this extra data pass. We moved the multiplication by the twiddle factors to be inside one of the loops that calls the FFT down recursively.

For right-most trees, it is more efficient to move the multiplication by the twiddle factors inside the second loop, the one that calls the leaf nodes, as now we can move multiplication by the twiddle factors inside of the left small-code-modules, unroll the multiplication, and re-optimize them. Below we summarize this algorithm for right-most trees.

Algorithm 5 (FFT-RT) *Out-of-place algorithm for right-most trees with twidddle factor multiplication done inside of left small-code-module*

1. $y = (I_r \otimes F_s) \cdot (L^n_r \cdot x)$
2. $(L^n_s \cdot y) = \left((I_s \otimes F_r) \cdot T^n_r\right) \cdot (L^n_s \cdot y)$

Figure 13 shows the pseudo-code for this algorithm. Multiplication by the twiddle

Pseudo-code 5 (FFT-RT) *Recursive implementation*

```
 1   fft(n, x : xs, y : ys) {
 2      if is_leaf_node(n)
 3         dft_small_module(n, x : xs, y : ys);
 4      else
 5         r=left_node(n);   s=right_node(n);
 6         for (ri=0;ri<r;ri++)
 7            fft(s, x + xs*ri : xs*r, y + ys*ri*s : ys*1);
 8         for (si=0;si<s;si++)
 9            if (si<>0)
10               for (ri=1;ri<r;ri++)
11                  y[ys*i] = y[ys*i] * w_n^(ri*si);
12            fft(r, y + ys*si : ys*s, y + ys*si : ys*s);
13   }
```

Figure 13: Pseudo-code for the FFT-RT algorithm

factors is performed on line 11. In pseudo-code notation, it is presented as multiplication by `w_n^(ri*si)`. In reality, however, we do not perform this computationally heavy operation here. Taking the primitive root of unity to some power corresponds to locating this root of unity in a table collecting roots of unity up to a certain degree. Also, we do not perform the multiplication `(ri*si)`. Rather, we note that multiplication is equivalent to accessing elements at stride `si`, when the inner loop is performed on `ri`. Also, since left nodes are leaves for right-most trees, we can bring the loop that performs the multiplication by the twiddle factors inside of the small-code-modules, then unroll the loop and carry out further optimizations. Noticeable speed-up is also achieved, if, just before the multiplication by the twiddle factors, we check for `(si==0)` condition. If the condition is satisfied, the twiddle factor is a unit, so no multiplication is required. The speed-up is achieved due to the fact that the multiplication of two complex numbers is much slower than the condition checking.

Summary

In this section we exploited different possibilities of deriving optimized codes for the algorithms from Section 3.3.

Our codes were written in assembly language for the Intel Pentium architecture in order to gain a deep insight on the DFT computation and to have freedom in carrying different optimizations. In Chapter 5 we will use this insight to come up with accurate analytical predictive performance models.

The most commonly used operation during the computation of small-code-modules is in-place butterfly computation. We found an efficient way of computing it on the Intel Pentium platform.

The computation effort required for the multiplication by twiddle factors is of the same order as the total computation effort for the 2-power FFT. For this reason, we experimentally tried many algorithmic optimizations, and found an efficient and generic way of storing twiddle factors and performing multiplications by them.

3.5 Conclusions

We derived three major types of algorithms for the FFT with completely different characteristics: FFT-BR; FFT-RT; FFT-TS. It is possible to write many other algorithms, but they will share the characteristics of the three algorithms we presented.

The bit-reversed algorithm (FFT-BR) is done in-place, requiring no additional temporary storage. It produces output in bit-reversed order (see Section 2.2 for the definition of a bit-reversal permutation), or it consumes an input in bit-reversed order. This may not be a problem. If we need an algorithm that computes an actual FFT, we will have to complete explicitly the bit-reversal. This is a time-consuming process on a general-purpose hardware that does not support bit-reversed memory addressing.

The right-most tree algorithm (FFT-RT) cannot be done in-place. It requires the input

and the output to have different memory addresses. It is limited to right-most trees only (see Chapter 2 for the definition). This algorithm has the major advantage of not requiring temporary storage.

The in-place algorithm with extra temporary storage (FFT-TS) can be done in-place, and supports arbitrary breakdown trees. Its disadvantage is that it requires a temporary storage of size $2 \cdot n$.

When optimizing small-code-modules, the total number of temporary variables, instructions, and loads/stores need to be minimized. Memory load and store instructions need to be reordered in a way that reduces cache misses due to collisions, when small-code-modules are called at large strides, aligned to the cache size.

In-place butterfly computation is the most commonly done operation. An efficient algorithm for implementing it is necessary. In addition, special attention needs to be paid to memory alignment to achieve better performance.

Multiplication by twiddle factors is a significant fraction of the total computation time, so twiddle factors need to be pre-computed and stored in an order that reduces cache misses. Reducing the storage size for twiddle factors does not necessarily lead to an improvement, as we saw in Figure 12.

These optimizations combined together with the algorithms in Section 3.3 helped us to derive optimized codes for the Intel Pentium architecture. The codes we have described support different breakdown trees. Now, in order to find an efficient way of computing a 2-power FFT of a given size, we will try all possible combinations of codes and breakdown trees, and find the most efficient pair. In order to find such a pair, we need to come up with tools for estimating the performance, and derive efficient search methods for the optimum. Chapter 4 addresses experimental performance evaluation (benchmarking), while Chapter 5 addresses analytical performance evaluation (performance prediction). Chapter 6 presents different search methods.

4 Experimental Measurement of Performance

4.1 Introduction

Motivation In many applications, it is necessary to get an estimate for the runtime for a given function. The problem of obtaining such an estimate is sometimes under-simplified. Not much effort is spent on choosing a proper strategy and on fine-tuning the parameters for it, in order to get timing measurements with a desired accuracy. In this chapter, we develop a strategy that does not require any a priori knowledge about the function that is to be timed.

Goal The goal of this chapter is to develop a benchmark tool for evaluating the performance of different implementations by obtaining reproducible non-biased estimates of the runtime with a desired accuracy in minimal possible time.

Methodology A benchmark tool requires an experimental procedure to obtain the reproducible estimates of the runtime. We divide this procedure into two steps. The first step is to estimate N – how many times the subroutine under experiment need to be executed in order to achieve a given quantization error. The value of N depends on the choice of the clock source. On the second step, we compute a timing estimate T from the N executions of the function, repeat the whole procedure M times, and then compute from these T_m values a final runtime estimate.

Description The chapter addresses the following issues:

1. Choosing the clock source
2. Coming up with the value of N – how many times to repeat
3. Deciding on the processing method

4.2 Characterizing Clock Sources

Depending upon the particular platform and on the operating system support and features, we may have access to more than one clock source. All clocks are granular in nature, and they can be characterized by their resolution Δ (i.e., how often the clock is updated) and accuracy (i.e., how accurately this update is happening).

In this report, we are going to consider three clocks:

```
SSC - standard system clock (<time.h>)
USC - unix system clock (<sys/time.h>)
TSC - pentium time stamp counter (rdtsc instruction)
```

In order to estimate the clock resolution we store the current value of the clock, and wait until it changes. The difference between ending and starting values gives such an estimate. Experiments show that with low probability we may get values for the resolution that are higher than the actual value. This can be explained by the fact that if the operating system is busy processing high priority tasks, it might delay the processing of events of less priority, such as timer events. This is taken care of by repeating the experiment, taking the difference several times, and then keeping the minimal value.

In Table 4 we present the resolution for different clocks. We see that the standard system clock (SSC) gets updated at very low rates of about 50-150 Hz. On the contrary, by accessing the Pentium time stamp counter (TSC), we can measure very accurately the number of elapsed clock cycles, as this counter gets updated at the processor internal clock rate. Since we have to spend some time accessing the counter and processing its value, the effective resolution will be about 150 (it takes these many clock cycles to read the counter and process the value) times slower than the clock rate, which corresponds to 3 Mhz on a 450 Mhz Pentium machine.

Processor	System	SSC Δ (sec.)	USC Δ (sec.)	TSC Δ (sec.)
Pentium II	MS Win32	1.00e-2	N/A	2.82e-7
Pentium II	Linux	5.90e-3	N/A	2.82e-7
Sparc	SUN OS	1.00e-2	1.00e-2	N/A
Alpha	DEC OS	1.67e-2	9.76e-4	N/A

Table 4: Clock resolution for different clocks

4.3 Quantization Errors

The most naive way of measuring the runtime for a function is to store the starting clock value, execute the function, and subtract the stored starting value from the final clock value. This approach returns 0 for functions that are smaller (in terms of the runtime) than the clock resolution Δ. This problem is taken care of by executing the function N times before reading the final value of the clock. Since we do not know a priori what the function runtime is, we cannot estimate how many times the function should be executed before it is safe to obtain the final clock reading. To find the value of N we run series of experiments for the given function. The most common solution is to start with $N = 1$, and keep multiplying N by $N_0 = 2$ or some other factor until the desired accuracy is achieved.

Our approach is similar, but allows us to achieve a desired accuracy in much less time. This is done by calculating the maximal safest value N_0 by which N should be multiplied in the loop so that the resulting value of N is not much higher than the smallest value needed to obtain the desired accuracy of the measurement $\varepsilon_{\text{quant}}$ (relative quantization error) for the given function.

The algorithm is as follows. Suppose we are given a function for which we are estimating the runtime, and the desired accuracy $\varepsilon_{\text{quant}}$ of the measurement. Then in order for quantization errors to be smaller than $\varepsilon_{\text{quant}}$, we need to run the experiment for at least $t_{\text{threshold}} = \frac{\Delta}{\varepsilon_{\text{quant}}} + \Delta$ seconds (e.g., to achieve 20% accuracy we need to run the experiment

for at least 6Δ seconds). We start with the value of $N = 1$. The function is executed N times before reading the final value t of the clock. Then we test if the value of t is zero. If $t = 0$, then N is multiplied by $N_0 = \left\lceil \frac{1.1}{\varepsilon_{\text{quant}}} \right\rceil$. If the value of $t \neq 0$, then we can set the new value of N to be $N = \left\lceil 1.1 \cdot N \cdot \frac{t_{\text{threshold}}}{t} \right\rceil$. The process is repeated until $t > t_{\text{threshold}}$.

