The Design and Implementation of FFTW3

Matteo Frigo and Steven G. Johnson

(Invited Paper)

Abstract— **FFTW is an implementation of the discrete Fourier transform (DFT) that adapts to the hardware in order to maximize performance. This paper shows that such an approach can yield an implementation that is competitive with hand-optimized libraries, and describes the software structure that makes our current FFTW3 version flexible and adaptive. We further discuss a new algorithm for real-data DFTs of prime size, a new way of implementing DFTs by means of machine-specific "SIMD" instructions, and how a special-purpose compiler can derive optimized implementations of the discrete cosine and sine transforms automatically from a DFT algorithm.**

Index Terms— **FFT, adaptive software, Fourier transform, cosine transform, Hartley transform, I/O tensor.**

I. Introduction

FFTW [1] is a widely used free-software library that computes the discrete Fourier transform (DFT) and its various special cases. Its performance is competitive even with vendor-optimized programs, but unlike these programs, FFTW is not tuned to a fixed machine. Instead, FFTW uses a *planner* to adapt its algorithms to the hardware in order to maximize performance. The input to the planner is a *problem*, a multi-dimensional loop of multi-dimensional DFTs. The planner applies a set of rules to recursively decompose a problem into simpler sub-problems of the same type. "Sufficiently simple" problems are solved directly by optimized, straight-line code that is automatically generated by a special-purpose compiler. This paper describes the overall structure of FFTW as well as the specific improvements in FFTW3, our latest version.

FFTW is fast, but its speed does not come at the expense of flexibility. In fact, FFTW is probably the most flexible DFT library available:

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFTs in $O(n \log n)$ time for any length n. (Most other DFT implementations are either restricted to a subset of sizes or they become $\Theta(n^2)$ for certain values of n, for example when n is prime.)
- FFTW imposes no restrictions on the rank (dimensionality) of multi-dimensional transforms. (Most other implementations are limited to one-dimensional, or at most two- and three-dimensional data.)
- FFTW supports multiple and/or strided DFTs; for example, to transform a 3-component vector field or a portion of a multi-dimensional array. (Most implementations support only a single DFT of contiguous data.)
- FFTW supports DFTs of real data, as well as of real symmetric/antisymmetric data (also called discrete cosine/sine transforms).

The interaction of the user with FFTW occurs in two stages: planning, in which FFTW adapts to the hardware, and execution, in which FFTW performs useful work for the user. To compute a DFT, the user first invokes the FFTW *planner*, specifying the *problem* to be solved. The problem is a data structure that describes the "shape" of the input data—array sizes and memory layouts—but does not contain the data itself. In return, the planner yields a *plan*, an executable data structure that accepts the input data and computes the desired DFT. Afterwards, the user can execute the plan as many times as desired.

The FFTW planner works by measuring the actual run time of many different plans and by selecting the fastest one. This process is analogous to what a programmer would do by hand when tuning a program to a fixed machine, but in FFTW's case no manual intervention is required. Because of the repeated performance measurements, however, the planner tends to be time-consuming. In performance-critical applications, many transforms of the same size are typically required, and therefore a large one-time cost is usually acceptable. Otherwise, FFTW provides a mode of operation where the planner quickly returns a "reasonable" plan that is not necessarily the fastest.

The planner generates plans according to rules that recursively decompose a problem into simpler sub-problems. When the problem becomes "sufficiently simple," FFTW produces a plan that calls a fragment of optimized straight-line code that solves the problem directly. These fragments are called *codelets* in FFTW's lingo. You can envision a codelet as computing a "small" DFT, but many variations and special cases exist. For example, a codelet might be specialized to compute the DFT of real input (as opposed to complex). FFTW's speed depends therefore on two factors. First, the decomposition rules must produce a space of plans that is rich enough to contain "good" plans for most machines. Second, the codelets must be fast, since they ultimately perform all the real work.

FFTW's codelets are generated automatically by a special-purpose compiler called `genfft`. Most users do not interact with `genfft` at all: the standard FFTW distribution contains a set of about 150 pre-generated codelets that cover the most common uses. Users with special needs can use `genfft` to generate their own codelets. `genfft` is useful because of the following features. From a high-level mathematical description of a DFT algorithm, `genfft` derives an optimized implementation automatically. From a complex-DFT algo-

M. Frigo is with the IBM Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78758. He was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

S. G. Johnson is with the Massachusetts Institute of Technology, 77 Mass. Ave. Rm. 2-388, Cambridge, MA 02139. He was supported in part by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334.

rithm, `genfft` automatically derives an optimized algorithm for the real-input DFT. We take advantage of this property to implement real-data DFTs (Section VII), as well as to exploit machine-specific "SIMD" instructions (Section IX). Similarly, `genfft` automatically derives codelets for the discrete cosine (DCT) and sine (DST) transforms (Section VIII). We summarize `genfft` in Section VI, while a full description appears in [2].

