Engineering
a
Fast Fourier Transform

Jørgen Fogh, 20072967

Master's Thesis, Computer Science
June 2013
Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Computing the discrete Fourier transform is one of the most important in applied computer science, with applications in fields as diverse as seismology, signal analysis, and various branches of engineering.

A great many algorithms exist for quickly computing different variations of the transform – fast Fourier transforms, or FFTs.

Because of their practical importance, even small improvements to the running time of an FFT implementation are important and much research has gone into improving the constant factors of algorithms, both in a theoretical and an applied context.

In this thesis I have sought to discover whether modern general-purpose compilers have become so advanced that they can generate optimal code for an FFT implementation automatically or if the programmer must perform compiler transformations manually in order to create an efficient implementation.

To answer this question, a number of FFT implementations were developed, primarily using simple transformations, which could have been performed by the compiler but were not.

During the development process, I discovered a new variation of the Cooley-Tukey factorization of the DFT matrix, which corresponds to a different evaluation order of the same computation as the normal Cooley-Tukey FFT. This evaluation order is shown to be faster than the ordinary evaluation order in practice, even though the theoretical complexity is the same.

I finally conclude that mid-low level optimizations can significantly improve an FFT implementation and that a modern general-purpose compiler does not generate optimal code without significant support from the programmer.

Acknowledgments

I would like to thank my thesis advisor, Gerth S. Brodal. Jesper A. S. Nielsen and Esben Andreasen deserve a special thanks for proof-reading my manuscripts when Gerth was too busy. I would also like to thank Olivier Danvy for giving me valuable advice throughout my studies. Last, but certainly not least, this thesis would not have been written without the lavish financing provided by the hardworking danish tax payers. They deserve much more praise than they usually receive.

Jørgen Fogh,
Aarhus, June 30, 2013.

Contents

List of Figures

Chapter 1

Introduction

1.1 History & Previous Work

The *discrete Fourier transform* or *DFT* is one of the most important problems in applied computer science, with applications in many areas of scientific computing and engineering.

The first *fast Fourier transform* or *FFT* was invented by Carl Friedrich Gauss[B10] in 1805 (even predating Fourier's work on harmonic analysis by two years) and reinvented by James W. Cooley and John W. Tukey in 1965[B7]. The name FFT is used today to refer to any "fast" method of computing the DFT, usually $O(N \log N)$. Sometimes the term is used more specifically to refer to some version of the Cooley-Tukey algorithm, typically for input sizes which are powers of 2.

A good theoretical survey of general FFTs can be found in [A2]. However, the paper only goes so far as to derive equations describing the recursive structures used to compute the DFT. It does not include the kind of detailed description of the algorithms needed to write a fast implementation in practice.

Charles Van Loan's monograph[B13] provides a beautiful and coherent treatment of several FFT algorithms, describing them all in terms of matrix algebra. This book also includes pseudo-MatLab of the algorithms, which is quite useful even though the comments about hardware are now a bit dated. To the best of my knowledge it is the only treatise which covers the relevant correctness proofs for all the major FFT frameworks and connects them with pseudo-code, all in a single monograph.

The Fastest Fourier Transform in the West (FFTW)[A3] is a highly general FFT implementation for general-purpose CPUs. It can efficiently compute FFTs of any size as well as multi-dimensional FFTs. It also has special support for the case where the input consists of real numbers.

1.2 The Purpose of this Thesis

This thesis seeks to answer the question: "Can the efficiency of the implementation of an FFT algorithm be improved significantly without changing the algorithm employed?" That is, can the performance be improved by using mid-low level techniques, rather than by applying more advanced algorithmic techniques or automatic code generation.

This thesis' main focus is to engineer a fast FFT implementation by hand on a concrete platform (the Nehalem Microarchitecture) in order to explore the advantages of hand-tuning an algorithm, as opposed to generating and tuning the code automatically, like the popular FFTW framework does[A3].

Using efficient techniques at the mid or low level is not only useful in engineering a concrete implementation. Once more efficient programming techniques have been identified, they can gradually be refined and standardized, until they can be automated in new generations of compilers.

The programs engineered for this thesis only deal with problem sizes which are powers of 2. This makes the implementations simpler, since the algorithms for computing the DFT are intimately related to the factorization of the problem size. A discussion of these issues is outside the scope of the thesis but can be found in [B13, ch. 2].

I assume the reader is familiar with the basic concepts of linear algebra, so I will only cover material which I find to be unusual or particularly relevant.

1.3 Thesis Structure

The thesis has two main parts. Part one covers the relevant theory needed to understand the FFT implementations, which are covered in part two.

The first part is further subdivided into three chapters. Chapter 2 covers the mathematical theory underpinning the computation of FFTs of size 2^t. Chapter 3 describes the parts of modern processor microarchitectures which are relevant to FFT computations. Finally, chapter 4 is about optimizing compiler technology, since understanding the compiler is vital in order to generate efficient code in practice.

The second part is divided into four chapters. Chapter 5 describes the different FFT implementations. Chapter 6 describes the experiments performed and shows results. In Chapter 7 the results are analyzed, while chapter 8 contains the conclusion.

The thesis introduces many terms and acronyms with which the reader may not be familiar. The reader is advised to consult the index when encountering unfamiliar terms.

Part I

Theory

Chapter 2

The Discrete Fourier Transform

"I am convinced that life as we know it would be considerable different if, from the 1965 Cooley-Tukey paper onwards, the FFT community had made systematic and heavy use of matrix-vector notation!"

— Charles Van Loan[B13, p. ix]

The discrete Fourier transform is usually described in terms of scalars in the literature. The definition of the DFT is cast in terms of sums of scalars and the computations (the fast Fourier transforms) are cast in terms of recursive equations at the scalar level.

An alternative, pioneered by Charles Van Loan[B13], is to use the notational tools of linear algebra. Instead of nested sums he uses matrix-vector products and instead of recursive scalar equations he factors a matrix into sparse matrices.

The advantage of this approach is not only that it greatly simplifies both the proofs and presentation of the material. It also makes it possible to compare different computational approaches more directly, in terms of the matrix factorizations they correspond to.

Definition 2.0.1. The size N *discrete Fourier transform* or *DFT* is the operation of multiplying a vector $x \in \mathbb{C}^N$ by the matrix $F_N \in C^{N\times N}$, given by

$$F_N = \begin{pmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \cdots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \cdots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \cdots & \omega_N^{(N-1)(N-1)} \end{pmatrix}$$

where $\omega_N = e^{i2\pi/N}$ is a primitive Nth root of unity in $\mathbb{C}$. I will use the ω_N notation throughout the text.

The corresponding scalar definition of the discrete Fourier transform is

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{kn}, \tag{2.1}$$

where $k = 1, \ldots, N$. The two definitions are equivalent, since $X = F_N x$ by the definition of matrix-vector multiplication.

Consider the following recursion:

$$F_N \Pi_N^T = B_N \begin{pmatrix} F_{\frac{N}{2}} & \mathbf{0} \\ \mathbf{0} & F_{\frac{N}{2}} \end{pmatrix} \tag{2.2}$$

where Π_N is a permutation called a *perfect shuffle* and B_N is a sparse matrix called the *butterfly matrix* (both are described below). The equation captures the intuition that the butterfly matrix relates F_N to $F_{\frac{N}{2}}$, while leaving the output permuted by the perfect shuffle.

Now contrast this with the system of scalar equations:[1]

$$X_{k_2} = \sum_{n_2=0}^{N/2-1} x_{2n_2} \omega_{N/2}^{n_2 k_2} + \omega_N^{k_2} \sum_{n_2=0}^{N/2-1} x_{2n_2+1} \omega_{N/2}^{n_2 k_2} \tag{2.3}$$

$$X_{N/2+k_2} = \sum_{n_2=0}^{N/2-1} x_{2n_2} \omega_{N/2}^{n_2 k_2} - \omega_N^{k_2} \sum_{n_2=0}^{N/2-1} x_{2n_2+1} \omega_{N/2}^{n_2 k_2} \tag{2.4}$$

While the discrete Fourier transform of size N is still related to the discrete Fourier transform of size $\frac{N}{2}$, the question of how the input and output is permuted is evaded entirely. It is not immediately obvious whether or not the input will have to be permuted and whether or not an algorithm based on the recursion will require non-constant working space.

Matrix-vector notation, on the other hand, allows for simple and elegant correctness proofs, while still being closely connected to the algorithms based on the factorizations.

2.1 Factoring the DFT Matrix

Many of the derivations use the *Kronecker product*, which is a generalisation of the outer product for vectors to matrices of arbitrary size. Like the outer product, the Kronecker product is denoted by $\otimes$.

Definition 2.1.1. Let $A \in \mathbb{C}^{n\times m}$ and $B \in \mathbb{C}^{p\times q}$ be arbitrary matrices. The *Kronecker product* of A and B is given by

$$A \otimes B = \begin{pmatrix} A_{00}B & A_{01}B & \cdots & A_{0,m-1}B \\ A_{10}B & \cdots & \cdots & A_{1,m-1}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0}B & A_{n-1,1}B & \cdots & A_{n-1,m-1}B \end{pmatrix} \in \mathbb{C}^{np\times mq}$$

[1] A derivation of the equations can be found in [B8] under “Radix-2 and Radix-4 algorithms.”

Definition 2.1.2. Define the *even-odd permutation* or *perfect shuffle* as the permutation matrix Π_N, which orders the elements of a vector x of size N so the even indices come before the odd indices:

$$\Pi_N x = (x_0, x_2, \cdots, x_1, x_3, \cdots)^T$$

Definition 2.1.3. Let $m = \frac{N}{2}$ and $\Omega_m = \operatorname{diag}\left(1, \omega_N, \omega_N^2, \ldots, \omega_N^{m-1}\right)$ be the diagonal matrix containing the first m powers of ω_N, in order. Then we define the size N radix-2 butterfly matrix B_N as the block matrix

$$B_N = \begin{pmatrix} I_m & \Omega_N \\ I_m & -\Omega_N \end{pmatrix}$$

We are now ready to show the radix-2 splitting theorem, which is of particular importance, since it is the foundation of many factorizations of the DFT matrix.

Theorem 1. *Let $N = 2m$. Then*

$$F_N \Pi_N = \begin{pmatrix} F_m & \Omega_m F_m \\ F_m & -\Omega_m F_m \end{pmatrix} = B_N \left(I_2 \otimes F_m\right)$$

Proof. The coordinate is shown by coordinate, one quadrant at a time. Let $0 \leq q < m$ and $0 \leq r < m$. Then

$$[F_N \Pi_N]_{qr} = \omega_N^{q(2r)} = \omega_m^{qr} = [F_m]_{qr}$$

$$[F_N \Pi_N]_{q,r+m} = \omega_N^{q(2(r+m))} = \omega_m^{q(r+m)} = \omega_m^{qr} = [F_m]_{qr}$$

$$[F_N \Pi_N]_{q+m,r} = \omega_N^{(q+m)(2r+1)} = \omega_N^{q} \omega_m^{qr} = [\Omega_m F_m]_{qr}$$

$$[F_N \Pi_N]_{q+m,r+m} = \omega_N^{(q+m)(2(r+m)+1)} = -\omega_N^{q} \omega_m^{qr} = [-\Omega_m F_m]_{qr}$$

□

The following lemma is useful for proving the correctness of the Cooley-Tukey factorization below:

Lemma 1. *Let $N = 2^t$, $m = \frac{N}{2}$ and $P_N = \Pi_N \left(I_2 \otimes P_m\right)$ for $N > 1$ and $P_1 = I_1$. If*

$$F_m P_m = C_{t-1} C_{t-2} \cdots C_1$$

then

$$F_n P_n = B_N \left(I_2 \otimes C_{t-1}\right) \left(I_2 \otimes C_{t-2}\right) \cdots \left(I_2 \otimes C_1\right)$$

Proof. By the radix-2 splitting theorem,

$$F_N \Pi_N = B_N \left(I_2 \otimes F_m\right)$$

and since P_m is a permutation it is orthogonal, so, by the hypothesis,

$$F_N \Pi_N = B_N \left(I_2 \otimes \left(C_{t-1} C_{t-2} \cdots C_1 P_m^T\right)\right)$$

and by the mixed-product property (where $AB = I_2 I_2 = I_2$)

$$F_N \Pi_N = B_N \left(I_2 \otimes C_{t-1}\right) \left(I_2 \otimes C_{t-2}\right) \cdots \left(I_2 \otimes C_1\right) \left(I_2 \otimes P_m^T\right)$$

Finally, since $I_2 \otimes P_m$ is a permutation it is orthogonal, so

$$\left(I_2 \otimes P_m^T\right)^{-1} = \left(I_2 \otimes P_m^T\right)^T = I_2 \otimes P_m$$

and by the definition of P_N,

$$F_N P_N = F_N \left(\Pi_N \left(I_2 \otimes P_m\right)\right) = B_N \left(I_2 \otimes C_{t-1}\right) \left(I_2 \otimes C_{t-2}\right) \cdots \left(I_2 \otimes C_1\right)$$

□

The next theorem is the factorization at the heart of the famous Cooley-Tukey algorithm:

Theorem 2. *If $N = 2^t$, then*

$$F_N = A_t \cdots A_1 P_N^T$$

where

$$A_q = I_r \otimes B_L, \quad L = 2^q, \quad r = \frac{N}{L}$$

Proof. The proof is by mathematical induction on t. Base case ($t = 1$):

$$F_N = F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = B_2 = \left(I_1 \otimes B_2\right) I_2 = A_1 P_N$$

Induction step: The induction step follows from Lemma 1, where we define C_q by $A_q = I_2 \otimes C_q$ for $0 < q < t$ and notice that $A_t = I_1 \otimes B_N = B_N$. □

The factorization directly yields the following high-level algorithm for computing the FFT:

```
function DIT Iterative(input)
    input ← $P_N^T \cdot$ input
    for $i = 1 \to t$ do
        input ← $A_i \cdot$ input
    end for
end function
```

This algorithm is the radix-2 decimation in time Cooley-Tukey FFT.

