# A Fast Fourier Transform Compiler

Matteo Frigo*
MIT Laboratory for Computer Science
545 Technology Square NE43-203
Cambridge, MA 02139
athena@theory.lcs.mit.edu

February 16, 1999

### Abstract

The FFTW library for computing the discrete Fourier transform (DFT) has gained a wide acceptance in both academia and industry, because it provides excellent performance on a variety of machines (even competitive with or faster than equivalent libraries supplied by vendors). In FFTW, most of the performance-critical code was generated automatically by a special-purpose compiler, called `genfft`, that outputs C code. Written in Objective Caml, `genfft` can produce DFT programs for any input length, and it can specialize the DFT program for the common case where the input data are real instead of complex. Unexpectedly, `genfft` "discovered" algorithms that were previously unknown, and it was able to reduce the arithmetic complexity of some other existing algorithms. This paper describes the internals of this special-purpose compiler in some detail, and it argues that a specialized compiler is a valuable tool.

## 1 Introduction

Recently, Steven G. Johnson and I released Version 2.0 of the FFTW library [FJ98, FJ], a comprehensive collection of fast C routines for computing the discrete Fourier transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. The DFT [DV90] is one of the most important computational problems, and many real-world applications require that the transform be computed as quickly as possible. FFTW is one of the fastest DFT programs available (see Figures 1 and 2) because of two unique features. First, FFTW automatically adapts the computation to the hardware. Second, the inner loop of FFTW (which amounts to 95% of the total code) was generated automatically by a special-purpose compiler written in Objective Caml [Ler98]. This paper explains how this compiler works.

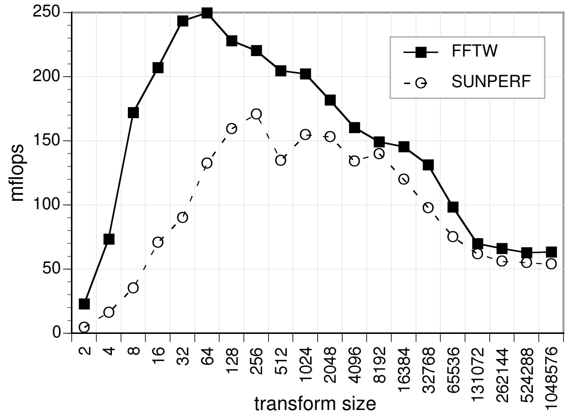


**Figure 1**: Graph of the performance of FFTW versus Sun's Performance Library on a 167 MHz UltraSPARC processor in single precision. The graph plots the speed in "mflops" (higher is better) versus the size of the transform. This figure shows sizes that are powers of two, while Figure 2 shows other sizes that can be factored into powers of 2, 3, 5, and 7. This distinction is important because the DFT algorithm depends on the factorization of the size, and most implementations of the DFT are optimized for the case of powers of two. See [FJ97] for additional experimental results. FFTW was compiled with Sun's C compiler (WorkShop Compilers 4.2 30 Oct 1996 C 4.2).

FFTW does not implement a single DFT algorithm, but it is structured as a library of ***codelets***—sequences of C code that can be composed in many ways. In FFTW's lingo, a composition of codelets is called a ***plan***. You can imagine the plan as a sort of bytecode that dictates which codelets should be executed in what order. (In the current FFTW implementation, however, a plan has a tree structure.) The precise plan used by FFTW depends on the size of the input (where "the input" is an array of $n$ complex numbers), and on which codelets happen to run fast on the underly-

*The author was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0270, and by a Digital Equipment Corporation fellowship.

To appear in *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, May 1999.

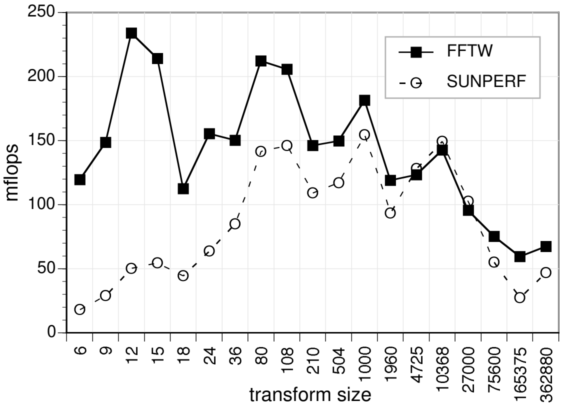


**Figure 2**: See caption of Figure 1.

ing hardware. The user need not choose a plan by hand, however, because FFTW chooses the fastest plan automatically. The machinery required for this choice is described elsewhere [FJ97, FJ98].

Codelets form the computational kernel of FFTW. You can imagine a codelet as a fragment of C code that computes a Fourier transform of a fixed size (say, 16, or 19).[1] FFTW's codelets are produced automatically by the ***FFTW codelet generator***, unimaginatively called `genfft`. `genfft` is an unusual special-purpose compiler. While a normal compiler accepts C code (say) and outputs numbers, `genfft` inputs the single integer $n$ (the size of the transform) and outputs C code. The codelet generator contains optimizations that are advantageous for DFT programs but not appropriate for a general compiler, and conversely, it does not contain optimizations that are not required for the DFT programs it generates (for example loop unrolling).

`genfft` operates in four phases.

1. In the ***creation*** phase, `genfft` produces a directed acyclic graph (dag) of the codelet, according to some well-known algorithm for the DFT [DV90]. The generator contains many such algorithms and it applies the most appropriate.

2. In the ***simplifier***, `genfft` applies local rewriting rules to each node of the dag, in order to simplify it. In traditional compiler terminology, this phase performs algebraic transformations and common-subexpression elimination, but it also performs other transformations that are specific to the DFT. For example, it turns out that if all floating point constants are made positive, the generated code runs faster. (See Section 5.) Another important transformation is ***dag transposition***, which derives from the theory of linear networks [CO75]. Moreover, besides noticing common subexpressions, the simplifier also attempts to create them. The simplifier is written in monadic style [Wad97], which allowed me to deal with the dag as if it were a tree, making the implementation much easier.

3. In the ***scheduler***, `genfft` produces a topological sort of the dag (a "schedule") that, for transforms of size $2^k$, provably minimizes the asymptotic number of register spills, *no matter how many registers the target machine has*. This truly remarkable fact can be derived from the analysis of the red-blue pebbling game of Hong and Kung [HK81], as we shall see in Section 6. For transforms of other sizes the scheduling strategy is no longer provably good, but it still works well in practice. Again, the scheduler depends heavily on the topological structure of DFT dags, and it would not be appropriate in a general-purpose compiler.

4. Finally, the schedule is unparsed to C. (It would be easy to produce FORTRAN or other languages by changing the unparser.) The unparser is rather obvious and uninteresting, except for one subtlety discussed in Section 7.