We have produced three major implementations of FFTW, each building on the experience of the previous system. FFTW1 [3] (1997) introduced the idea of generating codelets automatically, and of letting a planner search for the best combination of codelets. FFTW2 (1998) incorporated a new version of `genfft` [2]. `genfft` did not change much in FFTW3 (2003), but the runtime structure was completely rewritten to allow for a much larger space of plans. This paper describes the main ideas common to all FFTW systems, the runtime structure of FFTW3, and the modifications to `genfft` since FFTW2.

Previous work on adaptive systems includes [3]–[11]. In particular, SPIRAL [9], [10] is another system focused on optimization of Fourier transforms and related algorithms, but it has distinct differences from FFTW. SPIRAL searches at compile-time over a space of mathematically equivalent formulas expressed in a "tensor-product" language, whereas FFTW searches at runtime over the formalism discussed in Section IV, which explicitly includes low-level details, such as strides and memory alignments, that are not as easily expressed using tensor products. SPIRAL generates machine-dependent code, whereas FFTW's codelets are machine-independent. FFTW's search uses *dynamic programming* [12, chapter 16], while the SPIRAL project has experimented with a wider range of search strategies including machine-learning techniques [13].

The remainder of this paper is organized as follows. We begin with a general overview of fast Fourier transforms in Section II. Then, in Section III, we compare the performance of FFTW and other DFT implementations. Section IV describes the space of plans explored by FFTW and how the FFTW planner works. Section V describes our experiences in the practical usage of FFTW. Section VI summarizes how `genfft` works. Section VII explains how FFTW computes DFTs of real data. Section VIII describes how `genfft` generates DCT and DST codelets, as well as how FFTW handles these transforms in the general case. Section IX tells how FFTW exploits SIMD instructions.

II. FFT Overview

The (forward, one-dimensional) discrete Fourier transform of an array X of n complex numbers is the array Y given by

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

where $0 \leq k < n$ and $\omega_n = \exp(-2\pi\sqrt{-1}/n)$. Implemented directly, Eq. (1) would require $\Theta(n^2)$ operations; fast Fourier transforms are $O(n \log n)$ algorithms to compute the same result. The most important FFT (and the one primarily used in FFTW) is known as the "Cooley-Tukey" algorithm, after the two authors who rediscovered and popularized it in 1965 [14], although it had been previously known as early as 1805 by Gauss as well as by later re-inventors [15]. The basic idea behind this FFT is that a DFT of a composite size $n = n_1 n_2$ can be re-expressed in terms of smaller DFTs of sizes n_1 and n_2—essentially, as a two-dimensional DFT of size $n_1 \times n_2$ where the output is *transposed*. The choices of factorizations of n, combined with the many different ways to implement the data re-orderings of the transpositions, have led to numerous implementation strategies for the Cooley-Tukey FFT, with many variants distinguished by their own names [16], [17]. FFTW implements a space of *many* such variants, as described later, but here we derive the basic algorithm, identify its key features, and outline some important historical variations and their relation to FFTW.

The Cooley-Tukey algorithm can be derived as follows. If n can be factored into $n = n_1 n_2$, Eq. (1) can be rewritten by letting $j = j_1 n_2 + j_2$ and $k = k_1 + k_2 n_1$. We then have:

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{j_1 k_1} \right) \omega_n^{j_2 k_1} \right] \omega_{n_2}^{j_2 k_2} \ . \tag{2}$$

Thus, the algorithm computes n_2 DFTs of size n_1 (the inner sum), multiplies the result by the so-called *twiddle factors* $\omega_n^{j_2 k_1}$, and finally computes n_1 DFTs of size n_2 (the outer sum). This decomposition is then continued recursively. The literature uses the term *radix* to describe an n_1 or n_2 that is bounded (often constant); the small DFT of the radix is traditionally called a *butterfly*.

Many well-known variations are distinguished by the radix alone. A *decimation in time* (*DIT*) algorithm uses n_2 as the radix, while a *decimation in frequency* (*DIF*) algorithm uses n_1 as the radix. If multiple radices are used, e.g. for n composite but not a prime power, the algorithm is called *mixed radix*. A peculiar blending of radix 2 and 4 is called *split radix*, which was proposed to minimize the count of arithmetic operations [16]. (Unfortunately, as we argue in this paper, minimal-arithmetic, fixed-factorization implementations tend to no longer be optimal on recent computer architectures.) FFTW implements both DIT and DIF, is mixed-radix with radices that are *adapted* to the hardware, and often uses much larger radices (radix-32 is typical) than were once common. (On the other end of the scale, a "radix" of roughly $\sqrt{n}$ has been called a *four-step* FFT [18], and we have found that one step of such a radix can be useful for large sizes in FFTW; see Section IV-D.1.)