Summary In this section we presented an algorithm that enables us to quickly obtain an estimate of N – how many times the function under measurement needs to be repeated in a loop, given an upper bound for the error of the runtime measurements. This algorithm takes into account only quantization errors, i.e., errors which are due to the discrete nature of clocks.

4.4 Other Sources of Errors

In the previous section, we showed how to perform runtime measurements with small quantization errors. There are additional sources of errors including the following: the processor is used concurrently by the operating system and other processes while the experiment is running; the state of the computer is constantly changed (pipeline, branch predictor, memory hierarchy), etc.

Our goal is to specify an upper bound on the error, and get runtime estimates with errors that on average are smaller than this given upper bound.

Once the optimal value of N is determined using the procedure in the previous section, the process of obtaining a single sample is to start the timer, execute the function N times, and then stop the timer. The difference of the starting time and the ending time divided by N is the estimate of the runtime.

To minimize the effect of other sources of errors, we repeat these experiments several times, and then process these multiple measurements to obtain a final estimate. We could either take the mean value, or the minimum value. Reasons that support choosing the minimum value rather than the mean are to reduce the overhead due to the processor

being used concurrently by the operating system and other processes. Other less common methods are to choose the i-th minimal sample, or to find the histogram and to take the mean of the values in the highest bin.

We carried out experiments that show that, by taking a minimum, we reduce the variance, while getting biased estimates. Taking the mean of the values in the highest bin reduces the bias, but does not reduce the variance significantly. Since it is more important for us to get unbiased estimates, we decided to use the following processing method. We remove 10% of the samples, those that are far from the majority of the samples, and take the mean value of the remaining 90% of the samples.

4.5 Conclusions

We devised a strategy for obtaining reproducible runtime estimates with desired accuracy with a small overhead effort. Based on this strategy, we developed a benchmark tool that takes as its inputs the subroutine to be timed and a desired level of accuracy for the measurement, and that produces a measurement of the runtime. This benchmark tool will be used to compare the performance of different implementations of the FFT algorithm.

5 Analytical Modeling of Performance

5.1 Introduction

Motivation Finding experimentally the optimal implementation by exhaustive search over the set of all implementations (see Section 2.6) is not feasible for reasonable values of the data size due to the extremely large number of possible alternatives and high experiment execution time necessary to produce accurate estimates (see Chapter 4).

Goal The goal of this chapter is to develop analytical models that can predict the performance of any implementation much faster than running the actual experiment.

Methodology The basis for our cost models is the recurrence Equation (4) that reflects the Cooley-Tukey formula structure, but contains costs of computing 2-power point DFTs as its terms instead of sub-transforms. At first, we derive the leaf-based cost model. It enables us to cluster the set of all breakdown trees into smaller sub-sets and also sets the framework for comparing the small code modules. Then we present an advanced cost model. The advanced model takes into account the overhead of accessing data in real computational environments.

5.2 Leaf-Based Cost Model for the FFT

We consider the data size $n = 2^k$, and factor $2^k = 2^q \cdot 2^{k-q}$. In this case, the Cooley-Tukey formula, as presented in Section 2.4 of Chapter 2, is given by

$$F_{2^k} = (F_{2^q} \otimes I_{2^{k-q}}) \cdot T^{2^k}_{2^{k-q}} \cdot (I_{2^q} \otimes F_{2^{k-q}}) \cdot L^{2^k}_{2^q}.$$

This formula computes 2^q–point DFT 2^{k-q} times, and 2^{k-q}–point DFT 2^q times. Let $T_L(k)$ be the cost of computing a 2^k-point DFT. If we disregard any cost associated with accessing data at strides and multiplication by twiddle factors, then we can write the

following recurrence for the cost

$$T_L(k) = 2^{k-q} \cdot T_L(q) + 2^q \cdot T_L(k-q). \tag{4}$$

First, we solve Equation (4) for a simple case, namely, when all leaves are of the same size. Then, we will use these results to solve Equation (4) in the general scenario when the leaves are allowed to be of different sizes.

Case 1: Cost when all leaves are of the same size

Lemma 3 *Let the leaves be all 2^r-point DFT, assuming $r|k$. Then the solution to Equation (4) is*

$$T_L(k) = \frac{k}{r} \cdot 2^{k-r} \cdot T_L(r). \tag{5}$$

Proof: We use mathematical induction. The statement is true for $k' = r$, because $T_L(r) = \frac{r}{r} \cdot 2^0 \cdot T_L(r) = T_L(r)$. Now suppose the statement is true $\forall\ k' < k$. Based on this, we write recursively

$$\begin{aligned} T_L(k) &= 2^{k-q} \cdot T_L(q) + 2^q \cdot T_L(k-q) \\ &= 2^{k-q} \cdot \frac{q}{r} \cdot 2^{q-r} \cdot T_L(r) + 2^q \cdot \frac{k-q}{r} \cdot 2^{k-q-r} \cdot T_L(r) \\ &= 2^{k-r} \cdot \frac{q}{r} \cdot T_L(r) + 2^{k-r} \cdot \frac{k-q}{r} \cdot T_L(r) \\ &= \frac{k}{r} \cdot 2^{k-r} \cdot T_L(r). \end{aligned}$$

By induction we obtain the result. □

We note that from Equation (5) the cost model given by Equation (4) takes into account only leaves present in a tree, but fails to account for the structure of the tree.

Another conclusion that can be made from Equation (5) is that creating very well optimized implementations of the DFT for leaves of a breakdown tree is very important.

Equation (5) only defines a recurrent relationship for computing a 2^k-point DFT using DFTs of smaller 2^r-point data size. It says nothing about boundary conditions (i.e., if we really stopped the recurrence at r). Our goal is to choose one or more boundary conditions from all possible choices that will minimize the cost function $T_L(k)$. DFTs that are used as boundary conditions will have to be implemented without using the Cooley-Tukey rule. These DFTs will be called **small-code-modules**. We define a small-code-module to be the straight-line code that implements the DFT directly from the definition, with as many arithmetic optimizations as possible, carried out by a human and/or by a compiler [10].

Let $T_{\mathrm{cm}}(r)$ be the actual runtime performance of a 2^r-point DFT small-code-modules. We use these values as boundary conditions to Equation (5). Then we can rephrase the problem of minimization stated above: minimize $T_L(k)$ $\forall r \in Z_+ : r|k$ using Equation (5). This leads to the equation

$$\min_{r|k} T_L(k) = \frac{k}{r} \cdot 2^{k-r} \cdot T_{\mathrm{cm}}(r).$$

To solve this equation, consider

$$\min_{r>0} \frac{T_{\mathrm{cm}}(r)}{r \cdot 2^r}, \tag{6}$$

which no longer depends on k. Thus, the problem of finding the best breakdown tree is equivalent to identifying the best small-code-modules in the framework of the model given by Equation (4) and using them as leaves of the breakdown tree.

Having the definition of small-code-modules in mind, we should consider DFT small-code-modules only up to some fixed size. Large size small-code-modules do not perform well, because a straight-line code size increases exponentially with the problem size beyond a certain size, it no longer fits into the instruction cache of the processor, significantly decreasing the performance (see Chapter 2).

The minimization problem given by Equation (6) is interpreted graphically as finding the value of r at which the function $f(r) = \frac{T_{\mathrm{cm}}(r)}{r \cdot 2^r}$ achieves its absolute minimum.

Figure 14 gives, as an example, the results of measuring the runtime performance for small-code-module from three packages: FFTW [10], our package written in Pentium assembly code, and our earlier package written in Fortran. Although runtime for small-code-modules for larger data sizes is much larger than for smaller data sizes, the runtimes are scaled in such a way that they can be compared to each other, i.e., on the $\frac{t}{r \cdot 2^r}$ scale. In Figure 14, lower is better, i.e., the lower a plot is the better the corresponding performance. The middle plot is for small-code-modules from the FFTW package [10]. These small-code-modules are automatically generated in the C language by the FFTW code generator. According to our model, the best FFTW small-code-module is for $r = 3$, e.g., for the 8-point DFT. The bottom one (FFT-SP is in assembly language) is for our small-code-modules, hand-written for the Intel Pentium assembly language. Again, the best one is for $r = 3$. The top plot is for the performance of Fortran coded small-code-modules, optimized for the number of arithmetic operations and based on the implementation proposed in [5]. This plot is much higher than the other two, which clearly shows that in order to get the best possible implementation it is not enough to reduce the number of arithmetic operations, and supports our motivation given in Chapter 1.

As it is evident from Figure 14, some small-code-modules give better performance than others. If our decision criteria is to mimimize Equation (5), then it is best to use at all times small-code-modules corresponding to the minimum in Figure 14.

Case 2: Cost with leaves of different sizes

The model presented above is unrealistic because it assumes that all leaves have to be of the same size, while this is not always possible, as k might not be divisible by the value of r that we choose.