The DFT matrix can be factored in a number of ways in addition to the Cooley-Tukey factorization above. Each of those factorizations corresponds to one or more algorithms which compute the DFT.

I have stated three additional factorizations here. The correctness proofs can be found in [A5, Ch. 1]. Like the Cooley-Tukey factorization above, these factorizations only work for powers of 2 but more general versions exist.

The *Pease factorization* is somewhat similar to the Cooley-Tukey factorization, in that it requires the input to be bit-reversed:

$$F_N = H_t H_{t-1} \cdots H_1 P_N \tag{2.5}$$

Each of the factors H_q is given by:

$$H_q = \begin{bmatrix} I_{N/2} & (\Omega_L \otimes I_r) \\ I_{N/2} & -(\Omega_L \otimes I_r) \end{bmatrix}$$

where $L = 2^{q-1}$ and $r = 2^{-q}$.

The *Stockham factorization* is an example of an *auto-sorting FFT*, which means that the permutation of the input is interleaved with the computation. Each of the factors G_q is the product of a permutation and a matrix which represents a computation:

$$G_q = (B_L \otimes I_r)(\Pi_L \otimes I_r)$$

where $L = 2^q$ and $r = 2^{-q}$. This gives us the factorization:

$$F_N = G_t G_{t-1} \cdots G_1 \tag{2.6}$$

The *transposed Stockham factorization*[2] is also an auto-sorting factorization. It is given by:

$$F_N = S_t S_{t-1} \cdots S_1 \tag{2.7}$$

where

$$S_q = (I_r \otimes B_L)\left(\Pi_{2r} \otimes I_{L/2}\right)$$

$L = 2^q$ and $r = 2^{-q}$.

Since $[F_N]_{rk} = \omega_N^{rk} = \omega_N^{kr} = [F_N]_{kr}$, F_N is symmetric. In particular, this means that for every factorization of F_N there is another factorization obtained by transposition. The factorizations above are known as *decimation in time* or *DIT* factorizations. The transposed versions are known as *decimation in frequency* or *DIT* factorizations. The names are from signal analysis, where they correspond to intuitions about time space and frequency space[B6].

The DIF version of the Cooley-Tukey was discovered by Gentleman and Sande in 1966[B9]:

$$F_N = P_N^T A_1^T A_2^T \cdots A_t^T$$

Similarly, there is a DIF version of the Pease factorization:

$$F_N = H_1^T H_2^T \cdots H_t^T P_N^T \tag{2.8}$$

The Stockham factorization:

$$F_N = G_1^T G_2^T \cdots G_t^T \tag{2.9}$$

And the transposed Stockham factorization:

$$F_N = S_1^T S_2^T \cdots S_t^T \tag{2.10}$$

[2]Note that despite the name, the transposed Stockham factorization is not actually the transpose of the Stockham factorization. The transpose of the Stockham factorization is the decimation in frequency Stockham factorization

While some of the factorizations in [A5] are derived using recursive equations, the algorithms to compute those factorizations are stated in a strictly iterative manner. This may be because of the overhead associated with calling methods. Since [A5] was published, the speed of memory has become much more important when designing algorithms. This means that recursive calls may well be worth the overhead, provided that the problem instance is too large to fit in cache and the recursive calls provide better locality.

The following recursive algorithm is based directly on Theorem 1:

```
function DIT Recursive(input)
    DIT Recursive(input$_{low}$)
    DIT Recursive(input$_{high}$)
    input $\leftarrow B_N \cdot$ input
end function
```

Note that N is the size of the input at each recursive level. The notation input$_{low}$ and input$_{high}$ refers to the upper and lower half of the input vector. The subvectors are passed by reference, in order to avoid copying the data at each recursive level.

This recursive algorithm is based on the radix-2 Cooley-Tukey factorization but similar properties can be shown for other factorizations.

As mentioned in the introduction, this thesis is primarily concerned with input sizes which are powers of two. The factorizations for such input sizes are usually simpler than for other input sizes but more general factorizations do exist. All of the frameworks above can be extended to support radices other than 2 but they still rely on the input size being highly composite. Special algorithms have been devised for input sizes which are prime or almost prime. These usually rely on the fact that the DFT is intimately connected to circular convolution. A DFT of size N may be reduced to a circular convolution of size $N-1$ in constant time. This circular convolution may in turn be computed using another DFT of size $N-1$, where $N-1$ is hopefully highly composite. Otherwise the procedure may be repeated. More details may be found in [B8].

2.2 A Novel Factorization

While implementing the Cooley-Tukey FFT, I have come across a variation of that factorization which I believe to be new. The factorization is so similar that it can be derived simply from the fact that matrix multiplication is associative.

Consider the standard Cooley-Tukey factorization:

$$F_N = A_t \cdots A_1 P_N^T$$

If t is even we can write

$$F_N = A'_{t/2} \cdots A'_1 P_N^T$$

where $A'_q = A_{2q}A_{2q-1}$. For odd t the factorization is similar, with a single factor which is not paired with another. The advantage to this new factorization is that A'_q is still sparse, so a vector can be multiplied by A'_q in a single linear pass. This effectively halves the number of passes over the data, resulting in more effective use of the cache and load and store instructions, assuming the target platform has enough registers to perform each iteration without spills.

The pseudo-code for the resulting algorithm is completely analogous to the pseudo-code for the ordinary Cooley-Tukey FFT:

```
function DIT ITERATIVE'(input)
    input ← $P_N^T \cdot$ input
    for $i = 1 \to t$ do
        input ← $A_i' \cdot$ input
    end for
end function
```

It is possible to multiply by more than two factors at once. However, the resulting implementation is already very complicated to write. The register pressure also grows significantly when applying this trick, so applying it twice is likely to cause a large number of spills.

Just as with the ordinary Cooley-Tukey algorithm, it is possible to implement the factorization with a recursive algorithm. The necessary recursion can be derived by applying Theorem 1 twice, resulting in the following algorithm:

```
function DIT FOUR-WAY(input)
    DIT Four-Way(input$_{first}$)
    DIT Four-Way(input$_{second}$)
    DIT Four-Way(input$_{third}$)
    DIT Four-Way(input$_{fourth}$)
    input ← $B_N' \cdot$ input
end function
```

where $B_N' = B_N \left(I_2 \otimes B_{N/2} \right)$. The notation input$_{first}$, input$_{second}$, input$_{third}$ and input$_{fourth}$ refers to the four quarters of the input vector. Like in the normal recursive algorithm, they are passed by reference, so the data is not copied at each recursive level.

The pseudo-code given here only works for cases where t is even but more general cases can be handled easily as a special case at the bottom of the recursion tree.

2.3 Computational Complexity

The complexity of all the algorithms based on the factorizations shown in this thesis is $O(N \log N)$. To see why, notice that the number of factors is $t = \log_2 N$ (with the possible addition of a single bit-reversal permutation). Since it takes linear time to multiply by each factor, the overall complexity is $O(N \log N)$.

It also takes $O(N \log N)$ time to perform the bit-reversal permutation by performing t perfect shuffles.

2.4 The Inverse DFT

The DFT matrix is invertible, with the inverse given by[A5, p. 3]

$$F_N^{-1} = \frac{1}{N}\bar{F}_N$$

This means that the inverse DFT can be computed using the same FFT algorithms used to compute the DFT, with ω_N replaced by $\bar{\omega}_N$. The result then needs to be normalized, that is, multiplied by $\frac{1}{N}$. This can either be done during the FFT computation or as a separate pass afterward.

It is particularly interesting to note that it is possible to compute the FFT using a DIF Cooley-Tukey algorithm and then computing the inverse using the DIT version. This means that the two bit-reversal permutations can be left out entirely, assuming that whatever computation needs to be performed on the Fourier transformed data can be performed when the data is in bit-reversed order. Fortunately, this is often the case in practice. It is also the reason why I have not engineered at fast bit-reversal permutation for this thesis.

Chapter 3

Hardware

Modern computers are structured in many different ways. Graphics processors are fundamentally different from general purpose CPUs, which are again different from embedded processors.

This chapter will introduce the basic concepts of CPU architecture, with emphasis on the x64 platform and the Nehalem microarchitecture in particular.

The *clock per instruction* or *CPI* is the number of clock cycles spent executing a code sequence, divided by the number of instructions it contains. Depending on a number of factors, the CPI may range from several cycles to a fraction of a cycle, if several instructions are retired each cycle. [1]

The total CPU time of any program can be described by the following simple formula:

$$\text{CPI} \cdot \text{Instruction Count} \cdot \text{Clock Frequency} \tag{3.1}$$

This means that any speed improvement in a program must come from an increase in at least one of these three quantities, without similarly decreasing the rest.

We shall come to see that the CPI is the critical factor in improving the execution time of FFT computations.

3.1 Instruction Set Architectures

A processor's *instruction set architecture* or *ISA* describes which instructions and registers are available, as well as how they are encoded.

The two main architectural families are *reduced instruction set architectures* (*RISC*) and *complex instruction set architectures* (*CISC*).

RISC processors provide fewer, less specialized instructions, while CISC processors have more instructions which are more specialized. For instance, the x86 architecture includes an instruction, `fsin`, which computes the sine of an angle. This means that a program which needs the sine function does not need to include an implementation, which makes its binary representation shorter. However, the design and implementation of the processor becomes more complicated.

[1]It is worth noting that the definition does not refer to the number of cycles it takes to execute each instruction. We will see why below.

3.2 The Memory Hierarchy

The speed of CPUs has traditionally increased exponentially over time. Memory bandwidth and latency has also increased exponentially but unfortunately with a smaller base. This means that not only is the CPU faster than the memory subsystem, the speed gap grows exponentially over time. Latency is particularly important, since bandwidth grows with at least the square of the improvement in latency[A4, p. 18-19].

To overcome this problem, all modern CPUs include at least two levels of on-chip caches and most have three.

Hardware caches are a way to exploit the *locality principle*. Basically, programs tend to work in the same neighborhood of data, called a *working set*. The locality principle is subdivided into four classes of locality:

1. *Spatial locality*: The program references memory locations in close proximity to each other within a short period of time.

2. *Temporal locality*: The program references the same memory location within a short period of time.

3. *Branch locality*: The program tends to only have a few frequently executed code paths. Individual branches tend to go to the same location frequently and several conditional branches are often correlated in practice.

4. *Equidistant locality*: The program references equally spaced memory locations within a short time period. FFT computations often work using strided loops, which exhibit equidistant locality.

In order to take advantage of spatial locality, the cache is split into *lines*, similarly to how the data on hard disks are split into blocks. If a single byte in a cache line is needed, the entire line is read into the cache. This means that the processor will not have to access main memory again to retrieve the rest of the cache line. On the other hand, the processor may waste memory bandwidth, if it only needs part of the cache line. It is a very important goal for programmers and compilers to ensure that cache lines are utilized well.

Whenever the processor references a memory location which is not already present in the cache, it needs to be fetched from lower in the memory hierarchy. This is known as a *cache miss*. When a miss occurs, the fetched cache line needs to be stored in some location. For implementation reasons, the line may only be placed in certain specific locations. If each cache line can be placed in any location, the cache is called *fully associative*. If it can only be placed in a single location, the cache is called *direct mapped*. In between fully associative and direct mapped caches are k-way set associative caches. Such caches are split into *sets* of size k. Each memory address belongs to exactly one set. Whenever a new cache line is fetched, it evicts another line from its set.

Whenever a miss occurs, the cache needs a method to decide which line to evict. This is known as the cache's *replacement strategy*. Replacement strategies comprise a large and fascinating subject in their own right. A thorough treatment of the subject is outside the scope of this thesis.

If every cache line in L1 must be present in L2, the L2 cache is called *inclusive*. If the cache lines in L1 and L2 are disjoint, L2 is called exclusive (e.g. the AMD Athlon's cache). Some hybrid caches exist which are neither inclusive nor exclusive (e.g. Intel Pentium II, III, and 4).

Cache misses are usually divided into three categories:

1. *Compulsory misses* occur the first time the processor accesses a cache line. There is no way for the cache to contain the line without fetching it first.

   Note that this kind of miss does not necessarily affect the execution time of the program, since the processor may be able to fetch the line before an instruction actually needs the data. If there is sufficient memory bandwidth to serve the request without affecting other requests, the processor may not have to wait for the cache line at all.

2. *Capacity misses* are misses which occur when the cache is full and the processor accesses a cache line which is not present in cache. Another cache line then has to be evicted to make room for the new one.

3. *Conflict misses* are a subclass of capacity misses, which occur when the processor accesses a cache line where the corresponding entry set is already full. Since no practical caches are fully associative, all capacity misses are basically conflict misses in practice. [2]

Processors may either store instructions and data in the same *unified cache* or in a separate *instruction cache* (*i-cache*) and *data cache* (*d-cache*) (this scheme is referred to as *split caching*. The processor may also use unified caching at some cache levels, while using split caching at other levels (the Nehalem does this).