A key difficulty in implementing the Cooley-Tukey FFT is that the n_1 dimension corresponds to discontiguous inputs j_1 in X but contiguous outputs k_1 in Y, and vice-versa for n_2. This is a matrix transpose for a single decomposition stage, and the composition of all such transpositions is a (mixed-base) digit-reversal permutation (or *bit-reversal*, for radix-2). The resulting necessity of discontiguous memory access and data re-ordering hinders efficient use of hierarchical memory architectures (e.g., caches), so that the optimal execution order of an FFT for given hardware is non-obvious, and various approaches have been proposed.

One ordering distinction is between recursion and iteration. As expressed above, the Cooley-Tukey algorithm could be thought of as defining a tree of smaller and smaller DFTs; for example, a textbook radix-2 algorithm would divide size n into two transforms of size $n/2$, which are divided into four transforms of size $n/4$, and so on until a base case is reached (in principle, size 1). This might naturally suggest a recursive implementation in which the tree is traversed "depth-first"—one size $n/2$ transform is solved completely before processing the other one, and so on. However, most traditional FFT implementations are non-recursive (with rare exceptions [19]) and traverse the tree "breadth-first" [17]—in the radix-2 example, they would perform n (trivial) size-1 transforms, then $n/2$ combinations into size-2 transforms, then $n/4$ combinations into size-4 transforms, and so on, thus making $\log_2 n$ passes over the whole array. In contrast, as we discuss in Section IV-D.1, FFTW3 employs an explicitly recursive strategy that encompasses *both* depth-first and breadth-first styles, favoring the former since it has some theoretical and practical advantages.

A second ordering distinction lies in how the digit-reversal is performed. The classic approach is a single, separate digit-reversal pass following or preceding the arithmetic computations. Although this pass requires only $O(n)$ time [20], it can still be non-negligible, especially if the data is out-of-cache; moreover, it neglects the possibility that data-reordering during the transform may improve memory locality. Perhaps the oldest alternative is the Stockham *auto-sort* FFT [17], [21], which transforms back and forth between two arrays with each butterfly, transposing one digit each time, and was popular to improve contiguity of access for vector computers [22]. Alternatively, an explicitly recursive style, as in FFTW, performs the digit-reversal implicitly at the "leaves" of its computation when operating out-of-place (Section IV-D.1). To operate in-place with $O(1)$ scratch storage, one can interleave small matrix transpositions with the butterflies [23]–[26], and a related strategy in FFTW is described by Section IV-D.3. FFTW can also perform intermediate re-orderings that blend its in-place and out-of-place strategies, as described in Section V-C.

Finally, we should mention that there are many FFTs entirely distinct from Cooley-Tukey. Three notable such algorithms are the *prime-factor algorithm* for $\gcd(n_1, n_2) = 1$ [27, page 619], along with Rader's [28] and Bluestein's [27], [29] algorithms for prime n. FFTW implements the first two in its codelet generator for hard-coded n (Section VI) and the latter two for general prime n. A new generalization of Rader's algorithm for prime-size *real-data* transforms is also discussed in Section VII. FFTW does not employ the Winograd FFT [30], which minimizes the number of multiplications at the expense of a large number of addditions. (This tradeoff is not beneficial on current processors that have specialized hardware multipliers.)

III. BENCHMARK RESULTS

We have performed extensive benchmarks of FFTW's performance, along with that of over 50 other FFT implementations, on most modern general-purpose processors, comparing complex and real-data transforms in one to three dimensions and for both single and double precisions. We generally found FFTW to be superior to other publicly available codes and comparable to vendor-tuned libraries. The complete results can be found at [1]. In this section, we present data for a small sampling of representative codes for complex-data one-dimensional transforms on a few machines.

We show the benchmark results as a series of graphs. Speed is measured in "MFLOPS," defined for a transform of size n as $(5n \log_2 n)/t$, where t is the time in μs for one transform, not including one-time initialization costs. This count of floating-point operations is based on the asymptotic number of operations for the radix-2 Cooley-Tukey algorithm (see [17, page 45]), although the actual count is lower for most DFT implementations. The MFLOPS measure should thus be viewed as a convenient scaling factor rather than as an absolute indicator of CPU performance.

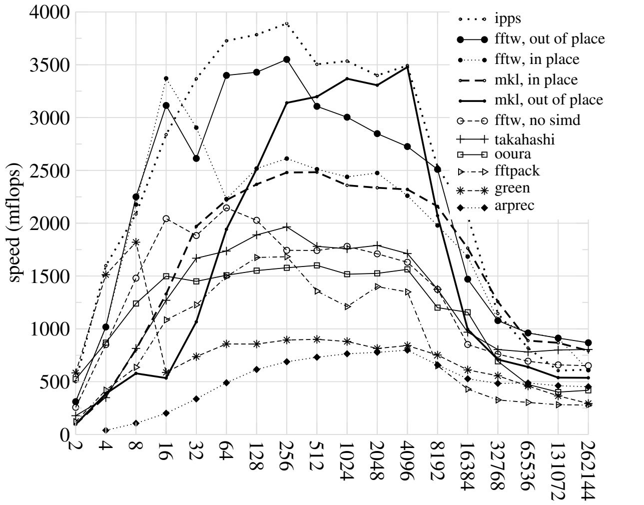


Fig. 1. Comparison of double-precision 1d complex DFTs, power-of-two sizes, on a 2.8 GHz Pentium IV. Intel C/Fortran compilers v. 7.1, optimization flags `-O3 -xW` (maximum optimization, enable automatic vectorizer).