Although the creation phase uses algorithms that have been known for several years, the output of `genfft` is at times completely unexpected. For example, for a complex transform of size $n = 13$, the generator employs an algorithm due to Rader, in the form presented by Tolimieri and others [TAL97]. In its most sophisticated variant, this algorithm performs 214 real (floating-point) additions and 76 real multiplications. (See [TAL97, Page 161].) The generated code in FFTW for the same algorithm, however, contains only 176 real additions and 68 real multiplications, because `genfft` found certain simplifications that the authors of [TAL97] did not notice.[2]

The generator specializes the dag automatically for the case where the input data are real, which occurs frequently in applications. This specialization is nontrivial, and in the past the design of an efficient real DFT algorithm required a serious effort that was well worth a publication [SJHB87]. `genfft`, however, automatically derives real DFT programs from the complex algorithms, and the resulting programs have the same arithmetic complexity as those discussed by [SJHB87, Table II].[3] The generator also produces real variants of the Rader's algorithm mentioned above, which to my knowledge do not appear anywhere in the literature.

[1] In the actual FFTW system, some codelets perform more tasks, however. For the purposes of this paper, we consider only the generation of transforms of a fixed size.

[2] Buried somewhere in the computation dag generated by the algorithm are statements of the form $a = c + d$, $b = c - d$, $e = a + b$. The generator simplifies these statements to $e = 2 * c$, provided $a$ and $b$ are not needed elsewhere. Incidentally, [SB96] reports an algorithm with 188 additions and 40 multiplications, using a more involved DFT algorithm that I have not implemented yet. To my knowledge, the program generated by `genfft` performs the lowest known number of additions for this problem.

[3] In fact, `genfft` saves a few operations in certain cases, such as $n = 15$.

I found a special-purpose compiler such as the FFTW codelet generator to be a valuable tool, for a variety of reasons that I now discuss briefly.

- *Performance* was the main goal of the FFTW project, and it could not have been achieved without `genfft`. For example, the codelet that performs a DFT of size 64 is used routinely by FFTW on the Alpha processor. The codelet is about twice as fast as Digital's DXML library on the same machine. The codelet consists of about 2400 lines of code, including 912 additions and 248 multiplications. Writing such a program by hand would be a formidable task for any programmer. At least for the DFT problem, these long sequences of straight-line code seem to be necessary in order to take full advantage of large CPU register sets and the scheduling capabilities of C compilers.

- Achieving *correctness* has been surprisingly easy. The DFT algorithms in `genfft` are encoded straightforwardly using a high-level language. The simplification phase transforms this high-level algorithm into optimized code by applying simple algebraic rules that are easy to verify. In the rare cases during development when the generator contained a bug, the output was completely incorrect, making the bug manifest.

- *Rapid turnaround* was essential to achieve the performance goals. For example, the scheduler described in Section 6 is the result of a trial-and-error process in search of the best result, since the schedule of a codelet interacts with C compilers (in particular, with register allocators) in nonobvious ways. I could usually implement a new idea and regenerate the whole system within minutes, until I found the solution described in this paper.

- The generator is effective because it can apply *problem-specific* code improvements. For example, the scheduler is effective only for DFT dags, and it would perform poorly for other computations. Moreover, the simplifier performs certain improvements that depend on the DFT being a linear transformation.

- Finally, `genfft` derived some *new algorithms*, as in the example $n = 13$ discussed above. While this paper does not focus on these algorithms *per se*, they are of independent theoretical and practical interest in the Digital Signal Processing community.

This paper, of necessity, brings together concepts from both the Programming Languages and the Signal Processing literatures. The paper, however, is biased towards a reader familiar with compilers [ASU86], programming languages, and monads [Wad97], while the required signal-processing knowledge is kept to the bare minimum. (For example, I am not describing some advanced number-theoretical DFT algorithms used by the generator.) Readers unfamiliar with the discrete Fourier transform, however, are encouraged to read the good tutorial by Duhamel and Vetterli [DV90].

The rest of the paper is organized as follows. Section 2 gives some background on the discrete Fourier transform and on algorithms for computing it. Section 3 overviews related work on automatic generation of DFT programs. The remaining sections follow the evolution of a codelet within `genfft`. Section 4 describes what the codelet dag looks like and how it is constructed. Section 5 presents the dag simplifier. Section 6 describes the scheduler and proves that it minimizes the number of transfers between memory and registers. Section 7 discusses some pragmatic aspects of `genfft`, such as running time and memory requirements, and it discusses the interaction of `genfft`'s output with C compilers.

## 2 Background

This section defines the discrete Fourier transform (DFT), and mentions the most common algorithms to compute it.

Let $X$ be an array of $n$ complex numbers. The (forward) ***discrete Fourier transform*** of $X$ is the array $Y$ given by

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} , \tag{1}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive $n$-th root of unity, and $0 \leq i < n$. In case $X$ is a real vector, the transform $Y$ has the ***hermitian symmetry***

$$Y[n-i] = Y^*[i] ,$$

where $Y^*[i]$ is the complex conjugate of $Y^*[i]$.

The ***backward*** DFT flips the sign at the exponent of $\omega_n$, and it is defined in the following equation.

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{ij} . \tag{2}$$

The backward transform is the "scaled inverse" of the forward DFT, in the sense that computing the backward transform of the forward transform yields the original array multiplied by $n$.

If $n$ can be factored into $n = n_1 n_2$, Equation (1) can be rewritten as follows. Let $j = j_1 n_2 + j_2$, and $i = i_1 + i_2 n_1$. We then have,

$$Y[i_1 + i_2 n_1] = \tag{3}$$
$$\sum_{j_2=0}^{n_2-1} \left[ \left( \sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} .$$

This formula yields the ***Cooley-Tukey fast Fourier transform*** algorithm (FFT) [CT65]. The algorithm computes $n_2$

transforms of size $n_1$ (the inner sum), it multiplies the result by the so-called ***twiddle factors*** $\omega_n^{-i_1 j_2}$, and finally it computes $n_1$ transforms of size $n_2$ (the outer sum).

If $\gcd(n_1, n_2) = 1$, the ***prime factor*** algorithm can be applied, which avoids the multiplications by the twiddle factors at the expense of a more involved computation of indices. (See [OS89, page 619].) If $n$ is a multiple of 4, the ***split-radix*** algorithm [DV90] can save some operations with respect to Cooley-Tukey. If $n$ is prime, `genfft` uses either Equation (1) directly, or ***Rader's algorithm*** [Rad68], which converts the transform into a circular convolution of size $n - 1$. The circular convolution can be computed recursively using two Fourier transforms, or by means of a clever technique due to Winograd [Win78] (`genfft` does not employ this technique yet, however). Other algorithms are known for prime sizes, and this is still the subject of active research. See [TAL97] for a recent compendium on the topic. Any algorithm for the forward DFT can be readily adapted to compute the backward DFT, the difference being that certain complex constants become conjugate. For the purposes of this paper, we do not distinguish between forward and backward transform, and we simply refer to both as the "complex DFT".