An advanced instruction cache may store instructions which have already been decoded. This means that when the processor fetches an instruction from the cache, it does not have to decode the instruction again, which may save time and/or power.

The Nehalem microarchitecture does not have a decoded i-cache, although its successors Sandy Bridge and Ivy Bridge do.

Whenever a cache miss occurs, the system has to service the memory request by fetching data from RAM. This task is performed by the *memory controller*. Traditionally, this has been a separate chip on the motherboard. In recent processors it has been integrated in the CPU in order to improve performance.

Recent microarchitectures have also started to have several *memory channels*, similarly to the discrete *memory banks* of early computers. The idea is to partition the physical memory addresses into a number of banks, which correspond to sets of physical slots in the motherboard. The memory controller is then able to access the banks in parallel, effectively multiplying the peak bandwidth by the number of memory channels. There is usually a constraint that each memory bank must contain the same number of identical RAM sticks.

[2] A number of schemes have been devised to make set-associative caches act more like fully associative caches in special cases. I do not consider these schemes.

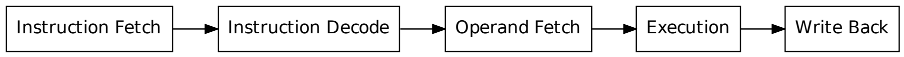


Figure 3.1: A five step RISC pipeline. This pipeline can have up to 5 instructions in flight simultaneously.

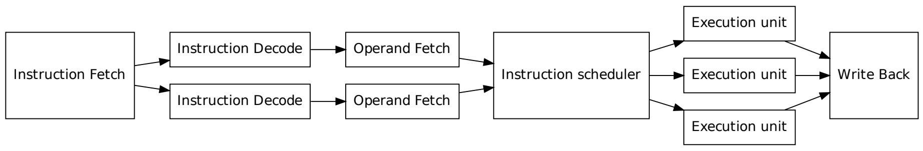


Figure 3.2: A simplified superscalar RISC pipeline. The pipeline can contain up to twice as many instructions in flight as a scalar version of the same pipeline.

3.3 Instruction-Level Parallelism

Recall the performance formula (3.1). The processor architects cannot change the instruction count in a program, so they must improve the CPI or the clock speed in order to improve overall performance. It turns out that *instruction-level parallelism* can be used to improve both.

Executing an instruction involves a number of tasks for the processor. The instruction needs to be fetched from the instruction cache or main memory, it needs to be encoded, the operands need to be fetched, etc. It turns out that it is possible to break instructions down into subtasks, which may then be implemented separately.

While it is not possible to execute a single instruction before it has been fetched from memory, it is possible to fetch the next instruction while another instruction is being executed. More generally, an instruction can be split into a number of *stages* where each stage is performed by a separate circuit, like workers on an assembly line. This idea is known as *pipelining.* A simple five step pipeline can be seen in Figure 3.1.

A pipeline can have up to one instruction *in flight* per stage. This means that under ideal conditions, the time per instruction[3] in a pipelined processor is[A4, p. C.1]:

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipeline stages}}$$

Unfortunately, it is not always possible to achieve this theoretical maximum. Sometimes an instruction cannot immediately proceed to the next stage of the pipeline. This is known as a *pipeline hazard.* Whenever a pipeline hazard occurs, the pipeline has to *stall* until the hazard has been resolved.

[3]Note the use of "time" rather than "clocks." Pipelining may improve the clock speed without changing the CPI.

There are three major classes of pipeline hazards:

1. *Structural hazards* occur when the hardware does not have the resources to advance the instruction to the next stage. This might happen when two division instructions are close to each other in the program, since divisions may take many cycles to complete in the execution unit.

2. *Data hazards* occur when the input for an instruction is not available. This may occur because the input has yet to be computed by a previous instruction or because the data was not available in cache.

3. *Control hazards* occur when the processor does not know which instruction to fetch next. This is caused by branching.

The following equation describes the speedup from pipelining in the face of hazards[A4, p. C-13]:

$$\frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

A processor's *arithmetic logic unit* (*ALU*) has separate circuits for performing operations such as addition or multiplication. These are known as *execution units*. If only one instruction is in the execution stage of the pipeline at a time, each of these circuits will be idle most of the time. In order to better utilize the execution units and provide additional parallelism, processor designers started to created *superscalar* processors. Such a processor has more than one pipeline working in parallel, as depicted in Figure 3.2. A superscalar processor might have several copies of some units (e.g. addition units), while having fewer or only one copy of other units, which are used less often (e.g. the division unit).

With superscalar processors came further problems with pipeline hazards. When more instructions are in flight, the chance that a later instruction depends on the output of an earlier one increases. However, an even later instruction might not have the same dependency, which would make it possible to execute it before some of its predecessors. This leads to the idea of *out of order execution* (*OOO execution*). The processor starts executing each instruction as soon as its inputs and a suitable execution unit is available.[4]

Since later instructions may proceed with execute while an earlier instruction is stalled, OOO execution may successfully hide a cache miss. If the later instructions also stall, other even later instructions may execute, as long as sufficient reservation stations and ROB entries are available. However, if the memory subsystem consistently fails to keep up with the processor it will eventually run out of resources, no matter how many resources it has. Because of this, a cache miss may not be a sign of a performance problem (although it probably is), while an actual memory speed problem will show up as a resource contention somewhere else in the processor.

Another consequence of OOO execution, combined with relatively slow memory, is that it may be more important to schedule instructions in order to

[4]This is true for newer processors based on Tomasulo's algorithm. Implementations based on scoreboarding are more limited.

optimize memory access, rather than optimizing the pipeline utilization. Compilers have traditionally tried to schedule instructions so dependent instructions are placed as far apart as possible. While this may improve the performance of a short code sequence, it will hurt overall performance if it is done at the expense of memory access efficiency. Unfortunately the memory access patterns of a program are global properties, which are hard for a compiler to deal with.

Since an OOO execution engine needs an execution unit of the correct type to execute an instruction (e.g. an adder to perform an addition), the *instruction mix* may limit the achievable instruction-level parallelism. In order to achieve the maximum FLOP throughput the processor typically needs a specific ratio of additions to multiplications in order to achieve max throughput. The number of loads and stores compared to arithmetic instructions may also play a role.

Whenever the control reaches a conditional branch, the processor does not know what the next instruction is until the branch has been evaluated, along with all its dependencies. To avoid stalling the pipeline, the processor tries to predict what the next instruction is. This is known as *branch prediction*. Since branch mispredictions often cause cache misses which may take many cycles to finish, it is paramount that the processor is able to predict branches most of the time. A very simple branch predictor has a misprediction rate of $0\% - 18\%$ on the SPEC89 benchmarks[B11, p. C-29].

The branch prediction may just be used to fetch the next instruction for the instruction cache. The processor may also execute the instruction before it even knows if the code path will be taken. This is known as *speculative execution*. If the prediction turns out to be wrong, the processor has to undo all the speculative work and restart the computation at the correct branch location.

Another form of instruction-level parallelism which is particularly important for FFT computations is *multimedia instructions* (also known as *SIMD instructions*, short for *Single Instruction, Multiple Data*). The idea is to have special registers with several data elements (typically 2, 4 or 8) which can be operated on in parallel by a single instruction. Typical multimedia instructions either perform arithmetic operations or shuffle the elements in a multimedia register.

While multimedia instructions are sometimes known as vector instructions, they are fundamentally different from real vector computers, which are more similar to modern GPUs. Multimedia registers are much shorter than typical vector computer registers and have a fixed length, which cannot be changed without amending the ISA. Vector computers are not covered in this thesis.

A crucial difference between multimedia instructions and real vector computers is the lack of gather/scatter instructions, which make it possible to implement algorithms with complicated memory access patterns using very few instructions.[B11, p. 329]

3.4 Performance Counters

In order to help programmers tune their programs, modern processors provide an ever increasing number of *performance counters*, which can be used to measure hardware events. Performance counters are usually implemented by special registers, which can be read and possibly reset but not written to.

Typical event counters include cache misses at each level of the cache, the number of stalls of a certain type and the number of a certain type of instructions retired at each clock cycle.

Superscalar architectures make event counters more difficult to interpret. Since the processor supports several instructions in the same pipeline stage simultaneously, the processor may be underutilized even though no pipeline stage is ever completely stalled. To help in this scenario, Intel processors support counters such as "the number of cycles with less than 2 instructions being decoded."

Another source of confusion is that certain performance counters are shared among running processes. This means that the operating system and other running programs affect the event counts. The problem may be minimized by running the program several times while performing measurements and taking the minimum count, since the count will never be lower than the count for the specific program being measured.

The program may be instrumented to query the performance counters in certain locations. This allows the programmer to measure the number of hardware events in a well-defined region of the program. Another way to use performance counters is through a *sampling profiler*. As the name suggests, sampling profilers run alongside the program and regularly sample the performance counters. This allows the profiler to attribute event counts to specific functions or even source lines. This is not possible through instrumentation, since the instrumentation would have too large an impact on the counters if it was added between every code line.

By their nature, performance counters are specific to the microarchitecture, since it does not make sense to measure L3 misses on a processor with only two layers of cache.

3.5 Heat, Power Consumption and Clock Stepping

In recent years, laptops have increasingly replaced desktop computers. Since laptops have to be able to run on battery power, this has meant that processors' power requirements have become more important when designing new processors.

There has also been a tendency that new processor generations have consumed gradually more energy over time. Because of the physical law of conservation of energy, all of the energy is eventually converted into heat, which has to be dissipated. Since processors have not grown in size, the power density and thus the heat density has increased along with the energy usage. This has eventually led to cooling being a major bottleneck in system design. Again, this makes the power requirements more important.

Many newer processors can change their clock speeds dynamically, with individual cores potentially operating at different clock speeds. If the operating system detects that the processor's temperature is high, it will attempt to increase the cooling and/or decrease the clock speed. The exact behavior depends on the availability of software controlled cooling as well as the sophistication and settings of the drivers and OS.

Conversely, some processors include a turbo mode (e.g. Intel's TurboBoost), which will allow the processor to operate at a higher than normal frequency for a short while, if it detects that it is under high load and the temperature is sufficiently low.

All of this means that adequate cooling is essential in order to ensure optimal performance. In particular, the fan coolers which are often bundled with new CPUs may not be sufficient to cool a processor running at a high sustained utilization, since most users will only run their processors at full utilization for short time intervals.

3.6 Thread-Level Parallelism

For compute-bound programs, it may be beneficial to utilize more parallelism than can be provided at the instruction level. *Thread-level parallelism* allows several independent computations to run simultaneously on the same processor. If a computation can be split into independent parts it may be possible to take advantage of this form a parallelism. However, merely being able to split the computation into separate parts does not guarantee that any speedup can be gained. The problem of limited memory bandwidth is further exacerbated, since the bandwidth requirement is increased with the number of threads. This means that if memory bandwidth is the bottleneck of a single threaded program, the running time of a multithreaded version of the same program might be as long or even worse.[5] In addition, the sequential parts of the program will not be sped up, so they limit the overall achievable speedup.

On CPUs, thread-level parallelism is primarily implemented through *simultaneous multithreading* (*SMT*) and *multi-core processors*.

Simultaneous multithreading is a natural extension of an OOO pipeline. Since such a pipeline already has several execution units operating independently, it is possible to create two separate pipelines, which share these execution resources. This means that the execution units can be utilized more fully than by a single thread. However, highly tuned programs may actually run more slowly using SMT, since such programs may already have a near-perfect resource utilization and are only hindered by the overhead caused by simultaneous multithreading.

A multi-core processor is basically a single chip which contains two or more independent processors, each with its own pipeline. The caches of such a processor may or may not be shared among the cores, but each core usually has at least one layer of dedicated cache. A multi-core processor must also contain some mechanism to ensure coherence and consistency among the individual cores.

[5]Yet again, there are exceptions to the rule. Multicore processors typically have dedicated caches for each core, so a multithreaded program will have access to more cache memory in total. This means that the bandwidth requirements may or may not scale linearly with the number of threads.

This may either be done through explicit communication among the cores or by having a shared inclusive cache (the Nehalem uses the latter approach). The details can be found in [B11, ch. 3].

While instruction-level parallelism requires little or no explicit support from the programmer, multi-threading usually requires the entire algorithm to be restructured. While a great deal of research has gone into the construction of compilers to automatically parallelize code at the thread level, no such system has ever been successful without explicit support from the programmer.

In addition, it is notoriously hard to write correct parallel programs, not to mention testing or proving that they are in deed correct.

Chapter 4

Compilers

"Will compilers ever produce code that is as good as what an expert assembly language programmer can write? The correct answer is no."
— Randall Hyde, Write Great Code, vol. 2, p. 5

A compiler's purpose is to transform programs from one form to another (not necessarily source code). This is usually done by a number of distinct phases, where each phase performs one or more transformations.

Certain transformations *must* be performed. If a compiler targets x64 CPUs, it has to transform the source language into instructions which are supported by the x64 architecture. This means that the compiler must perform the transformations regardless of how inefficient the generated code may be, assuming it cannot find a faster code sequence.

Other transformations are performed for performance purposes. In these cases, the compiler has to decide if the transformation will improve or worsen the program's efficiency. This is further complicated by the fact that "efficiency" is not a single concept. Some programs should be optimized for speed, others for memory use or low power consumption per unit of time. The goal affects which optimizations should be performed.