To overcome this assumption, we define the range of good small-code-modules (e.g., $r = 2, 3, 4, 5$) and try different possibilities of representing a given k as a sum of these small-code-modules, and choose the one that minimizes the cost function $T_L(k)$. This will

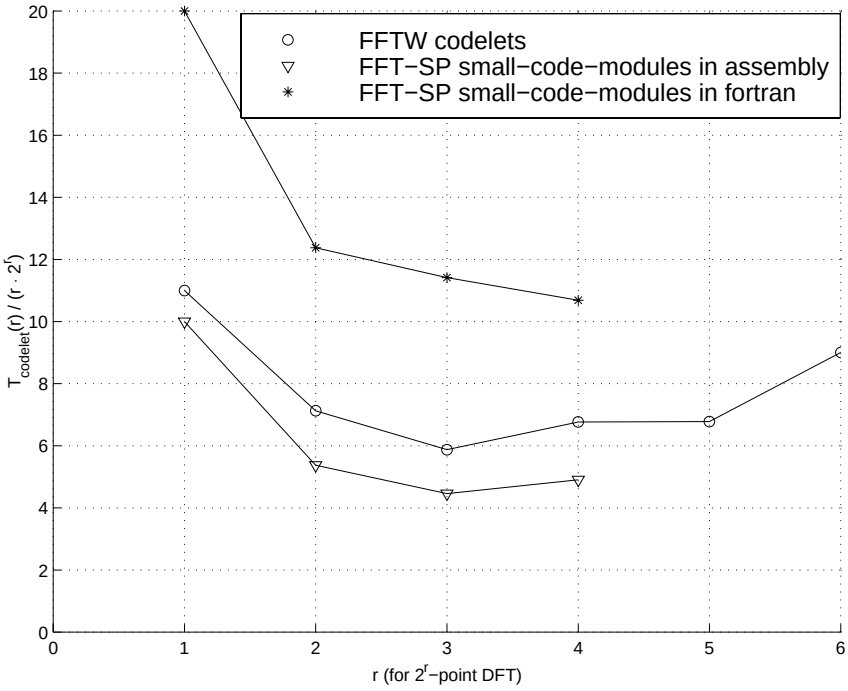


Figure 14: Comparing the Performance of Small-code-modules on $\frac{T_{\mathrm{cm}}(r)}{r \cdot 2^r}$ Scale

give us a computational based prediction model.

Lemma 4 *If* $k = \sum_{i=1}^{m} r_i$, *then* $T_L(k) = \sum_{i=1}^{m} 2^{k-r_i} \cdot T_L(r_i)$.

Proof: We prove by induction. For $m = 2$ the result follows from Equation (4). Assume the result is true for $m - 1$. Based on this assumption, we show that it is true for m.

$$\begin{aligned} T_L(k) &= 2^{k-r_m} \cdot T_L(r_m) + 2^{r_m} \cdot T_L(k - r_m) \\ &= 2^{k-r_m} \cdot T_L(r_m) + 2^{r_m} \cdot \sum_{i=1}^{m-1} \cdot 2^{k-r_m-r_i} \cdot T_L(r_i) \\ &= \sum_{i=1}^{m} 2^{k-r_i} \cdot T_L(r_i). \end{aligned}$$

This proves the lemma using mathematical induction. □

Result 1 *Let n_r be the number of leaves of size 2^r, $r = 1, \ldots, C$. Then*

$$T_L(k) = \sum_{r=1}^{C} n_r \cdot 2^{k-r} \cdot T_L(r), \qquad \textit{for } k = \sum_{r=1}^{C} r \cdot n_r. \tag{7}$$

Based on this result we can state the minimization problem as follows

$$\min_{k=\sum_{r=1}^{C} r \cdot n_r} \quad T_L(k) = \sum_{r=1}^{C} n_r \cdot 2^{k-r} \cdot T_{\mathrm{cm}}(r). \tag{8}$$

We optimize $T_L(k)$ over all allowed values of n_r. The results of evaluating such a model are explained below.

In Figure 15 we compare an actual runtime performance of three trees: one chosen as optimal by the model of Equation (8), one found to be optimal using the dynamic programming (DP) search method, introduced in Section 6.3, and one that gives the worst performance – radix-2 tree. The model of Equation (8) decides to choose the small-code-modules of size 2^3 most of the time (see Figure 16). In addition, it chooses few small-code-modules of size 2^2 whenever $3 \not| \; k$. On the contrary, an optimal tree found using search methods uses small-code-modules of size 2^2 most of the time (see Figure 18).

Figure 17 compares the estimate of the runtime given by the model and the actual runtime for the same breakdown tree. The actual runtime values are much higher as the problem size increases. This shows that there is overhead that is not taken into account. This overhead grows as the problem size increases.

Even with its limitations, the leaf-based model can still be used to compare two different trees and pick the best one. The major limitation of this model is that it treats trees with the same number of small-code-modules of a given size in its leaves to be equivalent, while experiments show that they are not.

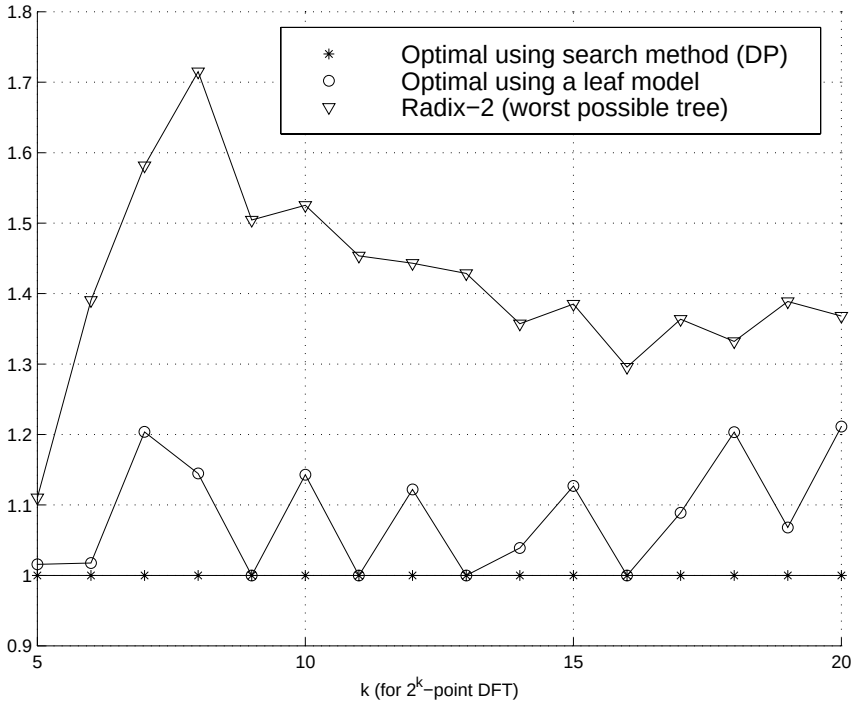


Figure 15: Comparing the actual performance of optimal trees

$r \backslash k$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	1	0	2	1	0	2	1	0	2	1	0	2	1	0	2	1
3	0	0	1	0	1	2	1	2	3	2	3	4	3	4	5	4	5	6	5	6
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 16: Values of n_k for the optimal tree found using the leaf model given by Equation (8)

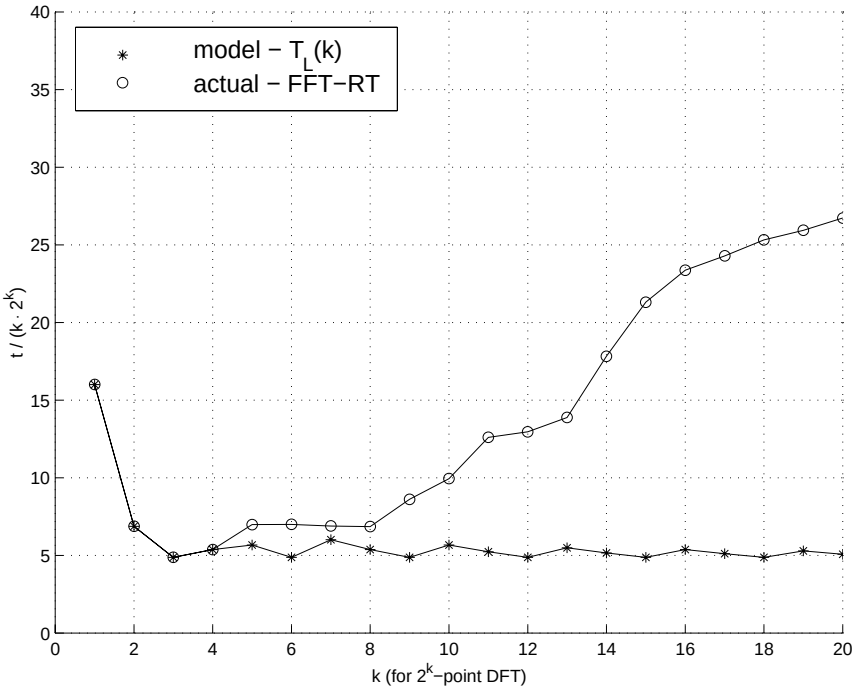


Figure 17: Comparing the performance for the actual and the estimated optimal tree

$r \backslash k$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1
2	0	1	0	0	1	1	0	0	1	1	2	2	3	3	4	4	5	5	6	6
3	0	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1
4	0	0	0	1	0	1	1	2	1	2	1	2	1	1	1	1	1	1	1	1

Figure 18: Values of n_k for the optimal tree found using search methods

We improve the performance prediction of the model in the next section by expanding it to take into account the overhead of accessing the data in memory.

5.3 Cache-Sensitive Cost Model for the FFT

Problem Formulation

We saw that the leaf-based cost models presented in Section 5.2 failed. They did not take into account overhead associated with data access. This is a reasonable assumption if the overhead is constant for any breakdown tree. In reality, this is not the case. So it is desirable to design a model that explicitly takes into account the overhead.

There are three types of overhead. The first is the overhead associated with recursive function execution. This overhead includes the costs of determining how to split at each step of the recursion, index calculations, etc. The second type of overhead relates to the multiplication by twiddle factors. Finally, the data access overhead is the overhead with accessing the input, the output and the twiddle factor arrays at variable strides.

Optimal decomposition trees are highly dependent on the code that implements the Cooley-Tukey formula. Thus, it is a challenge to create a model that predicts uniformly well regardless of the actual implementation of the Cooley-Tukey formula. We need to choose a good performance code, and create a custom-tailored performance model for it. Such a model will find the breakdown tree, which is guaranteed to be optimal only if it is used with this code. Based on extensive experiments, the best code has shown to be the one that implements the algorithm for right-most trees (FFT-RT), presented in Chapter 3. In this section we will create a cache-sensitive model tailored to this particular code. For reference the pseudo-code for the FFT-RT is given in Figure 19.