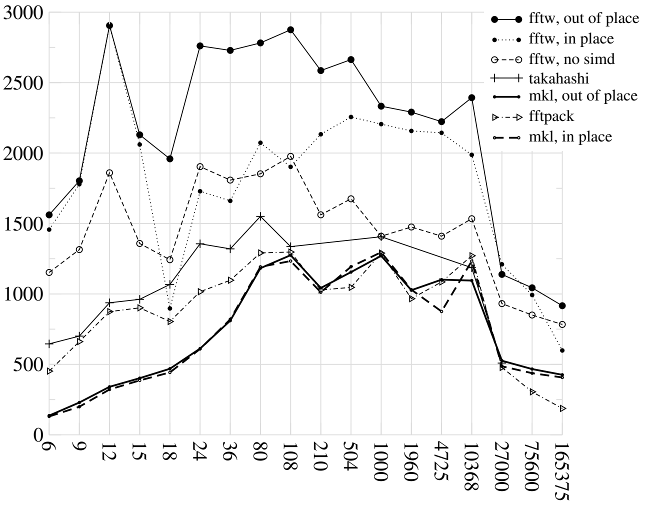


Fig. 2. Comparison of double-precision 1d complex DFTs, non-power-of-two sizes, on a 2.8 GHz Pentium IV. Compiler and flags as in Fig. 1.

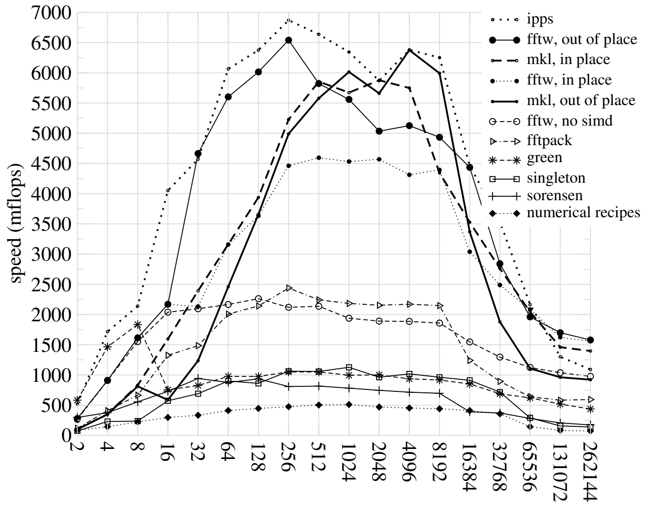


Fig. 3. Comparison of single-precision 1d complex DFTs, power-of-two sizes, on a 2.8 GHz Pentium IV. Compiler and flags as in Fig. 1. Note that fftpack, which was originally designed for vectorizing compilers (or vice versa), benefits somewhat from the automatic vectorization in this case.

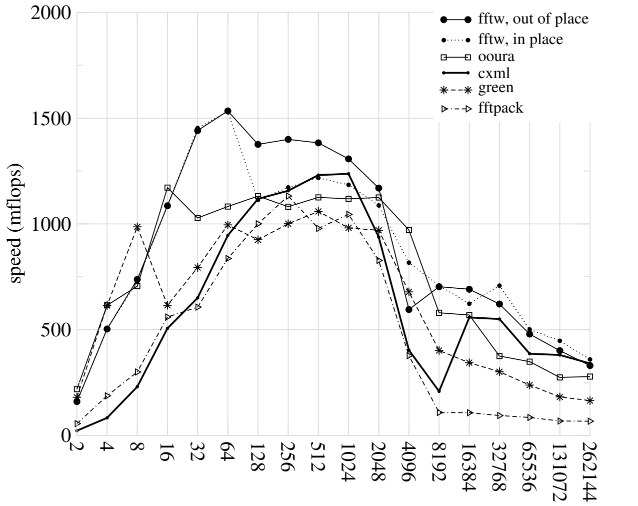


Fig. 5. Comparison of double-precision 1d complex DFTs, power-of-two sizes, on an 833 MHz Alpha EV6. Compaq C V6.2-505. Compaq Fortran X1.0.1-1155. Optimization flags: `-newc -w0 -O5 -ansi_alias -ansi_args -fp_reorder -tune host -arch host`.