For some transformations the decision is easy. Consider the standard technique *constant folding*, which is performed by all modern compilers. Under normal circumstances there is no reason why the expression "$5 + 5$" should not be replaced by the constant "10". If the compiler can prove with certainty that $x = 5$ then "$x + 5$" should likewise be replaced by "10". But what if $x = 5$ only 95% of the times the expression is evaluated?

It might still make sense to replace the call

```
complex-computation(x)
```

with

```
**if** x = 5 **then**
    complex-computation(5)
**else**
    complex-computation(x)
**end if**
```

as this transformation makes additional optimizations possible for the complex(5)-case. However, the compiler is now required to not only discover that x is 5 a large fraction of the time, it has to decide whether or not the trade-off is worth it, since the transformation makes the generated program larger and possibly even slower.

Modern compilers have both sophisticated capabilities for analyzing source code (*static analysis*) as well as the ability to use information from instrumented executions (*dynamic analysis*) of the code to improve optimization (*profile-guided optimization*). Even so, there are still a great number of properties that general purpose compilers are unable to infer and utilize. Compiler engineering remains an active area of research and the performance of the generated code may vary significantly even between different releases of the same compiler.

Since the focus of this thesis is not engineering a compiler, this chapter primarily aims to describe *what* a compiler does (or does not do), not *how* it does it.

4.1 Registers

Since processors have a limited number of registers, the compiler must somehow decide which variables should reside in a register at what time. This is known as *register allocation* and is usually performed in a separate phase. In addition, some ISAs have registers which are not perfect substitutes for each other. This includes x86 and x64 processors, where some instructions must store their results in specific registers. Some operations also require an additional destination register which cannot be used to hold the result simultaneously with an unrelated value. Choosing which variable should reside in which register is known as *register assignment*.

Performing register allocation and assignment involves approximating each variable's *live range*. A variable is considered *live* during execution, if it currently has a value and that value will be read later during the same execution. The live range of a variable is the set of program locations[1] where the variable may be live during an execution. Computing live ranges accurately is undecidable in general, since the halting problem prevents us from deciding whether an execution can ever reach a specific location. Compilers can nonetheless usually compute the live ranges of local variables very accurately, using a technique known as *data-flow analysis*.[2]

Once the compiler has computed the live ranges of the variables, it constructs a graph with a node for each variable and an edge between each pair of variables which have overlapping live ranges. Allocating registers on a machine with k registers now corresponds to a k-coloring of the graph. While optimal

[1]A program location is a location in the program's static representation, that is, a specific code line or instruction. A program location may occur many times in a concrete execution of the program.

[2]Data-flow analysis is a very broad family of techniques. Computing live ranges is just one possible application.

k-coloring is NP-complete in general, the instances encountered in practice are both so simple and small that the time required to color the graph is acceptable.

If the graph can not be colored with a color for each register, the compiler must generate *spills*. The compiler spills a register by placing its value on the stack and loading it back in when it is needed again. This temporarily frees up the register, making it possible to store more local variables.

It is generally accepted that compilers perform near-perfect register allocation and assignment and they certainly do a lot better job at these tasks than even the best human assembly programmers. This means that register allocation is not directly relevant when implementing FFTs, since the programmer does not perform the task manually. The relevance is the limitation that it puts on the programmer, since it means that the programmer should generally avoid to allocate and assign registers manually. If the programmer wants to write the assembly code by hand, it may be useful to look at the assembly output from an optimizing compiler, in order to reuse the compiler's register allocation.

4.2 Instruction Selection

There are a number of factors the compiler must consider when choosing which instructions to emit for a code sequence. Some instructions take longer to execute than others and processors do not have the same number of each kind of execution unit. Worse, the optimal instruction selection is different for each microarchitecture.

The compiler needs to ensure the right mix between different kinds of instructions, in order to ensure that the execution units are fully utilized. The optimal mix depends on the number of each kind of execution unit on the target microarchitecture and the instruction latency, that is, how long the instruction will occupy the execution unit.

Sometimes the compiler can replace an expensive instruction by one or more inexpensive ones, a process known as *strength reduction*. For instance, the multiplication $a * 2$ can be replaced by the addition $a + a$. Algorithms exist for replacing multiplications and divisions by sequences of additions, subtractions and bit shifts, provided one operand is a known constant[3].

Instructions on the x64 ISA have encodings which vary in length from 1 to 17 bytes. Because of the complexity of the encoding, some instructions may require more time to decode than others. This is also a factor when choosing instructions.

Last off, there is the obvious fact that the compiler can only emit instructions it knows about. Whenever the ISA is extended the compiler writers have to add support for the new instructions in order to take advantage of them. This means that newer instructions usually do not have the same level of compiler support as older instructions do. This is especially true for multimedia instructions,

[3]Notice the word "known." The compiler has to be able to deduce what the constant is, which is usually done through data-flow analysis. If a value is constant, the program may be more efficiently compiled if it is a compile-time constant, rather than a run time parameter. For certain programs this may improve the overall running time significantly.

since they are particularly difficult to support. FFT computations contain many floating point operations, so we might expect that multimedia instructions might be particularly useful when computing FFTs. In fact, the multimedia computations for which the instructions are named are often dominated by FFT computations, which are used for both video and audio compression, among other things.

I will explain in Section 4.4 how this weakness in current compilers may be alleviated.

4.3 Instruction Scheduling

When the instructions to emit have been selected, the compiler still has to choose in which order to emit them. Out of order pipelines have made this task less critical, since instructions can be rescheduled at run time. However, the ordering of instructions still matters, since the processor will still decode the instructions in order. This means that high latency instructions (typically ones expected to cause cache misses) should be scheduled as early as possible. Also, the processor's capacity to reorder instructions will always be finite, so the resources used for reordering should not be wasted.

In order to ensure that the program semantics are preserved, the compiler must detect the dependencies among the instructions being scheduled. There is a dependency among a pair of instructions when they access the same locations (memory or registers) in such a way that they must not be reordered.

If one instruction writes to a variable read or written by another instruction, the two instructions are said to share a *data dependence*. There are three kinds of such dependencies[A1, p. 710-711]:

1. *True dependence* (read after write): The second instruction depends on the output of the first. The compiler must not reorder the instructions but the OOO engine is free to do so, provided it stores the needed value.

2. *Output dependence* (write after write): The second instruction writes to the same location as the first. This should might occur in at least two cases

   (a) The second instruction is not necessarily executed whenever the first one is, so the output of the first instruction might be needed.

   (b) A third instruction may read the output from the first write instruction before the control reaches the second write.

3. *Anti-dependence* (write after read): The second instruction overwrites the input to the first.

Note that "read after read" is not counted as a dependence. Since reads do not modify memory or registers, this is not a real dependence. The instructions can be freely reordered.[A4, p. 154]

Since an instruction cannot be executed before all of its dependencies, it used to be an important goal to schedule instructions so dependent instructions were placed as far apart as possible. On modern OOO processors, it is more important to ensure that cache misses are serviced as early as possible, even if this means that the processor has to reorder more instructions at run time, since cache misses are much more likely to dominate the running time.

4.4 Pragmas and Intrinsics

In order to achieve optimal performance, an expert programmer might resort to writing assembly code by hand. However, writing assembly by hand requires great skill and is very time consuming. In addition, assembly code is difficult to change and maintain, which might makes the resulting program slower in practice, since too much engineering time is wasted on writing the assembly, rather than performing high-level optimization.

It is also generally recognized that some areas of code generation (particularly register allocation and assignment) are best performed by compilers. This sets the stage for pragmas and intrinsics.

A compiler pragma is a way to pass information to the compiler, which is somehow considered outside of the normal semantics of the program. For instance, OpenMP, a system for writing parallel programs in a number of languages, is implemented using compiler pragmas. A for-loop in C++ may be parallelized by preceding it with the pragma annotation

```
#omp parallel for
```

which tells the compiler that the loop should be evaluated in parallel at run time. It is then the programmer's responsibility to ensure that the parallelism does not affect the semantics of the program.

Pragmas are also used to tell the compiler about program properties which are difficult or impossible for the compiler to prove based on the program semantics alone. Examples include telling the compiler that two pointers do not reference the same memory location (aliasing) or that the least significant bits of a pointer will always be zero. The first property is useful to ensure the absence of data hazards, the second property allows the compiler to assume that the data is aligned to important boundaries.

While pragmas may enable the compiler to perform more aggressive optimizations, there is no guarantee that the compiler supports all the relevant optimizations for a given algorithm. This is where intrinsics are useful.

An intrinsic (or built-in) function is a special library function or keyword with compiler support. An intrinsic often corresponds to an instruction on a specific architecture. This allows the programmer to write pseudo-assembly, where the compiler is still responsible for managing registers and the final code generation.

Even though intrinsics often correspond to specific instructions and are named after their corresponding instructions, the compiler may still choose to emit something else in the final code. In fact it must, if it needs to generate a spill. Conversely, it may also remove copying intrinsics, if it determines that two variables can share a register or that the destination register will already contain the same value as the source register. As with normal functions, the compiler only guarantees that the semantics is preserved.

Of particular interest for FFT computations are intrinsics for using multimedia instructions, since current compilers are notoriously bad at choosing these instructions optimally without explicit help from the programmer.

4.5 Transformations

While it is not necessary to know how the compiler goes about proving or deriving program properties, it may be useful to know which transformations compilers may employ. The programmer may use this knowledge to perform the same transformations by hand, relying on human intuition when proving that the necessary preconditions are met. In the following paragraphs I will describe two transformations which are directly relevant to the FFT.

Loop unrolling: A loop with k iterations may be replaced by a loop with $\left\lfloor \frac{k}{2} \right\rfloor$ iterations, where the body of the loop is duplicated. If k is odd, an extra loop body must then be added after the loop. This transformation may have several advantages. Since it contains fewer branches, there is less opportunity to mispredict branches. The total number of executed instructions is usually also lower, since the iteration variable may still only need to be incremented once per iteration.

Note that unrolling a loop should *not* generally improve instruction-level parallelism by itself. An OOO execution engine with speculation already executes the instructions from successive iterations simultaneously, provided the loop branch is predictable.

The main disadvantage of the transformation are that the generated program code becomes larger. This will mainly be a problem if that means that the program will be too large for the i-cache and thus needs more memory bandwidth. The source code also becomes harder to read if the transformation is performed manually by the programmer.

Reordering code lines: It may be possible to make the compiler create a better instruction schedule simply by reordering lines in the source code. As described in Section 4.3, the compiler might simply try to ensure that dependent instructions are placed far apart and then not reorder the instructions further. It also might not be able to prove that two lines are in fact independent, when the programmer might know this to be the case. Just like loop unrolling, this transformation may make the code harder to read.

Part II

Practice

Chapter 5

Implementations and Engineering

When tuning a program, the process usually starts by locating the hot-spot where most of the time is spent. The Cooley-Tukey FFT is unique in this regard. There can be no doubt that most of the time is spent in the innermost loop. The question is how the time is spent and how to improve the implementation.

A very important but often overlooked tuning step is to experiment with different compiler flags and make sure that the compiler emits the best possible code for the target platform. Even if the compiler has a specific flag to target the specific microarchitecture, this flag may not be sufficient and it may even be harmful in practice. During the development, it turned out that a single flag in particular would have a large impact.

My earliest exploratory experiments told me that the complex arithmetic was being performed in a very inefficient manner. I had built my implementations on C++'s built-in complex type, `std::complex`.

GNU's implementation of `std::complex` (version 4.4.3) is based on compiler intrinsics, which implement complex numbers. These intrinsics must guarantee sound behavior in the edge cases for complex numbers. This includes NaNs, +/- infinity and denormal floating point numbers. A quick look at the generated code suggests that this soundness guarantee may be very expensive, since it involves a number of checks. Luckily, they can be switched off with the compiler flag `-ffast-math`.

Figure 5.1 shows how the wallclock execution time has improved significantly – roughly by a factor of two. This begs the question: Why?

As can be seen in Figure 5.2, the CPI was within the same range before and after the flag was added. The CPI for the version with `-ffast-math` added was higher for small inputs and smaller for large inputs, suggesting that the implementation is more sensitive to cache effects. Figure 5.3 shows us that the instruction count is significantly lower for the version with `-ffast-math`, which is the reason for the speedup. The extra instructions in the slow version dominate the running time, which also makes the program less sensitive to cache effects, since each complex operation only uses a small constant amount of data, while still involving many instructions.