Our right-code-module does an out-of-place computation of the DFT (on line 3). Our left-code-module first does an in-place multiplication by twiddle factors, and then does an in-place computation of the DFT (on line 9). Our recursive subroutine does not access any

Pseudo-code 6 (FFT-RT) *Recursive implementation*

```
 1   fft(n, x : xs, y : 1) {
 2      if is_leaf_node(n)
 3         dft_right(n, x : xs, y : 1);
 4      else
 5         r=left_node(n);   s=right_node(n);
 6         for (ri=0;ri<r;ri++)
 7            fft(s, x + xs*ri : xs*r, y + ri*s : 1);
 8         for (si=0;si<s;si++)
 9            dft_left(r, y + ys*si : ys*s, 2*n : si);
10   }
11   dft_right(n, x : xs, y : 1);
12      // compute out-of-place n-point DFT x:xs -> y:1
13   }
14   dft_left(n, x : xs, w : ws);
15      // twiddle factor multiplication x:xs * w:ws -> x:xs
16         if (ws<>0)            // w[0] is 1, so do not multiply
17            for (i=1;i<n;i++)  // also skip i=0, because w[0]=1
18               x[xs*i] = x[xs*i] * w[ws*i];
19      // now compute inplace n-point DFT x:xs -> x:xs
20   }
```

Figure 19: Pseudo-code for the FFT-RT algorithm

data. It is only responsible for scheduling an execution of the small-code-modules in proper order with the proper arguments.

Model Formulation

Suppose we have small-code-modules of sizes 2^c available, where $c = 1, 2, \ldots, C$, and let $\mathbf{r} = (r_0, r_1, \ldots, r_m)$, where $1 \leq r_i \leq C$. We define $k_i = \sum_{l=0}^{i} r_l$, where $i = 0, 1, 2, \ldots, m$, and set $k = k_m$.

We create the right-most tree, completely defined by the vector $\mathbf{r}$, as shown in Figure 20.

Right-most trees have one right leaf r_0 and m left leaves. The right leaf small-code-module is out-of-place. It is accessing the input array $\mathbf{x}$ at some stride and writing into the

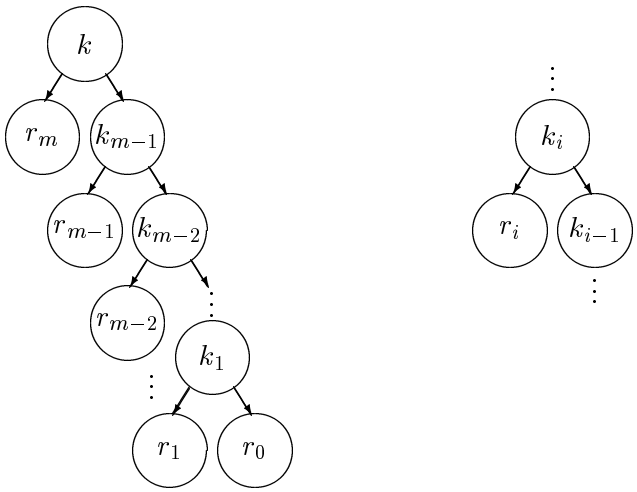


Figure 20: The right-most tree is completely defined by the vector $\mathbf{r} = (r_0, r_1, \ldots, r_m)$.

output array **y** at a stride 1. Left leaf small-code-modules are in-place. They access data in the output array y at a given stride, and store the results in the same array at the same stride. The performance of small-code-modules that correspond to left and right leaves can be modeled as a function of the stride, at which they are accessing the data.

Below we formulate important properties of right-most trees in the context of the pseudo-code of Figure 19.

Lemma 5 *Each small-code-module r_i, where $i \geq 0$, is called 2^{k-r_i} times, always at the same stride.*

Lemma 6 *The right small-code-module r_0 is accessing data at a stride 2^{k-r_0}.*

Lemma 7 *Each left small-code-module r_i, where $i > 0$, is accessing data at a stride of $2^{k_{i-1}}$.*

Lemma 8 *The recursive subroutine with input size being k_i is called 2^{k-k_i} times.*

Let $g_c^L(k)$ and $g_c^R(k)$ be cost functions for a left-code-module and a right-code-module, respectively, of sizes 2^c, accessing data at a stride 2^k. Let $f(k_i, r_i)$ be the cost of executing the recursive function of input size 2^{k_i}, which breaks as $k_i = r_i + k_{i-1}$, as shown in Figure 20. We define $T_C(\mathbf{r})$ to be the cost of computing a 2^k-point DFT.

Lemma 9 *The cost $T_C(\mathbf{r})$ of computing a 2^k-point DFT using a right-most tree and specified by $\mathbf{r}$ is*

$$T_C(\mathbf{r}) = 2^{k-r_0} \cdot g_{r_0}^R(k - r_0) + \sum_{i=1}^{m} 2^{k-r_i} \cdot g_{r_i}^L(k_{i-1}) + \sum_{i=1}^{m} 2^{k-k_i} \cdot f(k_i, r_i).$$

Proof: The lemma is easily proved by using the lemmas given above. □

Lemma 9 defines a cost function in terms of three cost functions $g_c^L(k)$, $g_c^R(k)$, and $f(k_i, r_i)$. The first two functions can be determined experimentally. The experiment calls each small-code-module at all different possible strides. Since there are C small-code-modules, and strides can only be powers of 2 up to 2^k, then the total number of experimental measurements needed is of the order of $2 \cdot C \cdot k$.

The function $f(k_i, r_i)$ does not access any data, so it is proportional to the total number of instructions executed for each particular instance of the recursive function given in Figure 19.

Lemma 10 $f(k_i, r_i) = a_1 \cdot 2^{r_i} + a_2 \cdot 2^{k_{i-1}} + a_3$.

Proof: We inspect the pseudo-code in Figure 19. The first loop is executed 2^{r_i} times, and the second loop is executed $2^{k_{i-1}}$ times. Let a_1 and a_2 be the costs of executing the bodies of the first and second loops respectively, and let a_3 be the cost of executing the code that is outside of the loops, not counting costs of calling any subroutines. Then the total cost is

$$a_1 \cdot 2^{r_i} + a_2 \cdot 2^{k_{i-1}} + a_3,$$

which proves the lemma. □

Theorem 11 *The cost $T_C(\mathbf{r})$ of computing a 2^k-point DFT using the right-most tree and specified by $\mathbf{r}$ is*

$$T_C(\mathbf{r}) = 2^{k-r_0} \cdot g^R_{r_0}(k - r_0) + \sum_{i=1}^{m} 2^{k-r_i} \cdot g^L_{r_i}(k_{i-1}) + \sum_{i=1}^{m} a_1 \cdot 2^{k-k_{i-1}} + a_2 \cdot 2^{k-r_i} + a_3 \cdot 2^{k-k_i}.$$

Proof: The theorem is easily proved by using the lemmas given above. □

Values of a_1, a_2, and a_3 can be determined simply by counting the number of assembly instructions in the body of our recursive subroutine. In the framework of SPIRAL, this counting is done by the formula translator (see Figure 1 in Chapter 1).

Formulation of the Optimization Problem

Given a fixed k, find a vector $\mathbf{r}$ that minimizes the cost function given by Theorem 11.

Evaluation of the Model

To evaluate the performance model given by Theorem 11, we ran a simulation that searches exhaustively for the optimal breakdown tree over the set of all possible breakdown trees for each fixed value of k and finds the optimal one. Results of evaluating the model and its comparison to experiment driven search methods are presented in Chapter 7.

5.4 Conclusions

We derived two analytical cost models in this chapter. The first model is coarse and generic. It is useful in advancing our understanding of the architecture of the optimal tree. It defines a framework for comparing the performance of different small-code-modules, and can be used for partitioning the search space of all breakdown trees. However, it cannot select a single tree, as it accounts only for a number of small-code-modules to be used and not for their position in the tree.

The second model improves the first model by taking into account different types of overhead that occur during actual computation. This model is cache-sensitive, as it real-

izes that access cost to memory is not constant throughout the computation. This is an implementation driven analytical model. It is custom-tailored to the best implementation we found, so that it can be trained to lead to very good cost predictions.

6 Optimal Implementation Search Methods

6.1 Introduction

Motivation The optimal implementation is found by searching for an optimal breakdown strategy for each possible code, and then by choosing the best combination of code and breakdown strategy.

Special search methods for finding the optimum are required to search over the set of all possible alternatives because this set is extremely large.

Goal The goal of this chapter is to derive effective search methods over a set of Cooley-Tukey breakdown trees to find the best one.

We concentrate on searching over the space of breakdown trees for 2^k-point DFTs. The size of the search space for 2^k-point DFTs is $O(4^k)$. Since it takes about 1 sec. to get runtime estimates with high enough accuracy (of about 1%) using the experimental measurement of performance of Chapter 4, we would need 14 hours to search exhaustively for the 1024-point DFT. Thus an exhaustive search method is unfeasible with our experimental measurement of performance, so we use dynamic programming.

We now explain in detail three search procedures.

6.2 Exhaustive Search

Exhaustive search is trivial. We take the set of all possible implementations, and using either the experimental measurement of performance of Chapter 4 or the analytical performance models of Chapter 5 we calculate the performance for each implementation in the set. The implementation that has the best performance is chosen as the optimal one.

The results of evaluating the exhaustive search method are presented in Chapter 7.

6.3 Dynamic Programming Approach

The dynamic programming (DP) approach solves the search problem by combining the search solutions to sub-problems. It is highly efficient when sub-problems are not independent and share sub-sub-problems. Dynamic programming solves every sub-sub-problem just once, and then saves its answer in a table for future reuse. It is used in a bottom-up fashion, i.e., first, problems of small sizes are solved, and then these solutions are used to solve problems for larger sizes. Thus, the solutions to larger size problems are defined in terms of solutions to identical problems of smaller sizes, which is the property of recursion. An object is said to be recursive if it is defined in terms of itself. Hence, the basic requirements imposed on the possible optimal solution to the problem are that it be defined recursively via solutions to sub-problems. In other words, if the optimal tree does not have a recursive structure, it cannot be found by the dynamic programming approach.

The optimal solution found by dynamic programming for a 2^k-point DFT uses optimal solutions to the smaller DFT sizes.

Dynamic programming cannot be defined in our full search space because not every tree exhibits a recursive structure, while, as we discussed above, an optimal tree found by dynamic programming is defined recursively. If we assume that only trees with recursive structure can be optimal, then applying the dynamic programming to the sub-space of such trees leads to the optimal tree.