In the case when the input is purely real, the transform can be computed with roughly half the number of operations of the complex case, and the hermitian output requires half the storage of a complex array of the same size. In general, keeping track of the hermitian symmetry throughout the recursion is nontrivial, however. This bookkeeping is relatively easy for the split-radix algorithm, and it becomes particularly nasty for the prime factor and the Rader algorithms. The topic is discussed in detail in [SJHB87]. In the real transform case, it becomes important to distinguish the forward transform, which takes a real input and produces an hermitian output, from the backward transform, whose input is hermitian and whose output is real, requiring a different algorithm. We refer to these cases as the "real to complex" and "complex to real" DFT, respectively.

In the DFT literature, unlike in most of Computer Science, it is customary to report the exact number of arithmetic operations performed by the various algorithms, instead of their asymptotic complexity. Indeed, the time complexity of all DFT algorithms of interest is $O(n \log n)$, and a detailed count of the exact number of operation is usually doable (which by no means implies that the analysis is easy to carry out). It is no problem for me to follow this convention in this paper, since `genfft` produces an exact arithmetic count anyway.

In the literature, the term FFT ("fast Fourier transform") refers to either the Cooley-Tukey algorithm or to any $O(n \log n)$ algorithm for the DFT, depending on the author. In this paper, FFT denotes any $O(n \log n)$ algorithm.

## 3 Related work

Researchers have been generating FFT programs for at least twenty years, possibly to avoid the tedium of getting all the implementation details right by hand. To my knowledge, the first generator of FFT programs was FOURGEN, written by J. A. Maruhn [Mar76]. It was written in PL/I and it generated FORTRAN.[4] FOURGEN is limited to transforms of size $2^k$.

Perez and Takaoka [PT87] present a generator of Pascal programs implementing a prime factor FFT algorithm. This program is limited to complex transforms of size $n$, where $n$ must be factorable into mutually prime factors in the set $\{2, 3, 4, 5, 7, 8, 9, 16\}$.

Johnson[5] and Burrus [JB83] applied dynamic programming to the automatic design of DFT modules. Selesnick and Burrus [SB96] used a program to generate MATLAB subroutines for DFTs of certain prime sizes. In many cases, these subroutines are the best known in terms of arithmetic complexity.

The EXTENT system by Gupta and others [GHSJ96] generates FORTRAN code in response to an input expressed in a ***tensor product*** language. Using the tensor product abstraction one can express concisely a variety of algorithms that includes the FFT and matrix multiplication (including Strassen's algorithm).

Another program called `genfft` generating Haskell FFT subroutines is part of the `nofib` benchmark for Haskell [Par92]. Unlike my program, this `genfft` is limited to transforms of size $2^k$. The program in `nofib` is not documented at all, but apparently it can be traced back to [HV92].

Veldhuizen [Vel95] used a template metaprograms technique to generate `C++` programs. The technique exploits the template facility of `C++` to force the `C++` compiler to perform computations at compile time.

All these systems are restricted to complex transforms, and the FFT algorithm is known *a priori*. To my knowledge, the FFTW generator is the only one that produces real algorithms, and in fact, which can *derive* real algorithms by specializing a complex algorithm. Also, my generator is the only one that addressed the problem of scheduling the program efficiently.

## 4 Creation of the expression dag

This section describes the creation of an expression dag. We first define the `node` data type, which encodes a directed acyclic graph (dag) of a codelet. We then describe a few

[4] Maruhn argues that PL/I is more suited than FORTRAN to this program-generation task, and has the following curious remark.

> One peculiar difficulty is that some FORTRAN systems produce an output format for floating-point numbers without the exponent delimiter "E", and this makes them illegal in FORTRAN statements.

[5] Unrelated to Steven G. Johnson, the other author of FFTW.

```
type node =
  | Num of Number.number
  | Load of Variable.variable
  | Store of Variable.variable * node
  | Plus of node list
  | Times of node * node
  | Uminus of node
```

**Figure 3**: Objective Caml code that defines the `node` data type, which encodes an expression dag.

ancillary data structures and functions that provide complex arithmetic. Finally, the bulk of the section describes the function `fftgen`, which actually produces the expression dag.

We start by defining the `node` data type, which encodes an arithmetic expression dag. Each node of the dag represents an operator, and the node's children represent the operands. This is the same representation as the one generally used in compilers [ASU86, Section 5.2]. A node in the dag can have more than one "parent", in which case the node represents a common subexpression. The definition of `node` is given in Figure 3, and it is straightforward. A node is either a real number (encoded by the abstract data type `Number.number`), a load of an input variable, a store of an expression into an output node, the sum of the children nodes, the product of two nodes, or the sign negation of a node. For example, the expression $a - b$, where $a$ and $b$ are input variables, is represented by `Plus [Load` $a$`; Uminus (Load` $b$`)]`.

The structure `Number` maintains floating-point constants with arbitrarily high precision (currently, 50 decimal digits), in case the user wants to use the quadruple precision floating-point unit on a processor such as the UltraSPARC. `Number` is implemented on top of Objective Caml's arbitrary-precision rationals. The structure `Variable` encodes the input/output nodes of the dag, and the temporary variables of the generated C code. Variables can be considered an abstract data type that is never used explicitly in this paper.

The `node` data type encodes expressions over real numbers, since the final C output contains only real expressions. For creating the expression dag of the codelet, however, it is convenient to express the algorithms in terms of complex numbers. The generator contains a structure called `Complex`, which implements complex expressions on top of the `node` data type, in a straightforward way.[6] The type `Complex.expr` (not shown) is essentially a pair of `node`s.

We now describe the function `fftgen`, which creates a dag for a DFT of size $n$. In the current implementation, `fftgen` uses one of the following algorithms.

[6] One subtlety is that a complex multiplication by a constant can be implemented with either 4 real multiplications and 2 real additions, or 3 real multiplications and 3 real additions [Knu98, Exercise 4.6.4-41]. The current generator uses the former algorithm, since the operation count of the dag is generally dominated by additions. On most CPUs, it is advantageous to move work from the floating-point adder to the multiplier.

- A split-radix algorithm [DV90], if $n$ is a multiple of 4. Otherwise,
- A prime factor algorithm (as described in [OS89, page 619]), if $n$ factors into $n_1 n_2$, where $n_i \neq 1$ and $\gcd(n_1, n_2) = 1$. Otherwise,
- The Cooley-Tukey FFT algorithm (Equation (3)) if $n$ factors into $n_1 n_2$, where $n_i \neq 1$. Otherwise,
- ($n$ is a prime number) Rader's algorithm for transforms of prime length [Rad68] if $n = 5$ or $n \geq 13$. Otherwise,
- Direct application of the definition of DFT (Equation (1)).