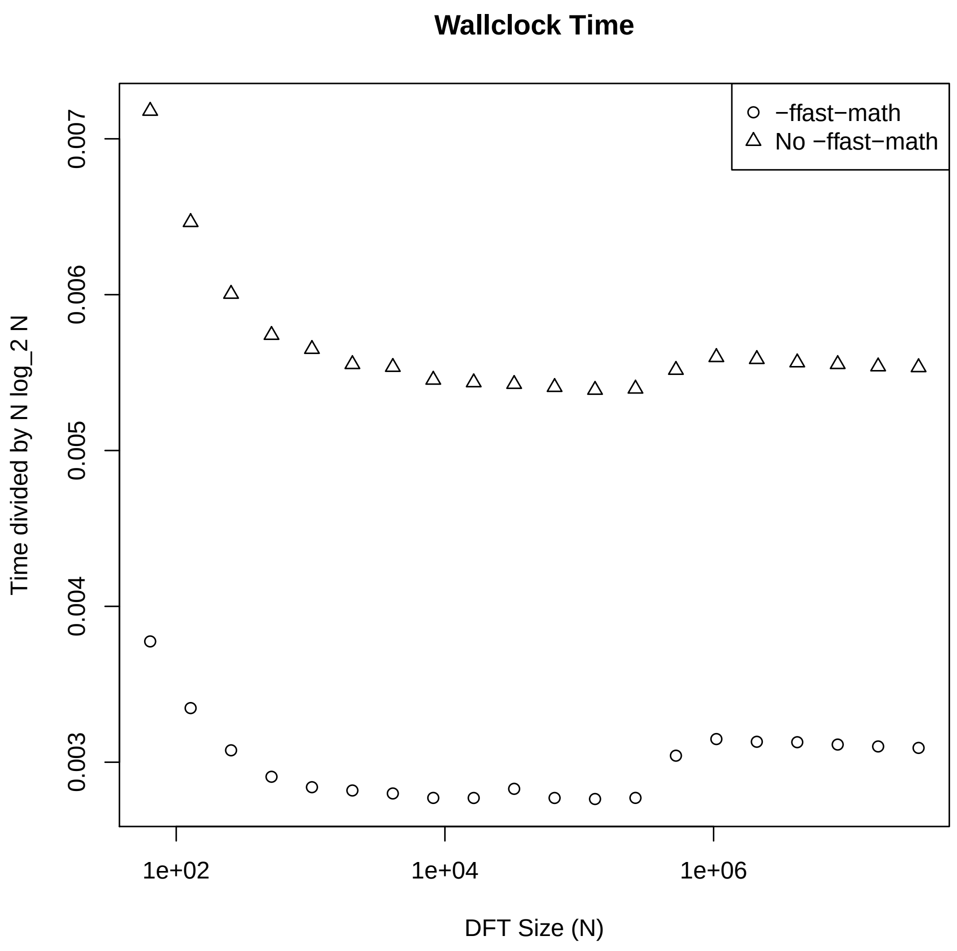


Figure 5.1: The CPI of the first iterative implementation of DIT Cooley-Tukey, before and after the `-ffast-math` compiler flag was set. The flag makes the program significantly faster for all input sizes. The plot uses a log scale on the x-axis.

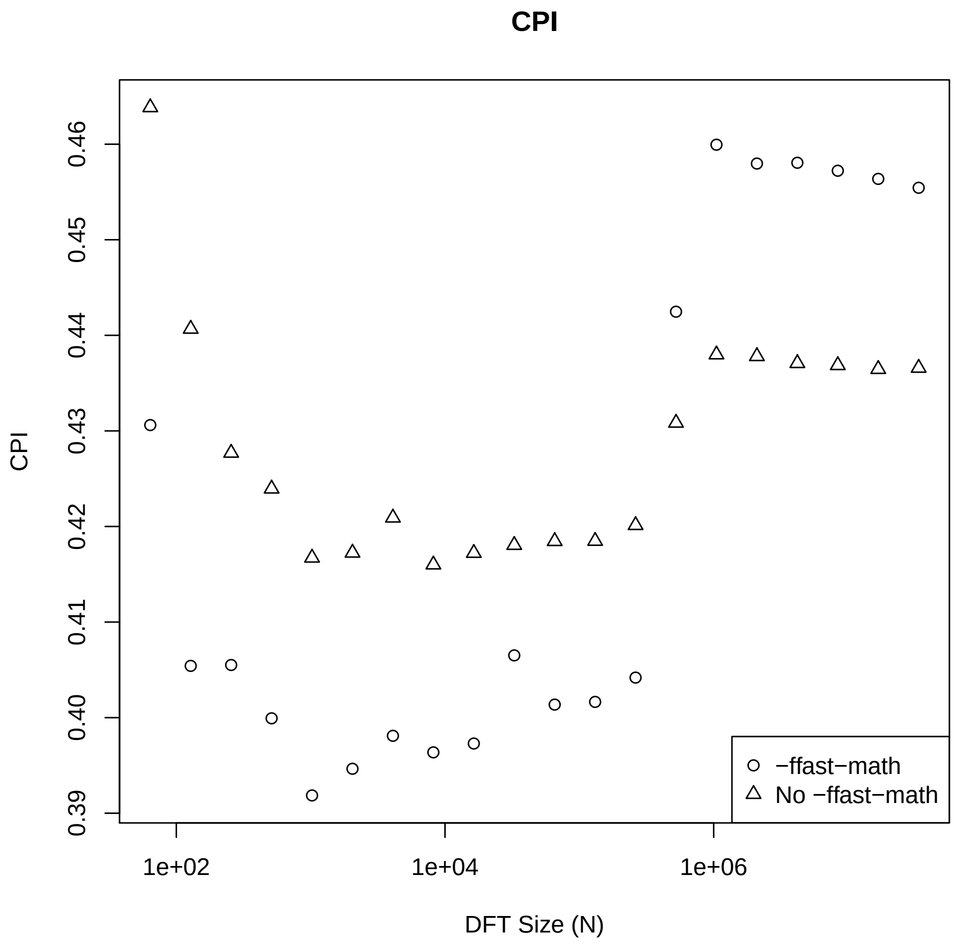


Figure 5.2: The CPI of the first iterative implementation of DIT Cooley-Tukey, before and after the `-ffast-math` compiler flag was set. The CPI is clearly within the same range for both versions, which means that the flag did not significantly affect the quality of the emitted code sequence. The plot uses a log scale on the x-axis.

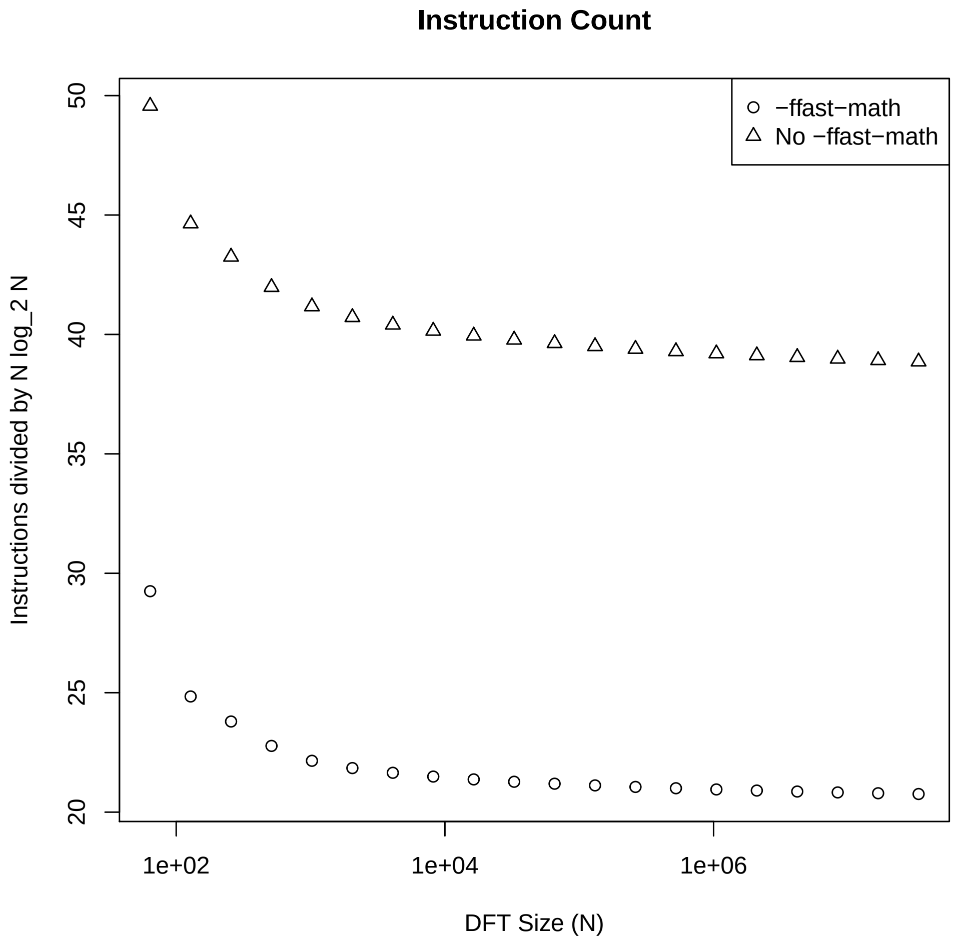


Figure 5.3: The total number of instructions retired for the first iterative implementation of DIT Cooley-Tukey, before and after the `-ffast-math` compiler flag was set. The number of instructions retired has gone down significantly after the flag was added. The plot uses a log scale on the x-axis.

5.1 Implementations

The following paragraphs describe the different implementations which are covered in the thesis. Most of the implementations exist in both at DIT and a DIF version and all of the described optimizations can be applied analogously in the two versions. The experimental results were essentially identical for the DIT and DIF versions, apart from the fact that the DIT version were always a bit slower. Since the discussion already requires a large number of plots, the results for the DIT implementations have been omitted.

All of the implementations work on double-precision floating points numbers. A pair of such numbers is used to represent each complex number, one to represent the real part and one to represent the imaginary part. Figure 5.4 shows how the different implementations are related to each other.

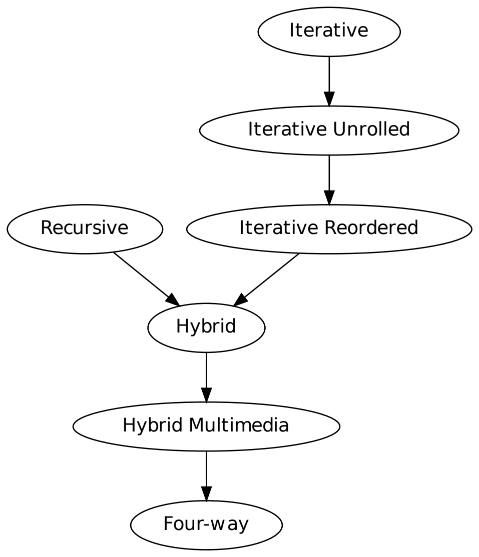


Figure 5.4: The tree of implementations, showing how each is related to the others. The edges signify that the destination was derived from the origin, using the optimizations described in the text.

Iterative A simple iterative implementation of the Cooley-Tukey or Gentleman-Sande factorization. The implementation corresponds to the pseudo-code for `DIT Iterative` and `DIF Iterative` given in Section 2.1.

Figure 5.5 shows how the multiplication $x \leftarrow A_q^T x$ is performed.

Iterative Unrolled The iterative implementation with the inner loop unrolled once. Since the innermost loop only has a single iteration for the last multiplication ($x \leftarrow A_1^T x$), the inner loop cannot be unrolled in this case. The overall structure of the algorithm becomes

```
for (int offset = 0; N > offset; offset += step) {
    field prod(1);
    for (int ii = 0; step / 2 > ii; ++ii) {
        field a = data[ii + offset];
        field b = data[ii + step / 2 + offset];
        data[ii + offset] = a + b;
        data[ii + step / 2 + offset] = prod * (a - b);
        prod *= root;
    }
}
```

Figure 5.5: The two inner loops of the iterative Gentleman-Sande (DIF Cooley-Tukey) implementation. The variable `step` contains 2^{t-q+1} and `root` contains $\omega_N^{2^q}$.

```
function DIF Iterative Unrolled(input)
    for $i = t$ downto 2 do
        input $\leftarrow A_i^T \cdot$ input
    end for
    input $\leftarrow A_1^T \cdot$ input
    input $\leftarrow P_N^T \cdot$ input
end function
```

The multiplications are then performed using the code in Figure 5.6 when $i > 1$ and using the code in Figure 5.5 when $i = 1$.

Iterative Unrolled and Reordered The iterative implementation with the inner loop unrolled once and the statements in the loop reordered to improve memory bandwidth utilization. All of the data loads have been placed at the very beginning of the loop iteration, in order to start servicing cache misses as early as possible. Figure 5.7 shows the reordered code.

Recursive A simple recursive implementation based on Theorem 1, as described in Section 2.1 (`DIT Recursive`).

As we will see in Section 7, the implementation turned out to be slower than the slowest iterative version due to a higher instruction count.

Hybrid This version recurses until the problem instance is small enough to fit in cache, after which it solves the instance iteratively. The following pseudocode shows how:

```
for (int offset = 0; N > offset; offset += step) {
    field prod(1);
    for (int ii = 0; step / 2 > ii; ++ii) {
        {
            field a = data[ii + offset];
            field b = data[ii + step / 2 + offset];
            data[ii + offset] = a + b;
            data[ii + step / 2 + offset] = prod * (a - b);
            prod *= root;
        }
        ++ii;
        {
            field a = data[ii + offset];
            field b = data[ii + step / 2 + offset];
            data[ii + offset] = a + b;
            data[ii + step / 2 + offset] = prod * (a - b);
            prod *= root;
        }
    }
}
```

Figure 5.6: The unrolled version of the inner loop of the iterative Gentleman-Sande (DIF Cooley-Tukey) implementation.

```
for (int offset = 0; N > offset; offset += step) {
    field prod(1);
    for (int ii = 0; step / 2 > ii; ++ii) {
        field a = data[ii + offset];
        field b = data[ii + step / 2 + offset];
        field c = data[ii + 1 + offset];
        field d = data[ii + 1 + step / 2 + offset];
        data[ii + offset] = a + b;
        data[ii + step / 2 + offset] = prod * (a - b);
        data[ii + 1 + offset] = c + d;
        prod *= root;
        data[ii + 1 + step / 2 + offset] = prod * (c - d);
        prod *= root;
    }
}
```

Figure 5.7: The unrolled and reordered version of the inner loop of the iterative Gentleman-Sande (DIF Cooley-Tukey) implementation.

```
function DIT Recursive(input)
    if $N >$ cutoff then
        DIT Recursive(input$_{low}$)
        DIT Recursive(input$_{high}$)
    else
        DIT Iterative(input$_{low}$)
        DIT Iterative(input$_{high}$)
    end if
    input $\leftarrow B_N \cdot$ input
end function
```

Hybrid with multimedia instructions The hybrid implementation with `std:complex` replaced by multimedia instruction intrinsics to improve the floating point performance. The code lines were not reordered by hand, so the change was limited to:

1. Changing the types of the variable definitions to multimedia types.
2. Inlining the definitions of all complex operations.
3. Replacing the operations by multimedia intrinsics.