Sub-Optimality Assumption: *The performance of computing an n-point DFT is only a function of n and its breakdown tree, and does not depend on the context in which it is computed.*

Stated differently, take a breakdown tree for computing a DFT, e.g., 2^8-point DFT, as shown by the tree on the left in Figure 21. Assume it has two equal sub-trees, e.g., 2^3-point DFT broken down as $2^3 = 2^1 \cdot 2^2$ in both cases. Then our assumption says that the performance of computing these sub-trees is the same, independently of where they are

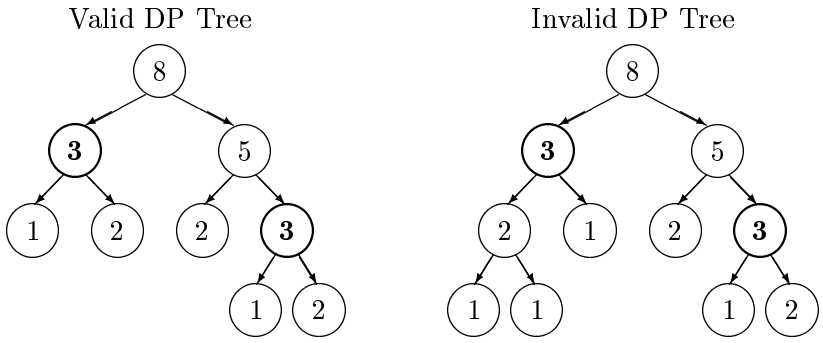


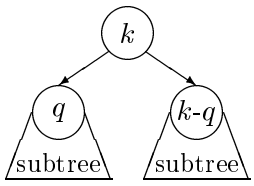
Figure 21: An example of a DP-valid and a DP-invalid tree

located in the tree.

By making this assumption, we claim that trees that contain non-equal sub-trees with identical parent nodes cannot have a better performance than ones with equal sub-trees, as is the case with the tree on the right in Figure 21, where $2^3 = (2^1 \cdot 2^1) \cdot 2^1$ for one node, and $2^3 = 2^1 \cdot 2^2$ for another node. This is because by replacing a sub-tree by a sub-tree with a better performance we obtain a tree that has a equal or better performance. Dynamical programming reduces substantially the search space by eliminating many trees that, according to our assumption, do not need to be considered.

When Does Dynamic Programming Hold? Our assumption holds if the performance of a signal processing transform is independent of the state of the memory hierarchy, and of the stride parameter, as defined in Chapter 3. In particular, this would be true when the access cost to memory is constant. We will discuss this topic in more detail in Chapter 7. There we also investigate if dynamic programming leads to a tree with near optimal performance even if the memory access cost is not constant.

Procedure for Applying DP to the Subset of DP-Valid Breakdown Trees We start with $k = 1$. There is only one tree for k=1, which is to compute a 2^1-point DFT



where $q = 1, 2, \ldots, k-1$

Figure 22: Finding an optimal tree for k by using optimal sub-trees for q and $k-q$

by definition, so it is already optimal. Now suppose that we already found the optimal breakdown trees $\forall k' < k$. We find the optimal tree for k. There are exactly $k-1$ ways of representing k as a sum of two positive numbers $k = q+(k-q)$, namely for $q = 1, 2, \ldots, k-1$. This corresponds to a split $2^k = 2^q \cdot 2^{k-q}$, i.e., we are computing a 2^k-point DFT using 2^q-point DFTs and 2^{k-q}-point DFTs. Since both, $q < k$ and $k - q < k$, we already know the optimal breakdown trees for both of them, as pictorially shown in Figure 22. We can also introduce the special case of $q = 0$, which we interpret as computing the DFT from the definition, i.e., without using a Cooley-Tukey formula. Then, we just need to try all possible k combinations of breaking k and using optimal breakdown trees for both left and right children of k to find an optimal breakdown tree for the 2^k-point DFT.

DP Search Space Size *The DP search space is only $O(k^2)$ compared to the original search space size of $O(4^k)$.*

Proof: Proof is easily obtained from the above-described procedure. □

Compact Representation for DP-valid Trees A compact representation for DP-valid trees is possible. To apply dynamic programming, we start with $k = 1$, and sequentially increment k by 1, choosing at each step a single value of q. This produces the optimal breakdown tree. Thus, we can represent all optimal trees of sizes $2^{k'}$ for $k' = 1, \ldots, k$ using only k values. Figure 23 shows an example of doing this. A stored value of 0 means that the DFT is computed directly from the definition. We will refer to this storage mechanism

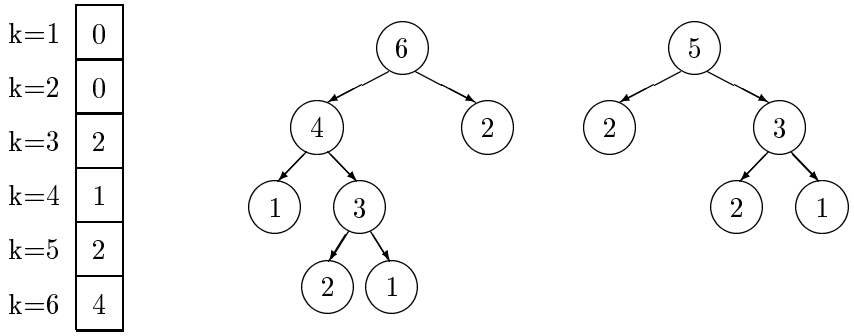


Figure 23: Compact DP-valid tree storage using a DP breakdown vector

as a **DP breakdown vector**.

The program that implements the Cooley-Tukey formula at each stage of the recursion needs to determine how to breakdown further. By representing trees in a compact and easy way to access, we reduce significantly the overhead. As we will see in Chapter 5, this has an exponentially growing impact on the total performance. Thus, the compact form of representing DP-valid trees should improve the overall performance.

Conclusions The dynamic programming approach dramatically reduces the search space. At the same time it simplifies the representation of trees, leading to less overhead on each step of the recursion. However, the assumption implied by dynamic programming is that the memory access cost is independent of the DFT computation context (accessing data elements at different strides, as explained in Chapter 3). We will discuss in Chapter 7 the validity of this assumption.

6.4 Soft Decision Dynamic Programming

In the dynamic programming approach, described in Section 6.3, at each step of a recursion, we represent $k = q + (k - q)$, where $q = 1, 2, ..., k - 1$, and we search for an optimal performance breakdown, keeping breakdowns for all sub-problems fixed. For each value of k, we then store only a single value of q. Thus, we make a hard decision when

choosing the optimum. We refer to this version of DP as hard decision DP. Hard decision DP will work fine for as long as the DP assumption is not violated. However, it is expected that under some circumstances the DP assumption might not hold.

We extend here the dynamic programming approach. At each step of the recursion, instead of storing only the breakdown with the smallest runtime, we store several breakdowns, those having the smallest runtimes. At search time, instead of picking a single optimal breakdown at each sub-problem, we search over all candidates stored for that sub-problem.

The number of candidates stored at each step of the soft decision dynamic programming approach will be referred to as **soft decision depth** D. By adjusting D, soft decision dynamic programming degenerates into either the dynamic programming (when $D = 1$), or, exhaustive search (when D is sufficiently large). If the number of candidates is constant for all values of k, then the search space size is D times bigger than the search space size for regular dynamic programming. It will be $O(\frac{D \cdot k^2}{2})$.

We can develop a compact way for representing the trees for soft decision DP. This extends the representation derived for the hard decision dynamic programming. Instead of storing a breakdown vector of left nodes, we will have a breakdown matrix of left nodes, a left-index matrix, and a right-index matrix. An example is shown in Figure 24. Suppose we want to read the structure of the tree starting in the location $k = 9, i = 1$. We look into the corresponding entries in the breakdown matrix, the left-index matrix, and the right-index matrix. The value of 2 in the breakdown matrix means that we split $2^9 = 2^2 \cdot 2^7$. Thus, we need to find out how the right factor 2^2 was computed, and how the left factor 2^7 was computed. For the left factor we read a value from the left-index matrix at location $k = 9, i = 1$, which is 2. This means that we compute a 2^7-point DFT by using an entry $k = 7, i = 2$. For the right factor similarly we use an entry $k = 2, i = 0$. We repeat the process described above recursively for both $k = 7, i = 2$ and $k = 2, i = 0$, until we reach a value of 0 in the breakdown matrix, which means that we reached a leaf.

	breakdown			left-index			right-index		
	i=0	i=1	i=2	i=0	i=1	i=2	i=0	i=1	i=2
k=1	0	0	0	0	0	0	0	0	0
k=2	**0**	0	0	**0**	0	0	**0**	0	0
k=3	0	0	0	0	0	0	0	0	0
k=4	2	1	3	0	0	0	0	0	0
k=5	2	**3**	1	0	**0**	0	0	**0**	0
k=6	3	1	2	0	0	0	0	0	0
k=7	2	2	**2**	1	0	**1**	0	0	**0**
k=8	3	2	3	0	0	1	0	0	0
k=9	3	**2**	2	0	**2**	2	0	**0**	0

Figure 24: Example of matrices for $k = 9$ and soft decision depth $D = 3$

A nice property of this storage method is that it also does not require much overhead, when accessed at each step by a recursive breakdown program to determine how to breakdown further. This storage method also allows to store any tree by setting a soft decision depth D high enough, not just a DP-valid tree.

6.5 Conclusions

The dynamic programming approach dramatically reduces the search space. At the same time it simplifies the representation of trees, leading to less overhead at each step of the recursion. However, the assumption made is that memory access cost is independent of the DFT computation context. This assumption can be relaxed by introducing a soft decision dynamic programming, for which a search space size is of the same order, while its structure supports any arbitrary tree.

Both hard-decision and soft-decision dynamic programming strategies are general and

not limited to the 2-power FFT. They will work universally for a family of signal processing algorithms.