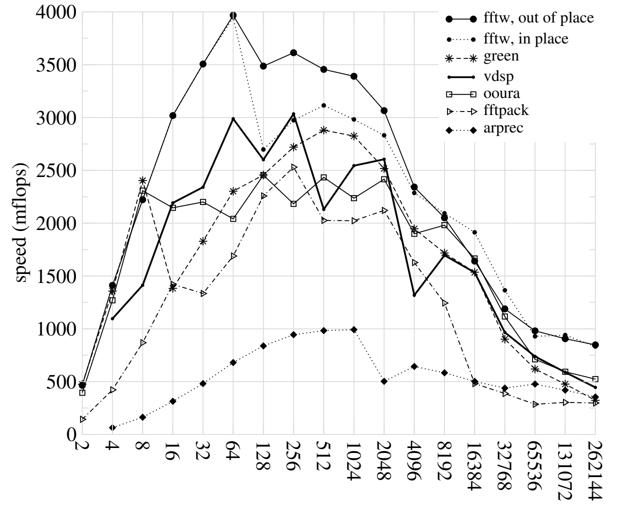


Fig. 4. Comparison of double-precision 1d complex DFTs, power-of-two sizes, on a 2 GHz PowerPC 970 (G5). Apple `gcc` v. 3.3, `g77` v. 3.4 20031105 (experimental). Optimization flags `-O3 -mcpu=970 -mtune=970`. The Apple vDSP library uses separate real/imaginary arrays to store complex numbers, and therefore its performance is not stricly comparable with the other codes, which use an array of real/imaginary pairs.

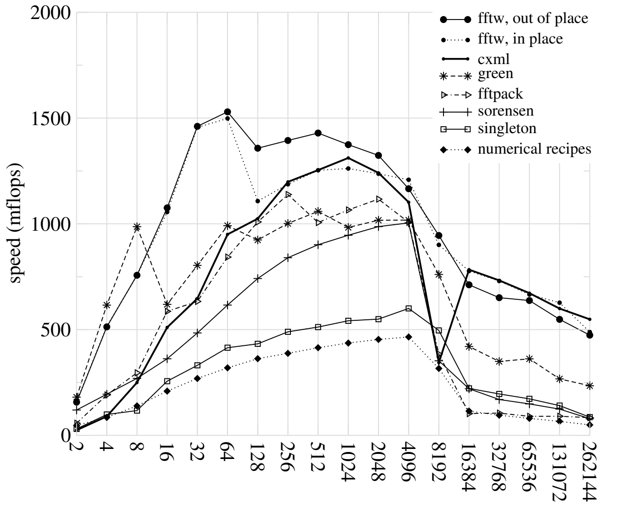


Fig. 6. Comparison of single-precision 1d complex DFTs, power-of-two sizes, on an 833 MHz Alpha EV6. Compilers and flags as in Fig. 5.

Fig. 1 shows the benchmark results for power-of-two sizes, in double precision, on a 2.8 GHz Pentium IV with the Intel compilers; in Fig. 2 are results for selected non-power-of-two sizes of the form $2^a 3^b 5^c 7^d$ on the same machine; in Fig. 3 are the single-precision power-of-two results. Note that only the FFTW, MKL (Intel), IPPS (Intel), and Takahashi libraries on this machine were specifically designed to exploit the SSE/SSE2 SIMD instructions (see Section IX); for comparison, we also include FFTW (out-of-place) with SIMD disabled ("fftw, no simd"). In Fig. 4 are the power-of-two double-precision results on a 2 GHz PowerPC 970 (G5) with the Apple `gcc` 3.3 compiler. In Fig. 5 are the power-of-two double-precision results on an 833 MHz Alpha EV6 with the Compaq compilers, and in Fig. 6 are the single-precision results on the same machine.

In addition to FFTW v. 3.0.1, the other codes benchmarked are as follows (some for only one precision or machine): *arprec*, "four-step" FFT implementation [18] (from the C++ ARPREC library, 2002); *cxml*, the vendor-tuned Compaq Extended Math Library on Alpha; *fftpack*, the Fortran library from [22]; *green*, free code by J. Green (C, 1998); *mkl*, the Intel Math Kernel Library v. 6.1 (DFTI interface) on the Pentium IV; *ipps*, the Intel Integrated Performance Primitives, Signal Processing, v. 3.0 on the Pentium IV; *numerical recipes*, the C `four1` routine from [31]; *ooura*, a free code by T. Ooura (C and Fortran, 2001); *singleton*, a Fortran FFT [32]; *sorensen*, a split-radix FFT [33]; *takahashi*, the FFTE library v. 3.2 by D. Takahashi (Fortran, 2004) [34]; and *vdsp*, the Apple vDSP library on the G5.