The intrinsics were for the SSE3 instruction set extension and the exact instructions were chosen as recommended in [B12, p. 6-17].

This algorithm was only implemented in a DIF version, since the use of multimedia intrinsics made the implementation much more time consuming to debug.

Four-way recursion This version is an implementation of the novel four-way recursion described in Section 2.2. It is a hybrid recursive/iterative implementation and uses compiler intrinsics for multimedia instructions. It uses the four-way recursion at the first highest levels of the recursion, the ordinary recursion at the middle levels and the ordinary iterative Cooley-Tukey algorithm as the base case.

This algorithm was also only implemented in the DIF version. It is worth noting that the inner loop of this implementation took more time to write (and especially debug) than all the other implementations combined.

Chapter 6

Experiments

The main engineering goal for the FFT is to make it run faster. That is, to reduce its total running time. This is known as the *wallclock time*. The wallclock time is distinguished from *processor time* in that processor time is the total amount of time the program has been running on any core. This may be less than the wallclock time if the program has not been running continuously on the processor or it may be greater, if the program runs on more than one core simultaneously.

It is also possible to distinguish between *user time* and *system time*. These times are when the program is running in user mode versus kernel mode, respectively. I will only consider the combined wallclock time, since the implementations spend a negligible amount of time in kernel mode.

When the processor's clock speed is held constant, the wallclock time is proportional to the number of elapsed clock cycles. However, counting elapsed cycles rather than wallclock is slightly misleading, since the computation might not take the same number of cycles at a slower clock speed, since the memory subsystem would be made relatively faster.

In addition to comparing the overall performance of different implementations, it is relevant to analyze why they perform as they do. Recalling the performance equation (Equation 3.1), we need to measure the *instruction count* (the total number of instructions retired) in addition to the wallclock time. We already know the clock frequency (2800MHz), since it is held constant.

The CPI can be calculated using the instruction count and either the wallclock time or the number of cycles. Plotting the instruction count tells us how much work the processor is doing, while the CPI tells us how quickly it does the work.

While the instruction count is uniquely determined by the instructions selected, the CPI is determined by a number of factors as described in Chapter 3. In order to find out why the processor might be retiring instructions quickly or slowly, I measured the total number of branch mispredictions for each implementation as well as the total number of cache-misses at each level of the cache.

6.1 Measurement Tools

Since each FFT computation is completed very quickly for small inputs, it is difficult to make accurate measurements on a single such computation. I performed 2^{25-t} computations for each $N = 2^t$, so the total running time of each experiment would be large enough to generate meaningful results.

PAPI The *Performance Application Programming Interface*, more commonly known as *PAPI* is a cross-platform API for accessing performance counters. Having such an API is useful for several reasons. It provides a coherent way to access counters across platforms, although deep knowledge of the concrete platform is still necessary. In addition, it may be necessary to have operating system support and/or to use special assembly instructions in order to access the counters. PAPI simplifies the measurements by hiding this complexity from the programmer.

PAPI was used to measure the instruction count, the total number of clock cycles, the number of branch mispredictions, and the number of cache misses at each level of the cache.

Wallclock time The wallclock time was measured using the system call `gettimeofday`. This timer has a precision of a single microsecond, although the accuracy is significantly lower in practice.

As described above, the wallclock time could in principle be derived from the number of clock cycles. The problem with this approach is that it does not consider the overhead from the operating system and system processes, which affect the wallclock time of the program in practice. On the other hand, it is simpler to consider the CPI of the program without this overhead, since the overhead is assumed to be the same for different implementations.

6.2 The Test Machine

The code was built using the GNU C++ compiler, g++, version 4.6.3. The experiments were performed on Ubuntu Linux version 12.04LTS, running in single-user mode.

The experiments were run on a Core i7 930 processor. The 930 version is very similar to the 920, which is used as a recurring example in [A4]. The only difference is that the 930 has a maximum clock speed of 2800MHz.

The processor has four cores, each with a private L1 and L2 cache and the ability to run two threads simultaneously. The L3 cache is shared among the cores. Each L1 cache is split, with 32KB for data and 32KB for instructions. Each L2 cache is unified with a 256KB capacity. The L3 cache is 8192KB. All levels of the cache use 64 byte lines.

The processor pipeline is an OOO engine with speculation, using an extended version of "Tomasulo's algorithm." The details are described in [A4, ch. 3].

The compiler was called with the following flags:

```
-O3 -Wall -ffast-math -fipa-pta -floop-interchange
-floop-strip-mine -floop-block -ftree-loop-distribution -fivopts
--fstrict-aliasing -flto --param l1-cache-line-size=64 --param
-l1-cache-size=2 --param l2-cache-size=256 -march=corei7 -lpapi
```

6.3 Plots

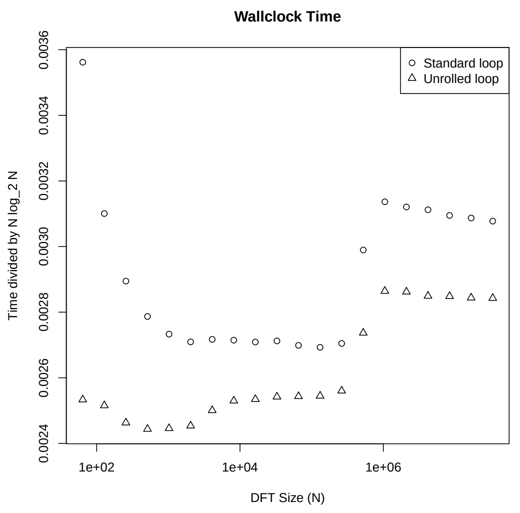
Based on the data collected from the experiments, a number of plots have been prepared for each implementation. Each plot compares the implementation being considered with its predecessor(s) (see Figure 5.4).

The effects of each performance improvement are first analyzed in terms of the the performance equation (Equation 3.1). For this purpose, a figure containing three plots has been inserted for each implementation (see Figure 6.1). The first plot shows the wallclock time, compared to the wallclock time of the predecessor(s). The second plots shows the instruction count, while the third plots shows the CPI.

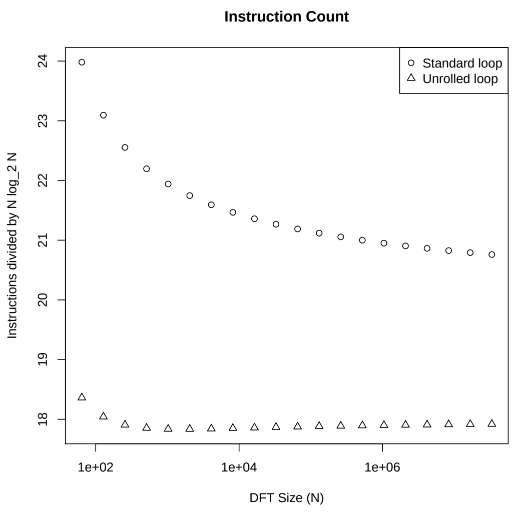
If the wallclock time and the instruction count have both improved and the CPI is unchanged or has worsened, the cause of the speed improvement can only be the decrease in the instruction count. On the other hand, if the wallclock time has decreased because of a decrease in CPI, more information is necessary to determine why the CPI has gone down.

There are three main causes for a high CPI: A large number of branch mispredictions, a large number cache misses at one or more levels of the cache, and finally inefficient use of the execution pipeline.

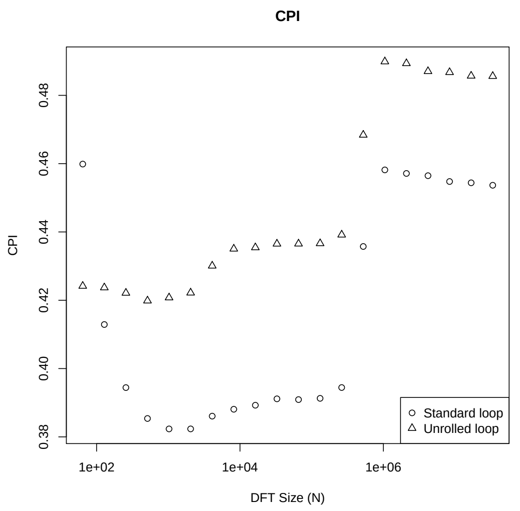
For the implementations for which wallclock time, instruction count and CPI were insufficient to explain the change in running time, four extra plots have been added: The total number of mispredicted branches, and the total number of cache misses at each level of the cache. The cache misses include both instruction and data cache misses.



(a) The wallclock time, divided by $N \log_2(N)$.

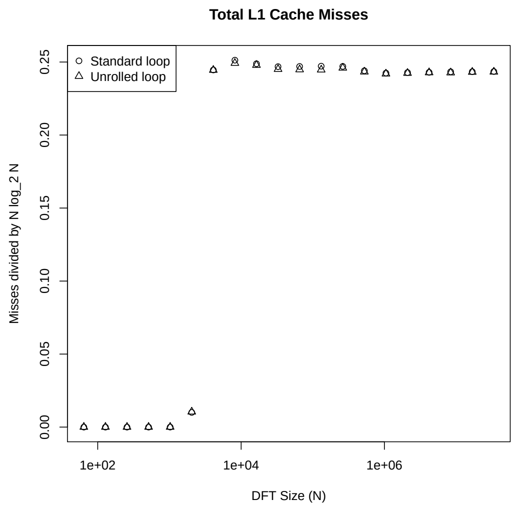


(b) The total number of instructions retired, divided by $N \log_2(N)$.

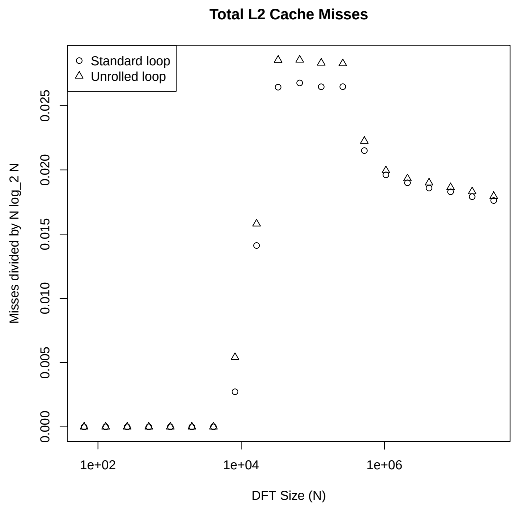


(c) The average clock per instruction for each of the implementations.

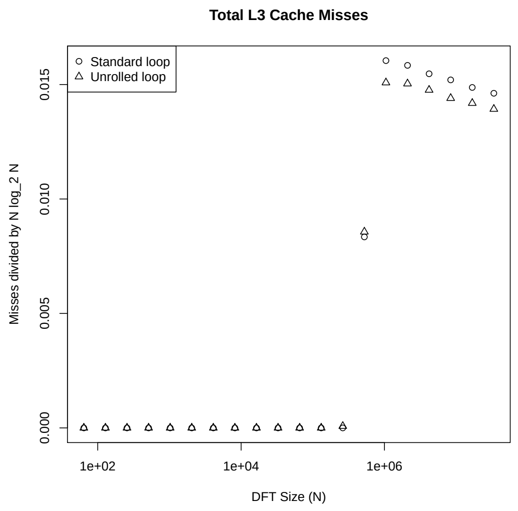
Figure 6.1: Plots comparing the simple iterative implementation with the version where the inner loop has been unrolled.



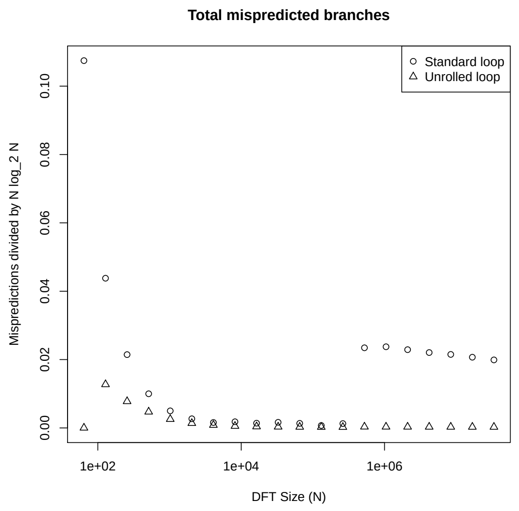
(a) The total number of L1 misses, divided by $N\log_2(N)$.



(b) The total number of L2 misses, divided by $N\log_2(N)$.