7 Test Results and Analysis

7.1 Testing Platform

All experiments were done on an Intel Pentium II CPU based computer with 450 Mhz internal clock and 384MB of SDRAM, running WinNT 4.0 Workstation. The compiler used was Microsoft Visual C++ 6.0 compiler, which allows an easy integration of C++ and assembly. Visual C++ Compiler optimizations do not affect subroutines written in assembly. This compiler also does not try to change the order of assembly instructions in order to achieve a better parallelism, so our assembly instructions were executed exactly in the order we wrote them. When we were benchmarking third party packages, we used a "release mode" of the compiler, which enables all optimizations. Benchmarking of both ours and third party packages was done using our benchmarking tool, described in Chapter 4, so that the comparison is fair.

7.2 Comparing FFT-BR, FFT-RT and FFT-TS Algorithms

The goal of this paragraph is to compare the performance of the three different algorithms that we derived in Chapter 3. The Figure 3 of Section 2.6 presents the functional block diagram of our system framework. For comparing the FFT-BR, FFT-RT-3 and FFT-TS algorithms, we fix one algorithm at a time in the left-top block of this figure, and obtain a set of implementations for this particular algorithm. In this way we obtain three sets of implementations, each corresponding to one of the three algorithms. Then we search for the optimal implementation in each of these three sets and obtain three near optimal implementations to characterize the given three algorithms. In order to perform the search, we still have to choose the way the performance is going to be evaluated, and the way the search for the optimum is going to be performed. For performance evaluation we use the experimental measurement of performance, derived in Chapter 4. We do not use analytical measurement of performance in this experiment because we have not presented the results

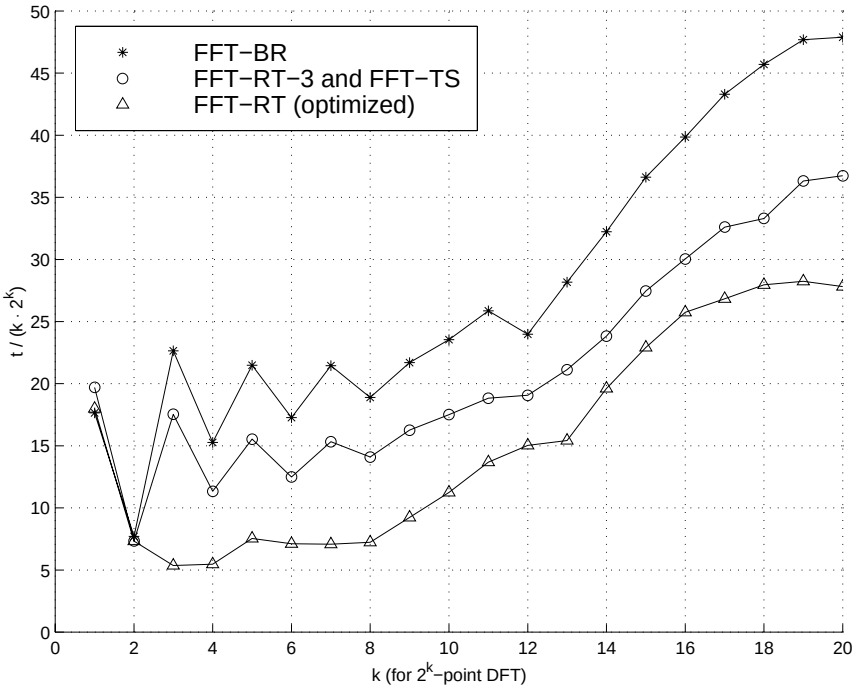


Figure 25: Comparing different algorithm implementations

that confirm its validity. In search of the optimum we cannot use the exhaustive search to make our experiment feasible. For this reason, we use the dynamic programming approach, given in Chapter 6.

The results of the comparison are presented in Figure 25. It compares the three optimal implementations for the FFT-BR, FFT-TS and FFT-RT-3 algorithms. For reference we provide a curve for the FFT-RT algorithm implementation as well. The first curve is for the FFT-BR algorithm implementation, while the second one is for both the FFT-RT-3 and the FFT-TS algorithm implementations. The second curve lies lower than the first one, which means that FFT-RT-3 and FFT-TS algorithms are better than the FFT-BR algorithm. It turned out that the best breakdown tree for FFT-TS is always the right-most tree. The FFT-TS and FFT-RT-3, as it was noted in Chapter 3, are equivalent for right-most trees.

This fact explains why the curves for FFT-RT-3 and FFT-TS coincide. The difference between them is that FFT-RT-3 does not support any other trees, while FFT-TS supports arbitrary trees, but has to use a temporary storage when the tree is not a right-most one. We can conclude that the use of temporary storage penalizes the FFT-TS implementation for non right-most trees. Thus, there is no advantage of using the FFR-TS algorithm, as we have to consider many more implementations for it. The third curve corresponds to the FFT-RT algorithm, which is much faster than the FFT-RT-3 algorithm, because it uses more small-code-modules and takes the advantage of more optimizations presented in Chapter 3.

We conclude that it is enough to consider only the right-most trees using the FFT-RT algorithm.

Statement *Right-most tree algorithms are not penalized for temporary storage, thus a well written code of the algorithm for right-most trees, such as our optimized code of the FFT-RT algorithm, will be faster than any other code for algorithms supporting all trees.*

7.3 Evaluating the Cache-Sensitive Cost Model and the Dynamic Programming Approach

Goal The goals of this section are to confirm the validity of the analytical cache-sensitive cost model for the FFT-RT algorithm derived in Chapter 5, and to verify that the dynamic programming approach is accurate even in case its assumptions, given in Section 6.3, are violated.

In the functional block diagram, presented by Figure 3, we need to choose a method of evaluating the performance, and then use some optimum search method to search over the set of performances. We find the first optimum using the analytical cache-sensitive cost model, described by Theorem 11, in combination with the exhaustive search. The

second optimum is found by using the analytical cache-sensitive cost model and the dynamic programming approach. They are compared against the optimum found using the experimental performance measurement and the dynamic programming approach (see Section 6.3). We do not provide the optimum for the experimental performance measurement and the exhaustive search combination, because it is not feasible to run such an experiment (it takes an unrealistic amount of time to complete it). In all three cases we consider only the best code derived from the FFT-RT algorithm for the right-most trees. Since both the algorithm and the code are fixed, the only degree of freedom in finding the optimum is to vary over the set of all right-most breakdown trees. Once the optimal breakdown trees for the three approaches are found, we measure their runtime using the experimental performance evaluation tool, and compare them against each other. The result of the comparison is presented in Figure 26.

The first curve in Figure 26 corresponds to the runtime measurements for the optimal breakdown tree, found by the analytical model with the exhaustive search. The second curve corresponds to the runtime measurements for the optimal breakdown tree, found by the analytical model combined with the dynamic programming approach. The third curve corresponds to the runtime measurement for the optimal breakdown tree, found by using the experimental measurement tools and the dynamic programming approach. We can see that all three optimums lie within approximately 10% percent from each other.

We consider the distributions of runtimes of all possible right-most trees for different values of k. These distributions show that the spread of runtimes from minimum to maximum is above 100% for $k > 8$. As an example, we present distributions of runtimes for $k = 11$ and $k = 14$ in Figure 27. The runtime estimates are obtained using the cost model given by Theorem 11. The horizontal axis is a normalized runtime, and the vertical axis is histogram values. Optimal trees are the ones that are located in the very first bin. We see that there are only few optimal or near to optimal trees.

Based on everything stated above, we conclude that the three optimums presented in

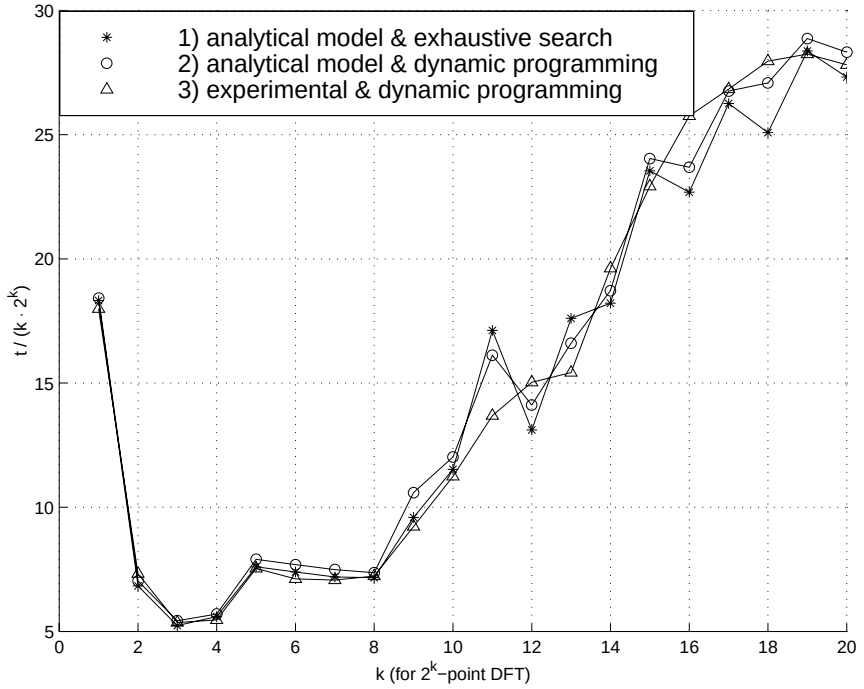


Figure 26: Evaluating the cache-sensitive cost model

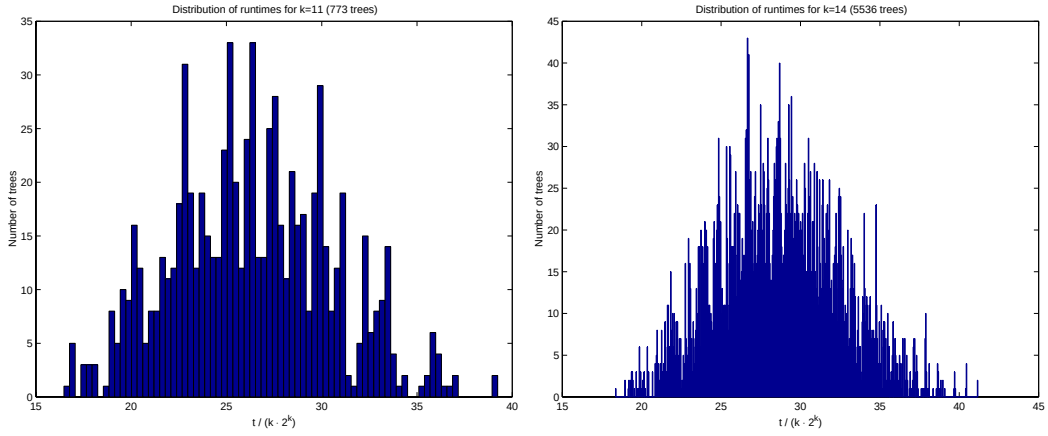


Figure 27: Distribution of runtimes for different values of $k = 11, 14$

Figure 26 are within a few percent ($< 3\%$) of "efficient" trees, i.e., the trees whose runtime is smaller than the runtime of the majority ($> 97\%$) of possible right-most trees.