We now offer some remarks to aid the interpretation of the performance results. The performance of all routines drops for large problems, reflecting the cache hierarchy of the machine. Performance is low for small problems as well, because of the overhead of calling a routine to do little work. FFTW is the only library that exploits SIMD instructions for non-power-of-two sizes, which gives it an advantage on the Pentium IV for this case. IPPS is limited to in-place contiguous inputs, whereas MKL and FFTW allow for strided input. Assuming contiguous input gives some speed advantage on a machine such as the Pentium IV where index computation is somewhat slow.

IV. The structure of FFTW3

In this section, we discuss in detail how FFTW works. Specifically, we discuss how FFTW represents the problem to be solved (Sections IV-A and IV-B), the set of plans that the planner considers during its search (Sections IV-C and IV-D), and the internal operation of the planner (Section IV-E). For simplicity, this section considers complex DFTs only; we discuss real DFTs in Section VII.

Of these components, the representation of the problem to be solved is a critical choice. Indeed, we view our definition of a "problem" as a fundamental contribution of this paper. Because only problems that can be expressed can be solved, the representation of a problem determines an upper bound to the space of plans that the planner can explore, and therefore it ultimately constrains FFTW's performance.

A. Representation of problems in FFTW

DFT problems in FFTW are expressed in terms of structures called I/O tensors, which in turn are described in terms of ancillary structures called I/O dimensions. (I/O tensors are unrelated to the tensor-product notation of SPIRAL.) In this section, we define these terms precisely.

An *I/O dimension* d is a triple $d = (n, \iota, o)$, where n is a nonnegative integer called the *length*, ι is an integer called the *input stride*, and o is an integer called the *output stride*. An *I/O tensor* $t = \{d_1, d_2, \ldots, d_\rho\}$ is a set of I/O dimensions. The nonnegative integer $\rho = |t|$ is called the *rank* of the I/O tensor. A *DFT problem*, denoted by $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, consists of two I/O tensors $\mathbf{N}$ and $\mathbf{V}$, and of two *pointers* $\mathbf{I}$ and $\mathbf{O}$. Roughly speaking, this describes $|\mathbf{V}|$ nested loops of $|\mathbf{N}|$-dimensional DFTs with input data starting at memory location $\mathbf{I}$ and output data starting at $\mathbf{O}$. We now give a more precise definition by induction on $|\mathbf{V}|$, yielding a set of assignments from input to output. Conceptually, all of the right-hand sides of these assignments are evaluated before writing their values to the left-hand sides, a fiction that defines the behavior precisely, e.g., when $\mathbf{I} = \mathbf{O}$. (See also the examples in Section IV-B.)

$\mathrm{dft}(\mathbf{N}, \{\}, \mathbf{I}, \mathbf{O})$, with $\rho = |\mathbf{N}|$, is the ρ-dimensional DFT, defined as follows. Let $\mathbf{N} = \{(n_\ell, \iota_\ell, o_\ell) \mid 1 \le \ell \le \rho\}$; for all output indices $0 \le k_\ell < n_\ell$, yield the assignment

$$\mathbf{O}\left[\sum_{\ell=1}^{\rho} k_\ell \cdot o_\ell\right] := \sum_{j_1,\ldots,j_\rho} \mathbf{I}\left[\sum_{\ell=1}^{\rho} j_\ell \cdot \iota_\ell\right] \prod_{\ell=1}^{\rho} \omega_{n_\ell}^{j_\ell k_\ell} ,$$

where each input index j_ℓ is summed from 0 to $n_\ell - 1$, ω_n is a primitive n-th root of unity as in Section II, and $\mathbf{X}[k]$ denotes the complex number at memory location $\mathbf{X} + k$ (with pointer arithmetic in units of complex numbers). By convention, we define the zero-dimensional problem $\mathrm{dft}(\{\}, \{\}, \mathbf{I}, \mathbf{O})$ to yield the assignment $\mathbf{O}[0] := \mathbf{I}[0]$.

$\mathrm{dft}(\mathbf{N}, \{(n, \iota, o)\} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$ is recursively defined as a "loop" of n problems: for all $0 \le k < n$, yield all assignments in $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I} + k \cdot \iota, \mathbf{O} + k \cdot o)$.

If two assignments write to the same memory location, the DFT problem is undefined. Such nonsensical problems are not normally encountered in practice, however, as discussed in Section IV-B.

One property of this definition is the fact that an I/O tensor t is equivalent to $t \cup \{(1, \iota, o)\}$. That is, length-1 DFT dimensions and length-1 loops can be eliminated. FFTW therefore internally canonicalizes I/O tensors by removing all I/O dimensions where $n = 1$. (Similarly, all I/O tensors of the form $t \cup \{(0, \iota, o)\}$ are equivalent.)