We now look at the operation of `fftgen` more closely. The function has type

```
fftgen : int -> (int -> Complex.expr) ->
         int -> (int -> Complex.expr)
```

The first argument to `fftgen` is the size `n` of the transform. The second argument is a function `input` with type `int -> Complex.expr`. The application `(input i)` returns a complex expression that contains the `i`-th input. The third argument `sign` is either 1 or $-1$, and it determines the direction of the transform.

Depending on the size $n$ of the requested transform, `fftgen` dispatches one of the algorithms mentioned above. We now discuss how the Cooley-Tukey FFT algorithm is implemented. The implementation of the other algorithms is similar, and it is not shown in this paper.

The Objective Caml code that implements the Cooley-Tukey algorithm can be found in Figure 4. In order to understand the code, recall Equation (3). This equation translates almost verbatim into Objective Caml. With reference to Figure 4, the function application `tmp1 j2` computes the inner sum of Equation (3) for a given value of $j_2$, and it returns a function of $i_1$. (`tmp1` is curried over $i_1$, and therefore $i_1$ does not appear explicitly in the definition.) Next, `(tmp1 j2 i1)` is multiplied by the twiddle factors, yielding `tmp2`, that is, the expression in square braces in Equation (3). Next, `tmp3` computes the outer summation, which is itself a DFT of size $n_2$. (Again, `tmp3` is a function of $i_1$ and $i_2$, curried over $i_2$.) In order to obtain the $i$-th element of the output of the transform, the index $i$ is finally mapped into $i_1$ and $i_2$ and `(tmp3 i1 i2)` is returned.

Observe that the code in Figure 4 does not actually perform any computation. Instead, it builds a symbolic expression dag that specifies the computation. The other DFT algorithms are implemented in a similar fashion.

At the top level, the generator invokes `fftgen` with the size `n` and the direction `sign` specified by the user. The `input` function is set to `fun i -> Complex.load (Variable.input i)`, i.e., a function that loads the $i$-th input variable. Recall now that `fftgen` returns a function

```
let rec cooley_tukey n1 n2 input sign =
  let tmp1 j2 = fftgen n1
    (fun j1 -> input (j1 * n2 + j2)) sign in
  let tmp2 i1 j2 =
    exp n (sign * i1 * j2) @* tmp1 j2 i1 in
  let tmp3 i1 = fftgen n2 (tmp2 i1) sign
  in
   (fun i -> tmp3 (i mod n1) (i / n1))
```

**Figure 4**: Fragment of the FFTW codelet generator that implements the Cooley-Tukey FFT algorithm. The infix operator `@*` computes the complex product. The function `exp n k` computes the constant $\exp(2\pi k\sqrt{-1}/n)$.

`output`, where `(output i)` is a complex expression that computes the $i$-th element of the output array. The top level builds a list of `Store` expressions that store `(output i)` into the $i$-th output variable, for all $0 \leq i < n$. This list of `Stores` is the codelet dag that forms the input of subsequent phases of the generator.

We conclude this section with a few remarks. According to the description given in this section, `fftgen` contains no special support for the case where the input is real. This statement is not completely true. In the actual implementation, `fftgen` maintains certain symmetries explicitly (for example, if the input is real, then the output is known to have hermitian symmetry). These additional constraints do not change the final output, but they speed up the generation process, since they avoid computing and simplifying the same expression twice. For the same reason, the actual implementation memoizes expressions such as `tmp1 i2 i1` in Figure 4, so that they are only computed once.[7]

At this stage, the generated dag contains many redundant computations, such as multiplications by 1 or 0, additions of 0, and so forth. `fftgen` makes no attempt to eliminate these redundancies. Figure 5 shows a possible C translation of a codelet dag at this stage of the generation process.

## 5 The simplifier

In this section, we present FFTW's ***simplifier***, which transforms code such as the one in Figure 5 into simpler code. The simplifier transforms a dag of `nodes` (see Section 4) into another dag of `nodes`. We first discuss how the simplifier transforms the dag, and then how the simplifier is actually implemented. The implementation benefits heavily from the use of monads.

### 5.1 What the simplifier does

We first illustrate the improvements applied by the simplifier to the dag. The simplifier traverses the dag bottom-up, and it applies a series of local improvements to every node. For explanation purposes, these improvements can be subdivided into three categories: algebraic transformations, common-subexpression elimination, and DFT-specific improvements. Since the first two kinds are well-known [ASU86], I just discuss them briefly. We then consider the third kind in more detail.

[7]These performance improvements were important for a user of FFTW who needed a hard-coded transform of size 101, and had not obtained an answer after the generator had run for three days. See Section 7 for more details.

```
tmp1 = REAL(input[0]);
tmp5 = REAL(input[0]);
tmp6 = IMAG(input[0]);
tmp2 = IMAG(input[0]);
tmp3 = REAL(input[1]);
tmp7 = REAL(input[1]);
tmp8 = IMAG(input[1]);
tmp4 = IMAG(input[1]);
REAL(output[0]) = ((1 * tmp1) - (0 * tmp2))
      + ((1 * tmp3) - (0 * tmp4));
IMAG(output[0]) = ((1 * tmp2) + (0 * tmp1))
      + ((1 * tmp4) + (0 * tmp3));
REAL(output[1]) = ((1 * tmp5) - (0 * tmp6))
      + ((-1 * tmp7) - (0 * tmp8));
IMAG(output[1]) = ((1 * tmp6) + (0 * tmp5))
      + ((-1 * tmp8) + (0 * tmp7));
```

**Figure 5**: C translation of a dag for a complex DFT of size 2, as generated by `fftgen`. Variable declarations have been omitted from the figure. The code contains many common subexpression (e.g., `tmp1` and `tmp5`), and redundant multiplications by 0 or 1.

*Algebraic transformations* reduce the arithmetic complexity of the dag. Like a traditional compiler, the simplifier performs constant folding, and it simplifies multiplications by 0, 1, or $-1$, and additions of 0. Moreover, the simplifier applies the distributive property systematically. Expressions of the form $kx + ky$ are transformed into $k(x + y)$. In the same way, expressions of the form $k_1x+k_2x$ are transformed into $(k_1 + k_2)x$. In general, these two transformations have the potential of destroying common subexpressions, and they might increase the operation count. This does not appear to be the case for all DFT dags I have studied, although I do not fully understand the reason for this phenomenon.

*Common-subexpression elimination* is also applied systematically. Not only does the simplifier eliminate common subexpressions, it also attempts to create new ones. For example, it is common for a DFT dag (especially in the case of real input) to contain both $x-y$ and $y-x$ as subexpressions, for some $x$ and $y$. The simplifier converts both expressions to either $x - y$ and $-(x - y)$, or $-(y - x)$ and $y - x$, depending on which expression is encountered first during the dag traversal.