Statement *The cache-sensitive cost model is accurate. It can be used in conjunction with either the exhaustive search or the dynamic programming approach to find one of the near-to-optimum trees.*

From Figures 26, 27 we conclude that the optimal breakdown tree found by dynamic programming lies in among the few best breakdown trees. This shows that dynamic programming can lead to very good trees even when the dynamic programming assumptions are violated, i.e., for large values of k (see Section 6.3).

Statement *Dynamic programming finds near-to-optimum breakdown trees even when its assumptions are violated. It can be used with either the experimental measurement of performance or with the analytical cache-sensitive cost model.*

The dynamic programming approach in conjunction with the analytical model finds the efficient tree much faster than other approaches, while the tree found by it is still very good.

Statement *The dynamic programming approach used with the analytical cache-sensitive cost model is extremely fast and accurate at the same time.*

7.4 The Best Implementation vs. FFTW

A very effective FFT system, called the **FFTW**, was developed by Frigo and Johnson, [10, 11, 12]. This is the most efficient FFT package currently available. The FFT computation runtime for this package is less than the runtime for all other existing DFT software, including FFTPACK, [35], and the code from Numerical Recipes, [8]. For this reason, we are going to compare our results only against FFTW.

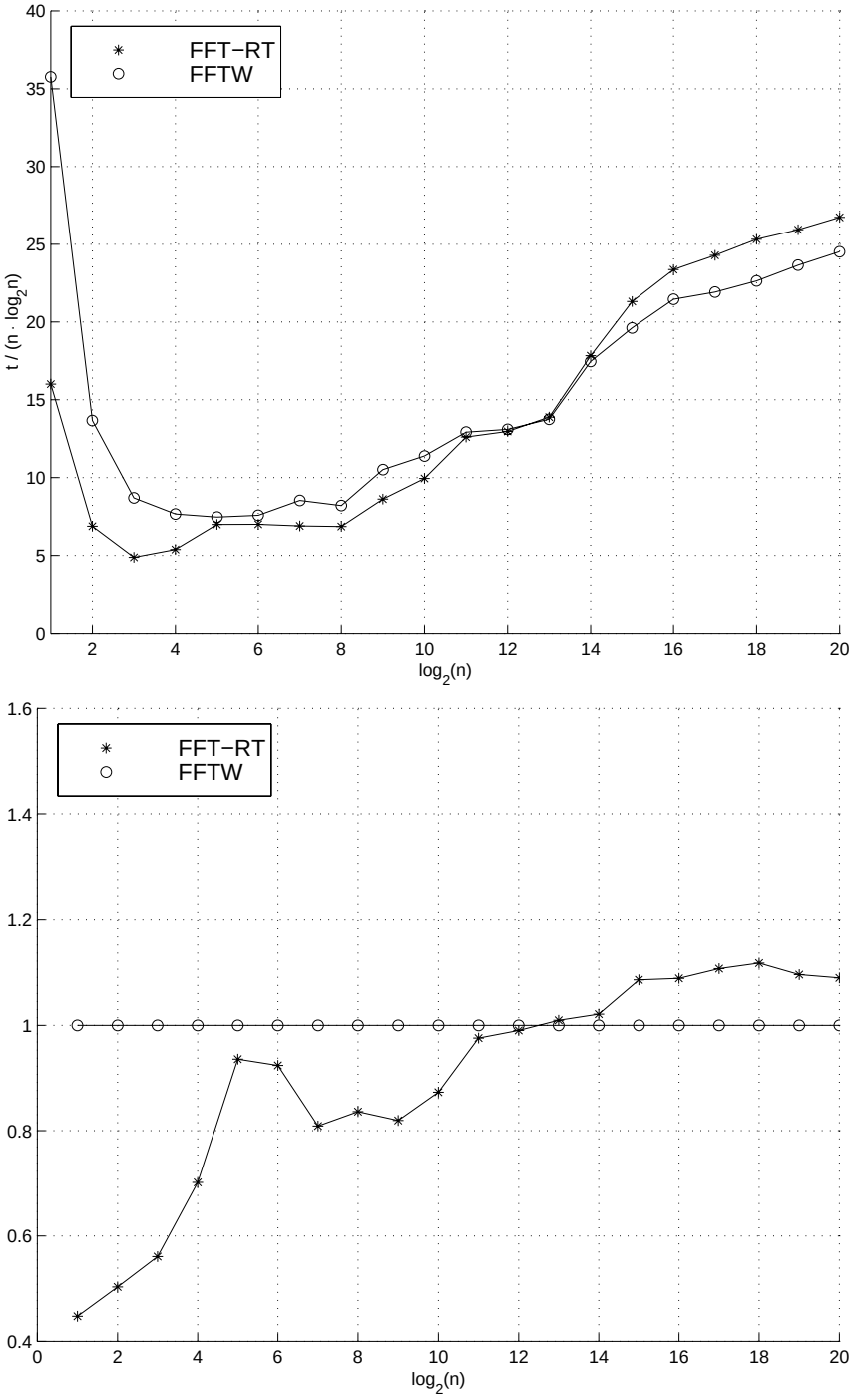


Figure 28: The Best Implementation vs. FFTW

k	Full Space Size	DP Space Size	Our Runtimes (Clock Cycles)	FFTW Runtimes (Clock Cycles)
1	N/A	N/A	32	72
2	N/A	N/A	55	109
3	N/A	N/A	117	209
4	N/A	N/A	344	490
5	15	4	1,118	1,195
6	29	9	2,687	2,908
7	56	15	6,181	7,643
8	108	30	14,044	16,796
9	208	22	39,689	48,448
10	401	39	101,823	116,616
11	773	49	283,919	290,908
12	1490	60	637,433	643,725
13	2872	72	1,478,281	1,464,090
14	5536	85	4,090,822	4,005,480
15	10671	99	10,475,971	9,642,840
16	20569	114	24,506,440	22,500,000
17	39648	130	54,122,631	48,857,100
18	76424	147	119,469,760	106,843,000
19	147312	165	258,359,370	235,671,000

Table 5: Numerical Values ($450,000,000$ clock cycles = 1 sec.)

Figure 28 compares our FFT package vs. the FFTW package. To derive our timings, we use our best code for the FFT-RT algorithm given by Pseudo-code 19, and find the near optimal implementation using the analytical cache sensitive cost model in conjunction with dynamic programming. Our implementation considerably outperforms FFTW for sizes up to $k = 13$. For sizes above $k = 13$, FFTW is up to 10% faster than our implementation. The numerical values of the evaluation are given in Table 5.

7.5 Conclusions

We compared the performance of the three major types of the FFT algorithms with completely different characteristics: FFT-BR; FFT-RT; FFT-TS.

We concluded that the right-most tree algorithms are not penalized for temporary storage, thus a well written code of the algorithm for right-most trees, such as our optimized code of the FFT-RT algorithm, will be faster than any other code for algorithms supporting all trees. In line with this conclusion, we considered only right-most trees in further experiments.

We confirmed the validity of the analytical cache-sensitive cost model for the FFT-RT, which can be used in conjunction with either exhaustive search or the dynamic programming approach to find one of the near-to-optimum trees. We also verified that the dynamic programming approach is accurate even in case its assumptions are violated. Dynamic programming can be used with either the experimental measurement of performance or with the analytical cache-sensitive cost model. The dynamic programming approach used with the analytical cache-sensitive cost model is preferable, as it is extremely fast over searching the possible implementations and accurate in finding near optimal implementation at the same time.

Based on the experiments, we decided to use the combination of the analytical cache-sensitive cost model with the dynamic programming approach as our FFT package.

We compared our package against the best FFT package currently available, the FFTW.

The implementations found by our package considerably outperform implementations found by the FFTW package for sizes up to $k = 13$. For sizes above $k = 13$, FFTW implementations are up to 10% better than ours. A main advantage of our package is that it finds near optimal implementations orders of magnitude faster than the FFTW package, while its implementation runtimes lie within the same range.

8 Conclusions and Future Work

The main result of this work lies in developing systematic methodologies for finding fast implementations of signal processing discrete transforms within the framework of the 2-power point fast Fourier transform (FFT). By employing rewrite rules (e.g., the Cooley-Tukey formula), we obtain a divide and conquer procedure (decomposition) that breaks down the initial transform into combinations of different smaller size sub-transforms, which are graphically represented as breakdown trees. Once the sub-transforms have reached a sufficiently small size so that their computation can be performed efficiently in the context of a particular computational platform, a significantly enhanced performance is observed. Recursive application of the rewrite rules generates a set of algorithms and alternative codes for the FFT computation. The set of "all" possible implementations (within the given set of the rules) results in pairing the possible breakdown trees with the code implementation alternatives.

The process of deriving the different code alternatives is done in two steps. First, we derive major algorithms, and then, we obtain the optimized codes for them. We have derived three major types of algorithms for the FFT with completely different characteristics: recursive in-place bit-reversed algorithm FFT-BR; recursive algorithm with temporary storage FFT-RT; out-of-place algorithm for right-most trees FFT-TS, and compared their performance. Right-most tree algorithms are not penalized for temporary storage, thus a well written code of the algorithm for right-most trees, such as our optimized code of the FFT-RT algorithm, is faster than any other code for algorithms supporting all trees. In line with this conclusion, we have concentrated only on right-most trees in our experiments.