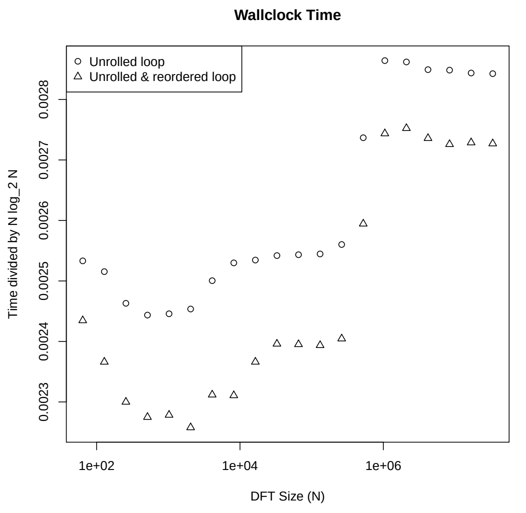


(c) The total number of L3 misses, divided by $N\log_2(N)$.

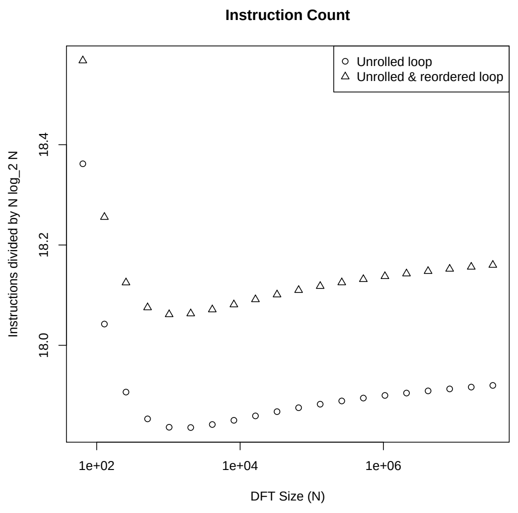


(d) The total number of branch mispredictions, divided by $N\log_2(N)$.

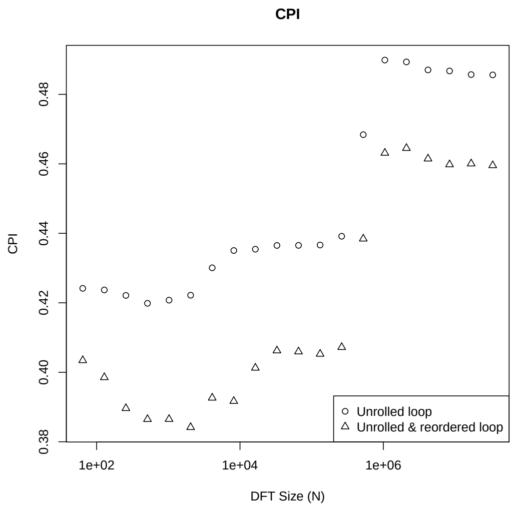
Figure 6.2: More plots comparing the simple iterative implementation with the version where the inner loop has been unrolled.



(a) The wallclock time, divided by $N \log_2(N)$.

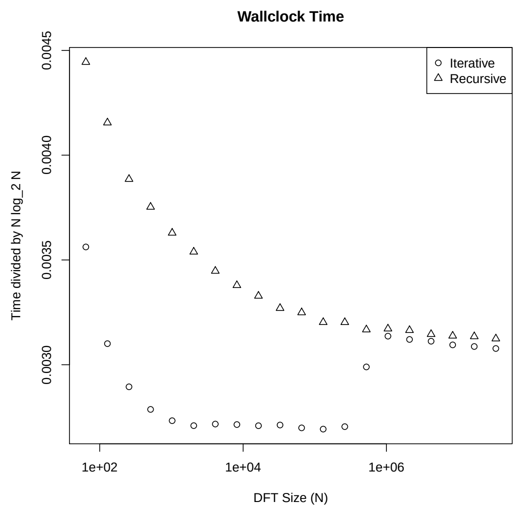


(b) The total number of instructions retired, divided by $N \log_2(N)$.

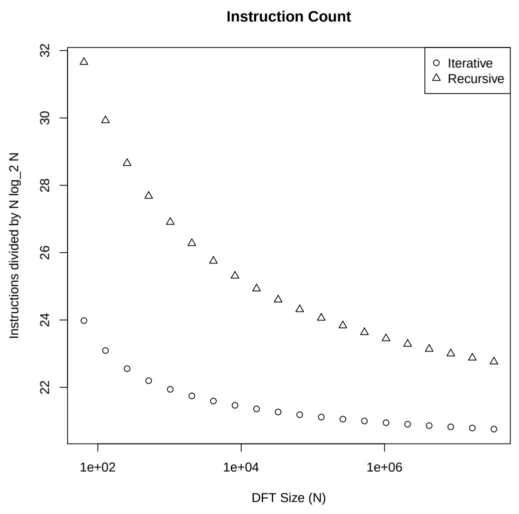


(c) The average clock per instruction for each of the implementations.

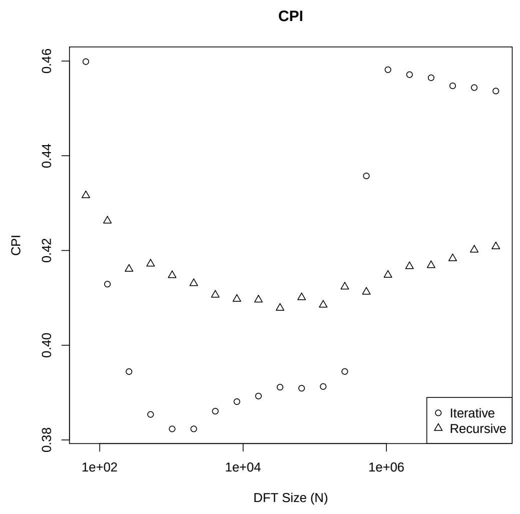
Figure 6.3: Plots comparing the iterative implementation with the inner loop has been unrolled with the implementation where the loop statements have also been reordered.



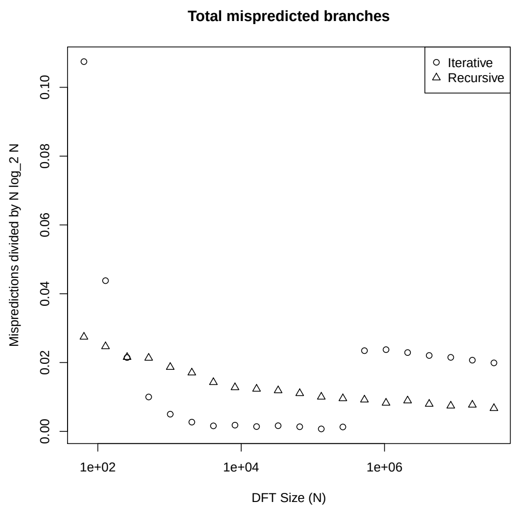
(a) The wallclock time, divided by $N \log_2 (N)$.



(b) The total number of instructions retired, divided by $N \log_2 (N)$.

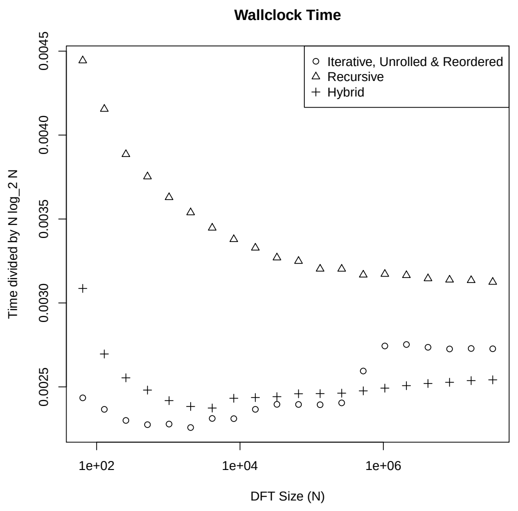


(c) The average clock per instruction for each of the implementations.

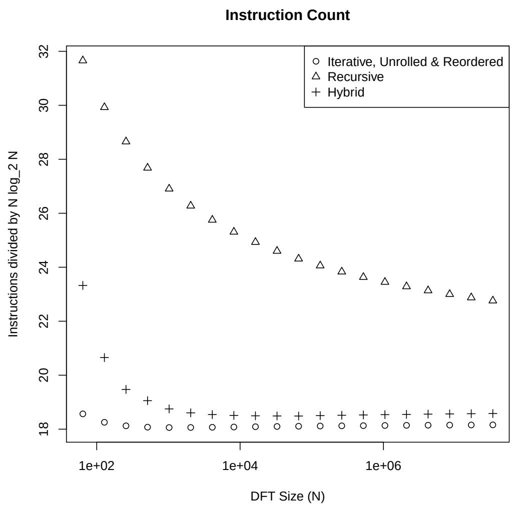


(d) The total number of branch mispredictions, divided by $N \log_2 (N)$.

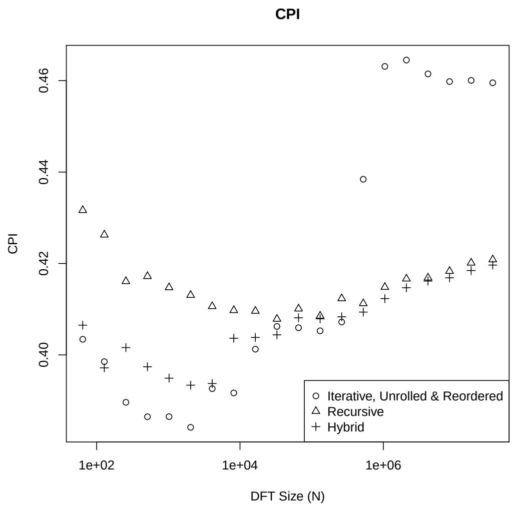
Figure 6.4: Plots comparing the first iterative implementation with the recursive implementation.



(a) The wallclock time, divided by $N \log_2(N)$.

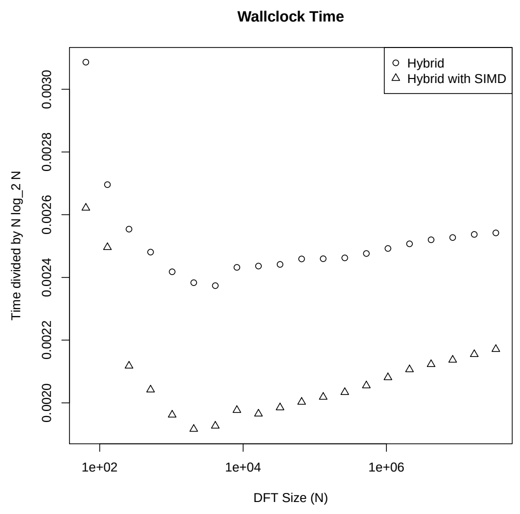


(b) The total number of instructions retired, divided by $N \log_2(N)$.

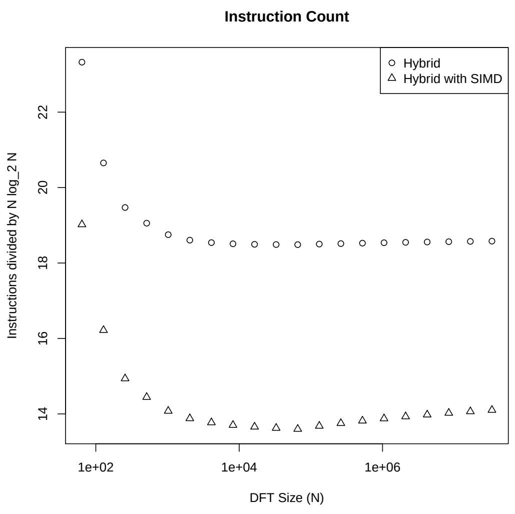


(c) The average clock per instruction for each of the implementations.

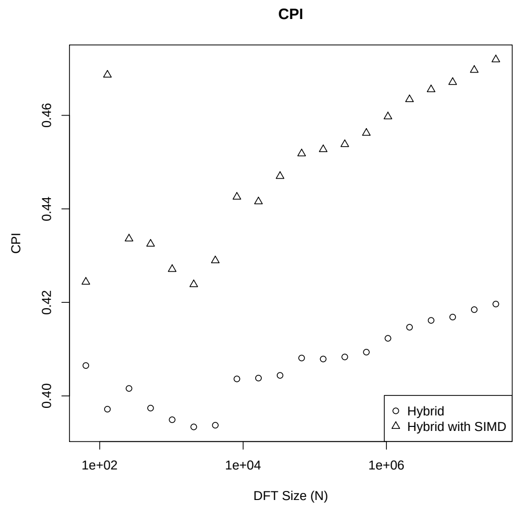
Figure 6.5: Plots comparing the first iterative implementation with unrolled inner loop and reordered loop statements, the recursive implementation and the iterative/recursive hybrid.



(a) The wallclock time, divided by $N \log_2(N)$.

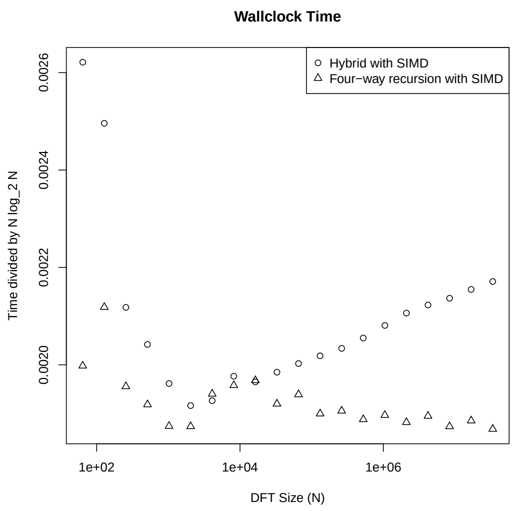


(b) The total number of instructions retired, divided by $N \log_2(N)$.