We call $\mathbf{N}$ the *size* of the problem. The *rank* of a problem is defined to be the rank of its size (i.e., the dimensionality of the DFT). Similarly, we call $\mathbf{V}$ the *vector size* of the problem, and the *vector rank* of a problem is correspondingly defined to be the rank of its vector size. One unusual feature of FFTW is that the vector rank is arbitrary: FFTW is not restricted to vector sizes of rank 1. Intuitively, the vector size can be interpreted as a set of "loops" wrapped around a single DFT, and we therefore refer to a single I/O dimension of $\mathbf{V}$ as a *vector loop*. (Alternatively, one can view the problem as defining a multi-dimensional DFT over a vector space.) The problem does not specify the order of execution of these loops, however, and therefore FFTW is free to choose the fastest or most convenient order.

An I/O tensor for which $\iota_k = o_k$ for all k is said to be *in-place*. Occasionally, the need arises to replace input strides with output strides and vice versa. We define $\text{copy-i}(t)$ to be the I/O tensor $\{(n, \iota, \iota) \mid (n, \iota, o) \in t\}$. Similarly, we define $\text{copy-o}(t)$ to be the I/O tensor $\{(n, o, o) \mid (n, \iota, o) \in t\}$.

The two pointers $\mathbf{I}$ and $\mathbf{O}$ specify the memory addresses of the input and output arrays, respectively. If $\mathbf{I} = \mathbf{O}$, we say that the problem is *in-place*, otherwise the problem is *out-of-place*. FFTW uses explicit pointers for three reasons. First, we can distinguish in-place from out-of-place problems, which is important because many FFT algorithms are inherently either in-place or out-of-place, but not both. Second, SIMD instructions usually impose constraints on the memory alignment of the data arrays; from the pointer, FFTW determines whether SIMD instructions are applicable. Third, performance may depend on the actual memory address of the data, in addition to the data layout, so an explicit pointer is in principle necessary for maximum performance.

B. DFT problem examples

The I/O tensor representation is sufficiently general to cover many situations that arise in practice, including some that are not usually considered to be instances of the DFT. We consider a few examples here.

An $n_1 \times n_2$ two-dimensional matrix is typically stored in C using *row-major* format: size-n_2 contiguous arrays for each row, stored as n_1 consecutive blocks starting from a pointer **I/O** (for input/output). This memory layout is described by the in-place I/O tensor $\mathbf{X} = \{(n_1, n_2, n_2), (n_2, 1, 1)\}$. Performing the $n_1 \times n_2$ two-dimensional DFT of this array corresponds to the rank-2, vector-rank-0 problem: $\mathrm{dft}(\mathbf{X}, \{\}, \mathbf{I}, \mathbf{O})$. The transform data can also be non-contiguous; for example, one could transform an $n_1 \times n_2'$ subset of the matrix, with $n_2' \leq n_2$, starting at the upper-left corner, by: $\mathrm{dft}(\{(n_1, n_2, n_2), (n_2', 1, 1)\}, \{\}, \mathbf{I}, \mathbf{O})$.

Another possibility is the rank-1, vector-rank-1 problem that performs a "loop" of n_1 one-dimensional DFTs of size n_2 operating on all the contiguous *rows* of the matrix: $\mathrm{dft}(\{(n_2, 1, 1)\}, \{(n_1, n_2, n_2)\}, \mathbf{I}, \mathbf{O})$. Conversely, to perform one-dimensional DFTs of the (discontiguous) *columns* of the matrix, one would use: $\mathrm{dft}(\{(n_1, n_2, n_2)\}, \{(n_2, 1, 1)\}, \mathbf{I}, \mathbf{O})$; if $n_2 = 3$, for example, this could be thought of as the size-n_1 one-dimensional DFT of a three-component "vector field" (with "vector components" stored contiguously).

Additionally, the rank-0, vector-rank-2 problem $\mathrm{dft}(\{\}, \mathbf{X}, \mathbf{I}, \mathbf{O})$ denotes a copy (loop of rank-0 DFTs) of $n_1 n_2$ complex numbers from $\mathbf{I}$ to $\mathbf{O}$. (If $\mathbf{I} = \mathbf{O}$, the runtime cost of this copy is zero.) Morever, this is equivalent to the problem $\mathrm{dft}(\{\}, \{(n_1 n_2, 1, 1)\}, \mathbf{I}, \mathbf{O})$—it is possible to combine vector loops that, together, denote a constant-offset sequence of memory locations, and FFTW thus canonicalizes all such vector loops internally.

Generally, rank-0 transforms may describe some in-place permutation, such as a matrix transposition, if $\mathbf{I} = \mathbf{O}$. For example, to transpose the $n_1 \times n_2$ matrix to $n_2 \times n_1$, both stored in row-major order starting at $\mathbf{I}$, one would use the rank-0, vector-rank-2 problem: $\mathrm{dft}(\{\}, \{(n_1, n_2, 1), (n_2, 1, n_1)\}, \mathbf{I}, \mathbf{I})$ (these two vector loops *cannot* be combined into a single loop).