The simplifier applies two kinds of *DFT-specific improvements*. First, all numeric constants are made positive, possibly propagating a minus sign to other nodes of the dag. This curious transformation is effective because constants generally appear in pairs $k$ and $-k$ in a DFT dag. To my knowl-

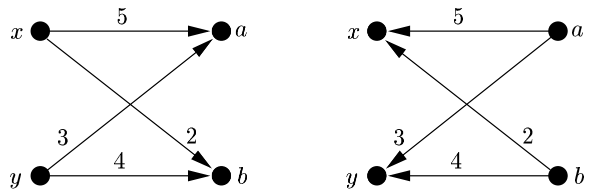


**Figure 6**: Illustration of "network" transposition. Each graph defines an algorithm for computing a linear function. These graphs are called ***linear networks***, and they can be interpreted as follows. Data are flowing in the network, from input nodes to output nodes. An edge multiplies data by some constant (possibly 1), and each node is understood to compute the sum of all incoming edges. In this example, the network on the left computes $a = 5x + 3y$ and $b = 2x + 4y$. The network on the right is the "transposed" form of the first network, obtained by reversing all edges. The new network computes the linear function $x = 5a + 2b$ and $y = 3a + 4b$. In general, if a network computes $x = My$ for some matrix $M$, the transposed network computes $y = M^T x$. (See [CO75] for a proof.) These linear networks are similar to but not the same as expression dags normally used in compilers and in `genfft`, because in the latter case the nodes and not the edges perform computation. A network can be easily transformed into an expression dag, however. The converse is not true in general, but it is true for DFT dags where all multiplications are by constants.

edge, every C compiler would store both $k$ and $-k$ in the program text, and it would load both constants into a register at runtime. Making all constants positive reduces the number of loads of constants by a factor of two, and this transformation alone speeds up the generated codelets by 10-15% on most machines. This transformation has the additional effect of converting subexpressions into a canonical form, which helps common-subexpression elimination.

The second DFT-specific improvement is not local to nodes, and is instead applied to the whole dag. The transformation is based on the fact that a dag computing a linear function can be "reversed" yielding a ***transposed*** dag [CO75]. This transposition process is well-known in the Signal Processing literature [OS89, page 309], and it operates a shown in Figure 6. It turns out that in certain cases the transposed dag exposes some simplifications that are not present in the original dag. (An example will be shown later.) Accordingly, the simplifier performs three passes over the dag. It first simplifies the original dag $D$ yielding a dag $E$. Then, it simplifies the transposed dag $E^T$ yielding a dag $F^T$. Finally, it simplifies $F$ (the transposed dag of $F^T$) yielding a dag $G$.[8] Figure 7 shows the savings in arithmetic complexity that derive from dag transposition for codelets of various sizes. As it can be seen in the figure, transposition can reduce the number of multiplications, but it does not reduce the number of additions.

Figure 8 shows a simple case where transposition is beneficial. The network in the figure computes $c = 4 \cdot (2a + 3b)$. It is not safe to simplify this expression to $c = 8a + 12b$, since this transformation destroys the common subexpressions $2a$ and $3b$. (The transformation destroys one operation and two common subexpressions, which might increase the operation count by one.) Indeed, the whole point of most FFT algorithms is to create common subexpressions. When the network is transposed, however, it computes $a = 2 \cdot 4c$ and $b = 3 \cdot 4c$. These transposed expressions *can* be safely transformed into $a = 8c$ and $b = 12c$ because each transformation saves one operation and destroys one common subexpression. Consequently, the operation count cannot increase. In a sense, transposition provides a simple and elegant way to detect which dag nodes have more than one parent, which would be difficult to detect when the dag is being traversed.

[8] Although one might imagine iterating this process, three passes seem to be sufficient in all cases.

| size | adds (not transposed) | muls (not transposed) | adds (transposed) | muls (transposed) |
|---|---|---|---|---|
| complex to complex | | | | |
| 5 | 32 | 16 | 32 | 12 |
| 10 | 84 | 32 | 84 | 24 |
| 13 | 176 | 88 | 176 | 68 |
| 15 | 156 | 68 | 156 | 56 |
| real to complex | | | | |
| 5 | 12 | 8 | 12 | 6 |
| 10 | 34 | 16 | 34 | 12 |
| 13 | 76 | 44 | 76 | 34 |
| 15 | 64 | 31 | 64 | 25 |
| complex to real | | | | |
| 5 | 12 | 9 | 12 | 7 |
| 9 | 32 | 20 | 32 | 18 |
| 10 | 34 | 18 | 34 | 14 |
| 12 | 38 | 14 | 38 | 10 |
| 13 | 76 | 43 | 76 | 35 |
| 15 | 64 | 37 | 64 | 31 |
| 16 | 58 | 22 | 58 | 18 |
| 32 | 156 | 62 | 156 | 54 |
| 64 | 394 | 166 | 394 | 146 |
| 128 | 956 | 414 | 956 | 374 |

**Figure 7**: Summary of the benefits of dag transposition. The table shows the number of additions and multiplications for codelets of various size, with and without dag transposition. Sizes for which the transposition has no effect are not reported in this table.

### 5.2 Implementation of the simplifier

The simplifier is written in monadic style [Wad97]. The monad performs two important functions. First, it allows the simplifier to treat the expression dag as if it were a tree, which makes the implementation considerably easier. Second, the monad performs common-subexpression elimination. We now discuss these two topics.

**Treating dags as trees.** Recall that the goal of the sim-

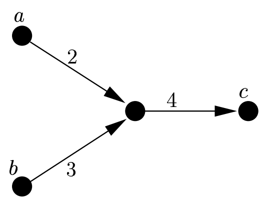


**Figure 8**: A linear network where which dag transposition exposes some optimization possibilities. See the text for an explanation.

plifier is to simplify an expression dag. The simplifier, however, is written as if it were simplifying an expression *tree*. The map from trees to dags is accomplished by memoization, which is performed implicitly by a monad. The monad maintains a table of all previously simplified dag nodes, along with their simplified versions. Whenever a node is visited for the second time, the monad returns the value in the table.

In order to continue reading this section, you really should be familiar with monads [Wad97]. In any case, here is a very brief summary on monads. The idea of a monadic-style program is to convert all expressions of the form

```
let x = a in (b x)
```

into something that looks like

```
a >>= fun x -> returnM (b x)
```

The code should be read "call `f`, and then name the result `x` and return `(b x)`." The advantage of this transformation is that the meanings of "then" (the infix operator `>>=`) and "return" (the function `returnM`) can be defined so that they perform all sorts of interesting activities, such as carrying state around, perform I/O, act nondeterministically, etc. In the specific case of the FFTW simplifier, `>>=` is defined so as to keep track of a few tables used for memoization, and `returnM` performs common-subexpression elimination.