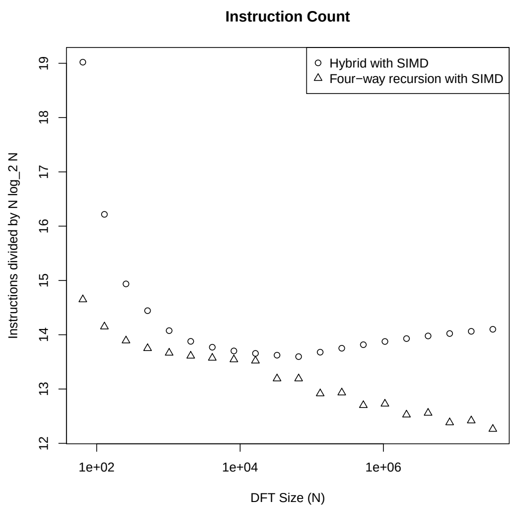


(c) The average clock per instruction for each of the implementations.

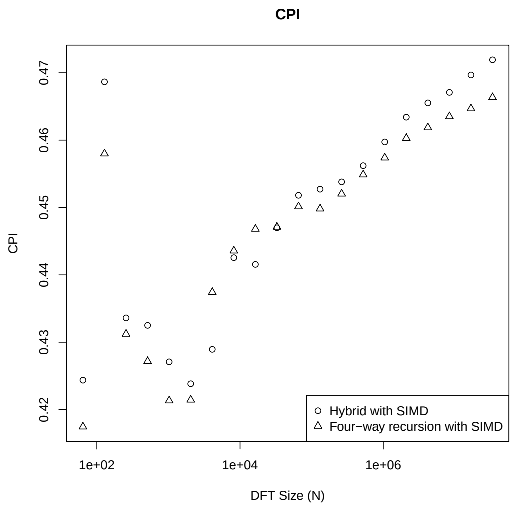
Figure 6.6: Plots comparing the hybrid implementation with the hybrid implementation with multimedia intrinsics.



(a) The wallclock time, divided by $N \log_2(N)$.

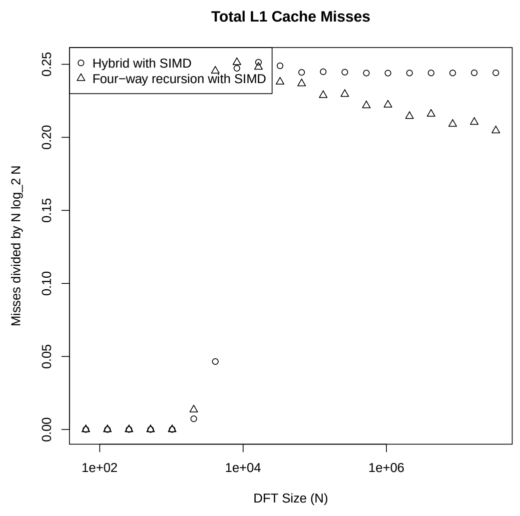


(b) The total number of instructions retired, divided by $N \log_2(N)$.

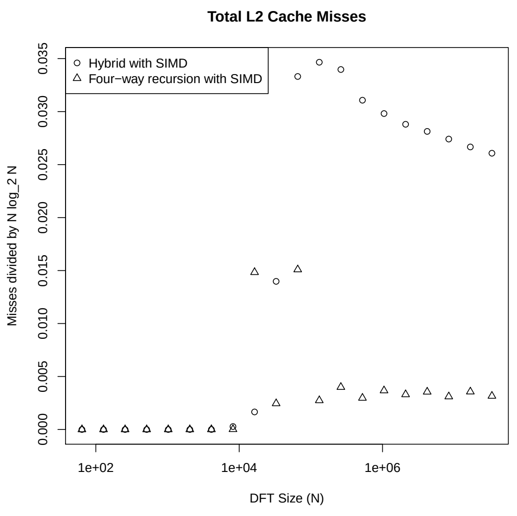


(c) The average clock per instruction for each of the implementations.

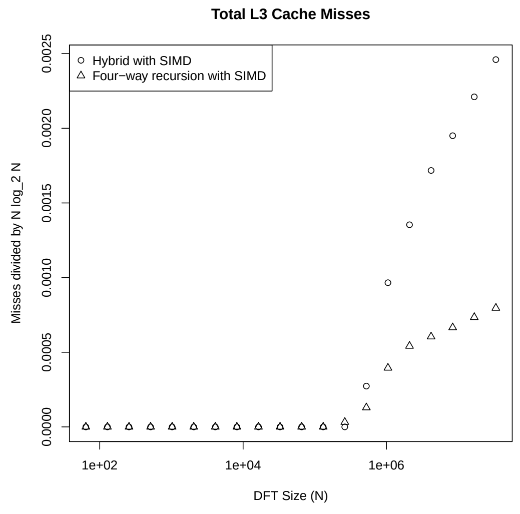
Figure 6.7: Plots comparing the hybrid implementation with multimedia intrinsics with the four-way recursive hybrid implementation.



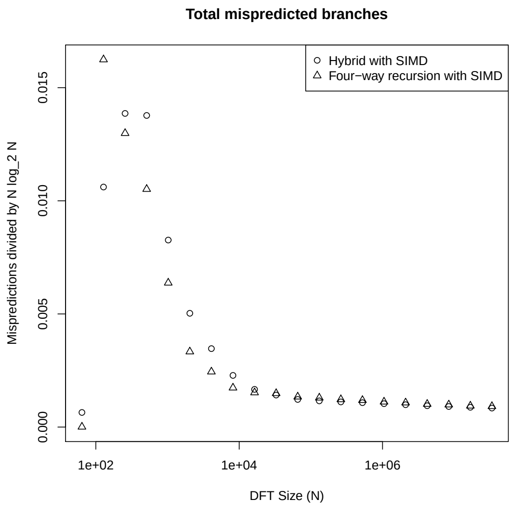
(a) The total number of L1 misses, divided by $N \log_2(N)$.



(b) The total number of L2 misses, divided by $N \log_2(N)$.



(c) The total number of L3 misses, divided by $N \log_2(N)$.



(d) The total number of branch mispredictions, divided by $N \log_2(N)$.

Figure 6.8: More plots comparing the hybrid implementation with multimedia intrinsics with the four-way recursive hybrid implementation.

Chapter 7

Analysis

Iterative Since this was the first implementation, it has no predecessor with which to compare it. Figure 6.1 shows the implementation compared to its successor, where the inner loop has been unrolled.

The implementation slows down (subfigure (a)) when the input becomes too large for the level 3 cache (Figure 6.2c). The branch-predictor starts to perform poorly at the same time, most likely due to some internal cache being full. The CPI goes up as a result.

Iterative Unrolled Figure 6.1 (a) shows that unrolling the inner loop always results in a faster wallclock time. The number of executed instructions as well as the number of branch mispredictions are also consistently lower (see Figure 6.2). Alas, the CPI is higher for all inputs but the very smallest one. This means that the processor pipeline is not utilized as well as in the first iterative version but the lower instruction count more than makes up for it.

Iterative Unrolled and Reordered This version was faster than its predecessor for all inputs (see Figure 6.3). Not only was the CPI consistently lower, the instruction count was too. The compiler apparently emits an entirely different code sequence for the reordered code lines, instead of merely reordering the same instruction sequence.

Recursive The completely recursive implementation always has the worst wallclock time of all the implementations. Figure 6.4 shows the recursive implementation compared to the simplest iterative implementation. The high wallclock time is clearly due to the high instruction count, since the recursive implementation is slower even for the largest cases, for which the cache efficiency of the recursive implementation causes it to have a lower CPI.

On the other hand, it is clear that the better locality does have a large positive effect for large cases, even if it is not sufficient to make up for the high instruction count.

It is interesting to note that the number of branch-mispredictions suddenly jumps when the input size crosses a certain threshold (subfigure (d)). This is due to a detail of the Nehalem microarchitecture. The processor uses a 16 entry special purpose cache to predict the target addresses of the `return` instruction, which is used for the return branches of function calls. These branches are predicted perfectly when the recursion is shallow enough and never predicted when the recursion becomes too deep for the cache.

Hybrid Figure 6.5 compares the hybrid version to its two predecessors. Subfigure (b) shows that the instruction count is in between that of its two predecessors.

Like the recursive implementation, the CPI is significantly lower than for the recursive implementations on large inputs. The resulting wallclock time is lower than for both its predecessors, since the instruction count is still lower than that of the recursive implementation.

On small instances it turns out to not only have a higher instruction count than its iterative predecessor, it also has a higher CPI. This leads to a significantly higher wallclock time.

The algorithm is implemented in such a way that the program always recurses once before calling the base case, even if the entire problem instance is small enough to be handled by the base case. This is probably the main reason for the bad performance for the small cases, since it raises the instruction count needlessly. This problem could be fixed relatively easy in a new version of the implementation.

Hybrid with multimedia instructions Figure 6.6 compares the hybrid versions with and without multimedia intrinsics. Using multimedia intrinsics made the implementation significantly faster on all inputs, even though the CPI is higher. This is due to the great savings in the instruction count, which come from being able to load, store or operate on a whole complex number at a time.

This implementation still has the problem of never calling the base case directly. It should be possible to fix it in the same manner as described above.

Four-way recursion Figure 6.7 compares the hybrid version with multimedia intrinsics with the four-way recursive hybrid. This implementation is the fastest for all the larger cases, in terms of wallclock time. It also has the lowest instruction count, again excepting the smallest three cases. The CPI is among the highest for all the implementations, so the lower instruction count must be the reason for the low wallclock time.

In this implementation the problem of not calling the base case directly for small instances has also been fixed. The result is a significantly lower wallclock time as well as instruction count for the smaller instances.

It is interesting to note that the CPI is relatively lower for the larger cases. Figure 6.8 shows that this is almost certainly due to more efficient use of the cache, because of the fewer passes. The number of branch mispredictions is virtually identical between the two implementations for large cases, while the number of L2 and L3 is significantly lower for the four-way version.

Chapter 8

Conclusion

This thesis seeks to answer whether or not the efficiency of FFTs can be improved significantly by using mid-low level techniques. Based on the experimental results, this question can be answered in the affirmative. A skilled programmer can use knowledge of the microarchitecture and the compiler to make the compiler generate significantly faster code than the code the compiler originally emitted.

In addition, the focus on efficient implementations led to the discovery of a new factorization of the DFT matrix. This factorization might not have been discovered otherwise, since it is completely equivalent to the known Cooley-Tukey or Gentleman-Sande factorization in terms of the usual metrics (FLOPS, complex multiplications, or complex additions).

Another way to view the results is that the compiler does not emit an optimal code-sequence without significant help from the programmer. In fact, it may not be possible to get the compiler to emit an optimal code sequence at all, without rewriting the assembly code by hand.

8.1 Future Work

This thesis only considers algorithms based on the Cooley-Tukey and Gentleman-Sande factorizations. It is quite possible that the same optimization techniques applied to a different factorization might yield an even faster implementation.

It might also be possible to improve the existing implementations by instructing the compiler to emit assembly code instead of an object file, and then editing the assembly by hand. This would allow the programmer to take advantage of the compiler's register allocation and -selection, while retaining full control of the instruction selection and scheduling. Intel's optimization manual[B12] contains a list of specific low-level optimization guidelines for how to achieve an optimal code sequence for specific microarchitectures. Inspecting the assembly generated by the compiler for the existing implementations reveals that the code sequence does not generally follow these suggestions. This leaves room for the programmer to improve the code sequence.

Part III

Appendices

Appendix A

Auxiliary Proofs

A.1 The Kronecker Product

Lemma 2. *The Kronecker product has the following properties:*

1. *If A, B, C and D are matrices, then $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$, assuming that the ordinary multiplications AC and BD are defined. This property is known as the* mixed-product property*, since it mixes the Kronecker product with the ordinary matrix multiplication.*

2. *If A is a matrix, then $I_p \otimes (I_q \otimes A) = I_{pq} \otimes A$.*

Proof. 1. This proof is a slightly more general version of the proof given in [A5, p. 8]. We write $U = A \otimes B$, $V = C \otimes D$ and $W = UV$. Let U_{kj}, V_{kj}, and W_{kj} be the kj blocks of these matrices and n be the width of A and the height of C. Then

$$W_{kj} = \sum_{q=0}^{n-1} U_{kq} V_{qj} = \sum_{q=0}^{n-1} (a_{kq} B)(c_{qj} D) = \left(\sum_{q=0}^{n-1} a_{kq} c_{qj} \right) BD = [AC]_{kj} BD$$

which yields the property.

2. The statement is a corollary of property 1.

□

Primary Bibliography (Pensum)

[A1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, chapter 10: Instruction Level Parallelism. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[A2] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art, sections 1, 2, 3, and 4. *Signal Process.*, 19(4):259–299, April 1990.

[A3] Matteo Frigo, Steven, and G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, pages 216–231, 2005.

[A4] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*, chapter 1, 3, and Appendix C. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[A5] Charles Van Loan. *Computational frameworks for the fast Fourier transform*, chapter 1. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

Secondary Bibliography

[B6] W.T. Cochran, James W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, Jr. Maling, G.C., D.E. Nelson, C.M. Rader, and Peter D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967.

[B7] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[B8] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Process.*, 19(4):259–299, April 1990.

[B9] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. ACM.

[B10] M. Heideman, D.H. Johnson, and C.S. Burrus. Gauss and the history of the fast fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, 1984.

[B11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[B12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-026. April 2012.

[B13] Charles Van Loan. *Computational frameworks for the fast Fourier transform.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

Index