To achieve a good runtime performance, we have developed small-code-modules in the Intel Pentium assembly for the DFTs of small sizes ($n = 2, 4, 8, 16$), and optimized their computation. We have tackled the problem of minimizing the runtime by reducing the total number of temporary variables, instructions, and loads/stores. Memory load and store

instructions were reordered in a way that reduced cache misses due to collisions occurring when small-code-modules are called at large strides, aligned to the cache size. Since in-place butterfly computation is the most commonly done operation, we have derived an efficient code for implementing it. In addition, to achieving better performance, special attention has been paid to memory alignment for data accesses.

We have also come to the conclusion that the multiplication by twiddle factors during the recursive computation is a significant fraction of the total computation time, so twiddle factors were pre-computed and stored in an order that reduces cache misses.

These optimizations combined together with the FFT-RT algorithm allowed us to derive its optimized code, supporting different breakdown trees, for the Intel Pentium architecture. In order to find an efficient way of computing a 2-power FFT of a given size, our package tries all possible combinations of breakdown trees, and finds the near optimal one.

For obtaining reproducible runtime estimates with desired accuracy in a minimal possible time, a benchmarking strategy has been devised. Based on this strategy, an accurate and consistent benchmarking tool has been proposed. This benchmark tool has been used to compare the performance of different implementations of the FFT.

Our major effort has been applied to developing analytical models that can predict the performance of any FFT implementation much faster than running the actual experiment. We have developed two such analytical cost models. The first model is coarse and generic. It is useful in advancing our understanding of the architecture of the optimal tree. It defines the framework for comparing the performance of different small-code-modules, and can be used for partitioning the search space of all breakdown trees. However, it cannot select a single tree, as it accounts only for a number of small-code-modules to be used and not for their position in the tree. The second model improves the first one by taking into account different types of overhead that occur during actual computation. This model is cache-sensitive, as it realizes that access cost to memory is not constant throughout the computation. This is an implementation driven analytical model. It is custom-tailored to

the best implementation we found – FFT-RT – so that it can be trained to lead to very good cost predictions.

A significant finding in this study is that the dynamic programming approach dramatically reduces the search space and gives good grounds for the generic and fastest SP transform implementation. We have applied the dynamic programming approach over a set of Cooley-Tukey breakdown trees for finding the best one. It also simplified the representation of trees, leading to less overhead at each step of the recursion. To use dynamic programming we had to impose a very strong assumption that memory access cost is independent of the DFT computation context. We have relaxed the assumption by introducing a soft decision dynamic programming, for which the search space size is of the same order. Both hard-decision and soft-decision dynamic programming strategies are general and not limited to the 2-power FFT. They can work universally for a family of signal processing algorithms.

We have confirmed the validity of the analytical cache-sensitive cost model for the FFT-RT, which can be used in conjunction with either exhaustive search or the dynamic programming approach to find one of the near-to-optimum trees. We have also verified that the dynamic programming approach is accurate even in case its assumptions are violated. The dynamic programming can be used with either the experimental measurement of performance or with the analytical cache-sensitive cost model. The dynamic programming approach used with the analytical cache-sensitive cost model is preferable, as it is extremely fast over searching the possible implementations and accurate in finding near optimal implementation at the same time.

Based on our experiments, we have decided to use the combination of the analytical cache-sensitive cost model with the dynamic programming approach as our FFT package.

The comparison of the developed package with one of the best available FFT packages – the FFTW – was carried out. The implementations found by our package considerably outperforms implementations found by the FFTW package for sizes up to $k = 13$. For

sizes above $k = 13$, FFTW implementations are up to 10% better than ours. But the main advantage of our package is that it finds near optimal implementations orders of magnitude faster than the FFTW package, while its implementation runtimes lies within the same range. Of course, FFTW runs, in contrast to our package, on any platforms and supports sizes that are not 2-powers.

In order to obtain the above stated results in developing the package that automatically finds the near optimal implementations of the FFT, we have taken a system approach to the problem. We do not choose the optimal code and the optimal breakdown tree independently of each other. Rather, we combine all possible code alternatives with all possible allowed breakdown trees, and then, employing performance models and search strategies that we have developed, we choose their best combination. By combining good algorithms and good codes with accurate performance evaluation models and effective search methods we obtain efficient FFT implementations. They are universal and could be applicable not only to the FFT, but to many other signal processing transforms.

Future Work

Localizing Data Access in the DFT Computation As a result of the cache-sensitive performance model we have found a new implementation of the FFT for right-most trees, which reduces data cache misses by localizing the data access with large arrays. This is achieved by realizing that data dependencies for right-most trees can be defined recursively, thus leading to a new recursive program, which reorders the execution of small-code-modules for left leaves in the order that minimizes the total number of cache misses. We will implement this idea and verify what further improvement can be gained.

References

[1] J. M. F. Moura, J. R. Johnson, R. V. Johnson, D. Padua, V. Prasanna, M. M. Veloso, "SPIRAL: Portable Library of Optimized Signal Processing Algorithms," `http://www.ece.cmu.edu/~spiral`

[2] L. Auslander, J. R. Johnson, and R. W. Johnson, "Automatic Implementation of FFT Algorithms," Technical Report, Department of MCS, Drexel University, Philadelphia, PA, 1996.

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

[4] A. K. Jain, *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[5] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*. Heidelberg, Germany: Springer-Verlag, second ed., 1982.

[6] A. V. Oppenheim, R. W. Schafer *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[7] K. Spindler, *Abstract Algebra with Applications*. New York, NY: Marcel Dekker, Inc., 1989.

[8] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, "Fast Fourier Transform," *Numerical Recipes in Fortran 77: The Art of Scientific Computing*, 2nd ed., Cambridge, England: Cambridge University Press, ch. 12, pp. 490-529, 1992.

[9] Intel Co., *Pentium Family of Processors Software Developer's Manual.* `http://developer.intel.com/design/PentiumII/manuals`

[10] M. Frigo, "A Fast Fourier Transform Compiler," *Laboratory for Computer Science*, MIT, Cambridge, MA, February 1999.

[11] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Laboratory for Computer Science*, MIT, Cambridge, MA, September 1997. Also, ICASSP-98 Proceedings, vol. 3, p. 1381, 1998.

[12] M. Frigo and S. G. Johnson, "The Fastest Fourier Transform in the West," Tech. Rep. MIT-LCS-TR-728, *Laboratory for Computer Science*, MIT, Cambridge, MA, Sep. 1997.

[13] C. S. Burrus, "Notes of the FFT," `http://www-dsp.rice.edu/research/fft/fft-note.asc`

[14] D. P. Kolba and T. W. Parks, "A Prime Factor FFT Algorithm using High Speed Convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, pp. 281-294, August 1977.

[15] H. W. Johnson and C. S. Burrus, "The Design of Optimal DFT Algorithms using Dynamic Programming," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 31, pp. 378-387, April 1983.

[16] S. Winograd, "On Computing the Discrete Fourier Transform," *Mathematics of Computation*, vol. 32, pp. 175-199, January 1978.

[17] S. Winograd, "On the Multiplicative Complexity of the Discrete Fourier Transform," *Advances in Mathematics*, vol. 32, pp. 83-117, May 1979.

[18] S. Winograd, *Arithmetic Complexity of Computation*. SIAM CBMS-NSF Series, No. 33, Philadelphia: SIAM, 1980.

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

[20] P. Duhamel, "Implementation of 'Split-radix' FFT Algorithms for Complex, Real, and Real-symmetric Data," *IEEE Trans. on ASSP*, vol. 34, pp. 285-295, April 1986.

[21] M. Vetterli and P. Duhamel, "Split-radix Algorithms for Length - p^m DFT's," *IEEE Trans. on ASSP*, vol. 37, pp. 57-64, January 1989.

[22] R. Stasinski, "The Techniques of the Generalized Fast Fourier Transform Algorithm," *IEEE Transactions on Signal Processing*, vol. 39, pp. 1058-1069, May 1991.

[23] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, "On Computing the Split-radix FFT," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, pp. 152-156, February 1986.

[24] R. N. Bracewell, *The Fourier Transform and its Applications.* New York: McGraw-Hill, 1965.

[25] R. N. Bracewell, *The Hartley Transform.* Oxford Press, 1986.

[26] M. Vetterli and H. J. Nussbaumer, "Simple FFT and DCT Algorithms with Reduced Number of Operations," *Signal Processing*, vol. 6, pp. 267-278, August 1984.

[27] F. M. Wang and P. Yip, "Fast Prime Factor Decomposition Algorithms for a Family of Discrete Trigonometric Transforms," *Circuits, Systems, and Signal Processing*, vol. 8, no. 4, pp. 401-419, 1989.

[28] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications.* San Diego, CA: Academic Press, 1990.

[29] R. Singleton, "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Transactions on Audio and Electroacoustics*, vol. AU-17, pp. 93-103, June 1969.

[30] J. A. Glassman, "A Generalization of the Fast Fourier Transform," *IEEE Transactions on Computers*, vol. C-19, pp. 105-116, Feburary 1970.

[31] W. E. Ferguson, Jr., "A Simple Derivation of Glassman general-N Fast Fourier Transform," *Computation and Mathematics with Applications*, vol. 8, no. 6, pp. 401-411, 1982.

[32] H. Guo and C. S. Burrus, "Fast Approximate Fourier Transform via Wavelet Transforms," *IEEE Transactions on Signal Processing*, January 1997.

[33] C. S. Burrus, R. A. Gopinath, and H. Guo, *Introduction to Wavelets and the Wavelet Transform.* Upper Saddle River, NJ: Prentice Hall, 1998.

[34] J. E. Hicks, "A High-Level Signal Processing Programming Language," MIT/LCS/TR-414, *Laboratory for Computer Science*, MIT, Cambridge, MA, March 1988.

[35] P. N. Swarztrauber, "Vectorizing the FFTs," *Parallel Computations*, G. Rodrigue ed., pp. 51-83, February 1982.