The core of the simplifier is the function `algsimpM`, as shown in Figure 9. `algsimpM` dispatches on the argument `x` (of type `node`), and it calls a simplifier function for the appropriate case. If the node has subnodes, the subnodes are simplified first. For example, suppose `x` is a `Times` node. Since a `Times` node has two subnodes `a` and `b`, the function `algsimpM` first calls itself recursively on `a`, yielding `a'`, and then on `b`, yielding `b'`. Then, `algsimpM` passes control to the function `stimesM`. If both `a'` and `b'` are constants, `stimesM` computes the product directly. In the same way, `stimesM` takes care of the case where either `a'` or `b'` is 0 or 1, and so on. The code for `stimesM` is shown in Figure 10.

The neat trick of using memoization for graph traversal was invented by Joanna Kulik in her master's thesis [Kul95], as far as I can tell.

**Common-subexpression elimination (CSE)** is performed behind the scenes by the monadic operator `returnM`. The CSE algorithm is essentially the classical bottom-up

```
let rec algsimpM x =
  memoizing
    (function
        Num a -> snumM a
      | Plus a ->
          mapM algsimpM a >>= splusM
      | Times (a, b) ->
          algsimpM a >>= fun a' ->
            algsimpM b >>= fun b' ->
              stimesM (a', b')
      | Uminus a ->
          algsimpM a >>= suminusM
      | Store (v, a) ->
          algsimpM a >>= fun a' ->
            returnM (Store (v, a'))
      | x -> returnM x)
    x
```

**Figure 9**: The top-level simplifier function `algsimpM`, written in monadic style. See the text for an explanation.

construction from [ASU86, page 592]. The monad maintains a table of all nodes produced during the traversal of the dag. Each time a new node is constructed and returned, `returnM` checks whether the node appears elsewhere in the dag. If so, the new node is discarded and `returnM` returns the old node. (Two nodes are considered the same if they compute equivalent expressions. For example, $a + b$ is the same as $b + a$.)

It is worth remarking that the simplifier *interleaves* common-subexpression elimination with algebraic transformations. To see why interleaving is important, consider for example the expression $a - a'$, where $a$ and $a'$ are distinct nodes of the dag that compute the same subexpression. CSE rewrites the expression to $a - a$, which is then simplified to $0$. This pattern occurs frequently in DFT dags.

## 6 The scheduler

In this section we discuss the `genfft` scheduler, which produces a topological sort of the dag in an attempt to maximize register usage. For transforms whose size is a power of 2, we prove that a schedule exists that is asymptotically optimal in this respect, even though the schedule is independent of the number of registers. This fact is derived from the red-blue pebbling game of Hong and Kung [HK81].

Even after simplification, a codelet dag of a large transform still contains hundreds or even thousands of nodes, and there is no way to execute it fully within the register set of any existing processor. The scheduler attempts to reorder the dag in such a way that register allocators commonly used in compilers [Muc97, Section 16] can minimize the number of register spills. Note that the FFTW codelet generator does not address the *instruction scheduling* problem; that is, the

```
let rec stimesM = function
  | (Uminus a, b) -> (* -a * b  ==> -(a * b) *)
     stimesM (a, b) >>= suminusM
  | (a, Uminus b) -> (* a * -b  ==> -(a * b) *)
     stimesM (a, b) >>= suminusM
  | (Num a, Num b) -> (* multiply two numbers *)
     snumM (Number.mul a b)
  | (Num a, Times (Num b, c)) ->
     snumM (Number.mul a b) >>= fun x ->
       stimesM (x, c)
  | (Num a, b) when Number.is_zero a ->
     snumM Number.zero   (* 0 * b  ==> 0 *)
  | (Num a, b) when Number.is_one a ->
     returnM b           (* 1 * b  ==> b *)
  | (Num a, b) when Number.is_mone a ->
     suminusM b          (* -1 * b  ==> -b *)
  | (a, (Num _ as b')) -> stimesM (b', a)
  | (a, b) -> returnM (Times (a, b))
```

**Figure 10**: Code for the function `stimesM`, which simplifies the product of two expressions. The comments (delimited with `(* *)`) briefly discuss the various simplifications. Even if it operates on a dag, this is exactly the code one would write to simplify a tree.

maximization of pipeline usage is left to the C compiler.

Figure 11 illustrates the scheduling problem. Suppose a processor has 4 registers, and consider a "column major" execution order that first executes all nodes in the diagonally-striped box (say, top-down), and then proceeds to the next column of nodes. Since there are 8 values to propagate from column to column, and the machine has 4 registers, at least four registers must be spilled if this strategy is adopted. A different strategy would be to execute all operations in the gray box before executing any other node. These operations can be performed fully within registers once the input nodes have been loaded. It is clear that different schedules lead to different behaviors with respect to register spills.

A lower bound on the number of register spills incurred by any execution of the FFT graph was first proved by Hong and Kung [HK81] in the context of the so-called "red-blue pebbling game". Paraphrased in compiler terminology, Theorem 2.1 from [HK81] states that the execution of the FFT graph of size $n = 2^k$ on a machine with $C$ registers (where $C \leq n$) requires at least $\Omega(n \log n / \log C)$ register spills.[9] Aggarwal and Vitter [AV88] generalize this result to disk I/O, where a single I/O operation can transfer a block of elements. In addition, Aggarwal and Vitter give a schedule that matches the lower bound. Their schedule is constructed as in the example that follows. With reference to Figure 11, assume again that the machine has $C = 4$ registers. The schedule loads the four topmost input nodes of the dag, and then executes all nodes in the gray box, which can be done completely using the 4 registers. Then, the four outputs of the gray box are written to some temporary memory location, and the same process is repeated for the four inputs in the bottom part of the dag. Finally, the schedule executes the rightmost column of the dag. In general, the algorithm proceeds by partitioning the dag into blocks that have $C$ input nodes and $\Theta(\log C)$ depth. Consequently, there are $\Theta(n \log n / (C \log C))$ such blocks, and each block can be executed with $O(C)$ transfers between registers and memory. The total number of transfers is thus at most $O(n \log n / \log C)$, and the algorithm matches the lower bound.

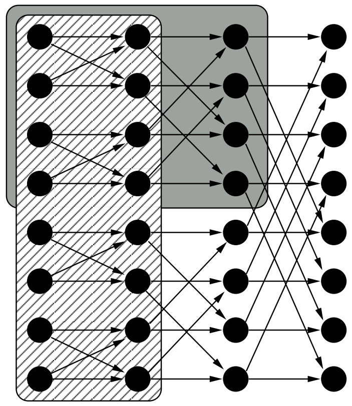

**Figure 11**: Illustration of the scheduling problem. The butterfly graph (in black) represents an abstraction of the data flow of the fast Fourier transform algorithm on 8 inputs. (In practice, the graph is more complicated because data are complex, and the real and imaginary part interact in nontrivial ways.) The boxes denote two different execution orders that are explained in the text.

Unfortunately, Aggarwal and Vitter's algorithm depends on $C$, and a schedule for a given value of $C$ does not work well for other values. The aim of the FFTW generator is to produce portable code, however, and the generator cannot make any assumption about $C$. It is perhaps surprising that a schedule exists that matches the asymptotic lower bound for all values of $C$. In other words, a single sequential order of execution of an FFT dag exists that, for all $C$, requires $O(n \log n / \log C)$ register spills on a machine with $C$ registers. We say that such a schedule is ***cache-oblivious***.[10]

We now show that the Cooley-Tukey FFT becomes cache-oblivious when the factors of $n$ are chosen appropriately (as in [VS94a, VS94b]). We first formulate a recursive algorithm, which is easier to understand and analyze than a dag. Then, we examine how the computation dag of the algorithm should be scheduled in order to mimic the register/cache behavior of the cache-oblivious algorithm.

Consider the Cooley-Tukey algorithm applied to a trans-

[9] The same result holds for any two-level memory, such as L1 cache vs. L2, or physical memory vs. disk.

[10] We say "cache-" and not "register-oblivious" since this notion first arose from the analysis of the caching behavior of Cilk [FLR98] programs using shared memory. Work is still in progress to understand and define cache-obliviousness formally, and this concept does not yet appear in the literature. Simple divide-and-conquer cache-oblivious algorithms for matrix multiplication and LU decomposition are described in [BFJ+96].

form of size $n = 2^k$. Assume for simplicity that $k$ is itself a power of two, although the result holds for any positive integer $k$. At every stage of the recursion, we have a choice of the factors $n_1$ and $n_2$ of $n$. Choose $n_1 = n_2 = \sqrt{n}$. The algorithm computes $\sqrt{n}$ transforms of size $\sqrt{n}$, followed by $O(n)$ multiplications by the twiddle factors, followed by $\sqrt{n}$ more transforms of size $\sqrt{n}$. When $n < \Theta(C)$, the transform can be computed fully within registers. Thus, the number $M(n)$ of transfers between memory and registers when computing a transform of size $n$ satisfies this recurrence.

$$M(n) = \begin{cases} 2\sqrt{n}M(\sqrt{n}) + O(n) & \text{when } n > \Theta(C) \text{ ;} \\ \Theta(C) & \text{otherwise .} \end{cases}$$

The recurrence has solution $M(n) = O(n \log n / \log C)$, which matches the lower bound.

We now reexamine the operation of the cache-oblivious FFT algorithm in terms of the FFT dag as the one in Figure 11. Partitioning a problem of size $n$ into $\sqrt{n}$ problems of size $\sqrt{n}$ is equivalent to cutting the dag with a vertical line that partitions the dag into two halves of (roughly) equal size. Every node in the first half is executed before any node in the second half. Each half consists of $\Theta(\sqrt{n})$ connected components, which are scheduled recursively in the same way.

The `genfft` scheduler uses this recursive partitioning technique for transforms of all sizes (not just powers of 2). The scheduler cuts the dag roughly into two halves. "Half a dag" is not well defined, however, except for the power of 2 case, and therefore the `genfft` scheduler uses a simple heuristic (described below) to compute the two halves for the general case. The cut induces a set of connected components that are scheduled recursively. The scheduler guarantees that all components in the first half of the dag (the one containing the inputs) are executed before the second half is scheduled. For the special case $n = 2^k$, because of the previous analysis, we know that this schedule of the dag allows the register allocator of the C compiler to minimize the number of register spills (up to some constant factor). Little is known about the optimality of this scheduling strategy for general $n$, for which neither the lower-bound nor the upper-bound analyses hold.

Finally, we discuss the heuristic used to cut the dag into two halves. The heuristic consists of "burning the candle at both ends". Initially, the scheduler colors the input nodes red, the output nodes blue, and all other nodes black. After this initial step, the scheduler alternates between a red and a blue coloring phase. In a red phase, any node whose predecessors are all red becomes red. In a blue phase, all nodes whose successors are blue are colored blue. This alternation continues while black nodes exist. When coloring is done, red nodes form the first "half" of the dag, and blue nodes the second. When $n$ is a power of two, the FFT dag has a regular structure like the one shown in Figure 11, and this process has the effect of cutting the dag in the middle with a vertical line, yielding the desired optimal behavior.

## 7 Pragmatic aspects of `genfft`

This section discusses briefly the running time and the memory requirements of `genfft`, and also some problems that arise in the interaction of the `genfft` scheduler with C compilers.

The FFTW codelet generator is not optimized for speed, since it is intended to be run only once. Indeed, users of FFTW can download a distribution of generated C code and never run `genfft` at all. Nevertheless, the resources needed by `genfft` are quite modest. Generation of C code for a transform of size 64 (the biggest used in FFTW) takes about 75 seconds on a 200MHz Pentium Pro running Linux 2.2 and the native-code compiler of Objective Caml 2.01. `genfft` needs less than 3 MB of memory to complete the generation. The resulting codelet contains 912 additions, 248 multiplications. On the same machine, the whole FFTW system can be regenerated in about 15 minutes. The system contains about 55,000 lines of code in 120 files, consisting of various kinds of codelets for forward, backward, real to complex, and complex to real transforms. The sizes of these transforms in the standard FFTW distribution include all integers up to 16 and all powers of two up to 64.

A few FFTW users needed fast hard-coded transforms of uncommon sizes (such as 19 and 23), and they were able to run the generator to produce a system tailored to their needs. The biggest program generated so far was for a complex transform of size 101, which required slightly less than two hours of CPU time on the Pentium Pro machine, and about 10 MB of memory. Again, a user had a special need for such a transform, which would be formidable to code by hand. In order to achieve this running time, I was forced to replace a linked-list implementation of associative tables by hashing, and to avoid generating "obvious" common subexpressions more than once when the dag is created. The naive generator was somewhat more elegant, but had not produced an answer after three days.

The long sequences of straight-line code produced by `genfft` can push C compilers (in particular, register allocators) to their limits. The combined effect of `genfft` and of the C compiler can lead to performance problems. The following discussion presents two particular cases that I found particularly surprising, and is not intended to blame any particular compiler or vendor.

The optimizer of the `egcs-1.1.1` compiler performs an instruction scheduling pass, followed by register allocation, followed by another instruction scheduling pass. On some architectures, including the SPARC and PowerPC processors, `egcs` employs the so-called "Haifa scheduler", which usually produces better code than the normal `egcs/gcc` scheduler. The first pass of the Haifa scheduler, however, has the unfortunate effect of destroying `genfft`'s schedule (computed as explained in Section 6). In `egcs`, the first instruction scheduling pass can be disabled with the option

```
void foo(void)
{
 double a;
 double b;

 .. lifetime of a ..
 .. lifetime of b ..
}
```

```
void foo(void)
{
 {
  double a;
  .. lifetime of a ..
 }
 {
  double b;
  .. lifetime of b ..
 }
}
```

**Figure 12**: Two possible declarations of local variables in C. On the left side, variables are declared in the topmost lexical scope. On the right side, variables are declared in a private lexical scope that encompasses the lifetime of the variable.

`-fno-schedule-insns`, and on a 167 MHz UltraSPARC I, the compiled code is between 50% and 100% faster and about half the size when this option is used. Inspection of the assembly code produced by `egcs` reveals that the difference consists entirely of register spills and reloads.

Digital's C compiler for Alpha (DEC C V5.6-071 on Digital UNIX V4.0 (Rev. 878)) seems to be particularly sensitive to the way local variables are declared. For example, Figure 12 illustrates two ways to declare temporary variables in a C program. Let's call them the "left" and the "right" style. `genfft` can be programmed to produce code in either way, and for most compilers I have tried there is no appreciable performance difference between the two styles. Digital's C compiler, however, appears to produce better code with the right style (the right side of Figure 12). For a transform of size 64, for example, and compiler flags `-newc -w0 -O5 -ansi_alias -ansi_args -fp_reorder -tune host -std1`, a 467MHz Alpha achieves about 450 MFLOPS with the left style, and 600 MFLOPS with the right style. (Different sizes lead to similar results.) I could not determine the exact source of this difference.

## 8 Conclusion

In my opinion, the main contribution of this paper is to present a real-world application of domain-specific compilers and of advanced programming techniques, such as monads. In this respect, the FFTW experience has been very successful: the current release FFTW-2.0.1 is being downloaded by more than 100 people every week, and a few users have been motivated to learn ML after their experience with FFTW. In the rest of this concluding section, I offer some ideas about future work and possible developments of the FFTW system.

The current `genfft` program is somewhat specialized to computing linear functions, using algorithms whose control structure is independent of the input. Even with this restriction, the field of applicability of `genfft` is potentially huge. For example, signal processing FIR and IIR filters fall into this category, as well as other kinds of transforms used in image processing (for example, the discrete cosine transform used in JPEG). I am confident that the techniques described in this paper will prove valuable in this sort of application.

Recently, I modified `genfft` to generate crystallographic Fourier transforms [ACT90]. In this particular application, the input consists of 2D or 3D data with certain symmetries. For example, the input data set might be invariant with respect to rotations of 60 degrees, and it is desirable to have a special-purpose FFT algorithm that does not execute redundant computations. Preliminary investigation shows that `genfft` is able to exploit most symmetries. I am currently working on this problem.

In its present form, `genfft` is somewhat unsatisfactory because it intermixes programming and metaprogramming. At the programming level, one specifies a DFT algorithm, as in Figure 4. At the metaprogramming level, one specifies how the program should be simplified and scheduled. In the current implementation, the two levels are confused together in a single binary program. It would be nice to have a clean separation of these two levels, but I currently do not know how to do it.

## 9 Acknowledgments

I am grateful to Compaq for awarding me the Digital Equipment Corporation fellowship. Thanks to Compaq, Intel, and Sun Microsystems for donations of hardware that was used to develop `genfft`. Prof. Charles E. Leiserson of MIT provided continuous support and encouragement. FFTW would not exist without him. Charles also proposed the name "codelets" for the basic FFT blocks. Thanks to Steven G. Johnson for sharing with me the development of FFTW. Thanks to Steven and the anonymous referees for helpful comments on this paper. The notion of cache obliviousness arose in discussions with Charles Leiserson, Harald Prokop, Sridhar "Cheema" Ramachandran, and Keith Randall.

## References

[ACT90] Myoung An, James W. Cooley, and Richard Tolimieri. Factorization method for crystallographic Fourier transforms. *Advances in Applied Mathematics*, 11:358–371, 1990.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, March 1986.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[CO75] R. E. Crochiere and A. V. Oppenheim. Analysis of linear digital networks. *Proceedings of the IEEE*, 63:581–595, April 1975.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[DV90] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19:259–299, April 1990.

[FJ] Matteo Frigo and Steven G. Johnson. The FFTW web page. `http://theory.lcs.mit.edu/~fftw`.

[FJ97] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the West. Technical Report MIT-LCS-TR-728, MIT Lab for Computer Science, September 1997. The description of the codelet generator given in this report is no longer current.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, June 1998. ACM.

[GHSJ96] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A framework for generating distributed-memory parallel programs for block recursive algorithms. *Journal of Parallel and Distributed Computing*, 34(2):137–153, 1 May 1996.

[HK81] Jia-Wei Hong and H. T. Kung. I/O complexity: the red-blue pebbling game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326–333, Milwaukee, 1981.

[HV92] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language—a case study: the fast Fourier transform. In G. Hains and L. M. R. Mullin, editors, *Arrays, functional languages, and parallel systems (ATABLE)*, pages 52–66, June 1992.

[JB83] H. W. Johnson and C. S. Burrus. The design of optimal DFT algorithms using dynamic programming. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 31:378–387, April 1983.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 2 (Seminumerical Algorithms). Addison-Wesley, 3rd edition, 1998.

[Kul95] Joanna L. Kulik. Implementing compiler optimizations using parallel graph reduction. Master's thesis, Massachussets Institute of Technology, February 1995.

[Ler98] Xavier Leroy. *The Objective Caml system release 2.00*. Institut National de Recherche en Informatique at Automatique (INRIA), August 1998.

[Mar76] J. A. Maruhn. FOURGEN: a fast Fourier transform program generator. *Computer Physics Communications*, 12:147–162, 1976.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[OS89] A. V. Oppenheim and R. W. Schafer. *Discrete-time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1989.

[Par92] Will Partain. The `nofib` benchmark suite of Haskell programs. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer Verlag, 1992.

[PT87] F. Perez and T. Takaoka. A prime factor FFT algorithm implementation using a program generation technique. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(8):1221–1223, August 1987.

[Rad68] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. of the IEEE*, 56:1107–1108, June 1968.

[SB96] I. Selesnick and C. S. Burrus. Automatic generation of prime length FFT programs. *IEEE Transactions on Signal Processing*, pages 14–24, January 1996.

[SJHB87] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849–863, June 1987.

[TAL97] Richard Tolimieri, Myoung An, and Chao Lu. *Algorithms for Discrete Fourier Transform and Convolution*. Springer Verlag, 1997.

[Vel95] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[VS94a] J. S. Vitter and E. A. M. Shriver. Optimal algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994. double special issue on Large-Scale Memories.

[VS94b] J. S. Vitter and E. A. M. Shriver. Optimal algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994. double special issue on Large-Scale Memories.

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

[Win78] S. Winograd. On computing the discrete Fourier transform. *Mathematics of Computation*, 32(1):175–199, January 1978.