Finally, one can imagine problems where the different DFTs in the vector loop or a multi-dimensional transform operate on overlapping data. For example, the "two-dimensional" $\mathrm{dft}(\{(n_1, 1, 1), (n_2, 1, 1)\}, \{\}, \mathbf{I}, \mathbf{O})$ transforms a "matrix" whose subsequent rows overlap in $n_2 - 1$ elements. The behavior of FFTW is undefined in such cases, which are, in any case, prohibited by the ordinary user interface (Section V-A).

C. The space of plans in FFTW

The FFTW planner, when given a problem, explores a space of valid plans for that problem and selects the plan (a particular composition of algorithmic steps in a specified order of execution) that happens to execute fastest. Many plans exist that solve a given problem, however. Which plans does FFTW consider, exactly? This section addresses this and related questions.

Roughly speaking, to solve a general DFT problem, one must perform three tasks. First, one must reduce a problem of arbitrary vector rank to a set of loops nested around a problem of vector rank 0, i.e., a single (possibly multi-dimensional) DFT. Second, one must reduce the multi-dimensional DFT to a sequence of of rank-1 problems, i.e., one-dimensional DFTs. Third, one must solve the rank-1, vector rank-0 problem by means of some DFT algorithm such as Cooley-Tukey. These three steps need not be executed in the stated order, however, and in fact, almost every permutation and interleaving of these three steps leads to a correct DFT plan. The choice of the set of plans explored by the planner is critical for the usability of the FFTW system: the set must be large enough to contain the fastest possible plans, but it must be small enough to keep the planning time acceptable.

The remainder of this section enumerates the class of plans considered by current FFTW planner. This particular set of plans is reasonably simple, it can express a wide variety of algorithms, and it seems to perform well on most architectures. We do not claim that this set is the absolute optimum: many more possibilities exist that are a topic of future research, and the space of plans will likely change in future FFTW releases. The plans that we now describe usually perform some simple "atomic" operation, and it may not be apparent how these operations fit together to actually compute DFTs, or why certain operations are useful at all. We shall discuss these matters in Section IV-D. For now, we ask for the reader's patience while we describe the precise set of plans generated by FFTW.

1) No-op plans: The simplest plans are those that do nothing. FFTW generates no-op plans for problems $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ in the following two cases:

- when $\mathbf{V} = \{(0, \iota, o)\}$, that is, no data is to be transformed; or
- when $\mathbf{N} = \{\}$, $\mathbf{I} = \mathbf{O}$, and the I/O tensor $\mathbf{V}$ is in-place. In this case, the transform reduces to a copy of the input array into itself, which requires no work.

It is possible for the user to specify a no-op problem if one is desired (FFTW solves it really quickly). More often, however, no-op problems are generated by FFTW itself as a by-product of buffering plans. (See Section IV-C.7.)

2) Rank-0 plans: The rank-0 problem $\mathrm{dft}(\{\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ denotes a permutation of the input array into the output array. FFTW does not solve arbitrary rank-0 problems, only the following two special cases that arise in practice.

- When $|\mathbf{V}| = 1$ and $\mathbf{I} \neq \mathbf{O}$, FFTW produces a plan that copies the input array into the output array. Depending on the strides, the plan consists of a loop or, possibly, of a call to the ANSI C function `memcpy`, which is specialized to copy contiguous regions of memory. (The case $\mathbf{I} = \mathbf{O}$ is discussed in Section IV-C.1.)
- When $|\mathbf{V}| = 2$, $\mathbf{I} = \mathbf{O}$, and the strides denote a matrix-transposition problem, FFTW creates a plan that transposes the array in-place. FFTW implements the square transposition $\mathrm{dft}(\{\}, \{(n, \iota, o), (n, o, \iota)\}, \mathbf{I}, \mathbf{O})$ by means of the "cache-oblivious" algorithm from [35], which is fast and, in theory, uses the cache optimally regardless of the cache size. A generalization of this idea is employed for non-square transpositions with a large common factor or a small difference between the dimensions [36], and otherwise the algorithm from [37] is used.

An important rank-0 problem that is describable but not currently solvable in-place by FFTW is the general in-place digit-reversal permutation [20], which could be used for some DFT algorithms.

3) Rank-1 plans: Rank-1 DFT problems denote ordinary one-dimensional Fourier transforms. FFTW deals with most rank-1 problems as follows. (Other kinds of rank-1 plans exist, which apply in certain special cases such as DFTs of prime size. See Section IV-C.7.)

a) Direct plans: When the DFT rank-1 problem is "small enough," FFTW produces a *direct plan* that solves the problem directly. This situation occurs for problems $\mathrm{dft}(\{(n, \iota, o)\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $|\mathbf{V}| \leq 1$ and $n \in \{2, \ldots, 16, 32, 64\}$. These plans operate by calling a fragment of C code (a *codelet*) specialized to solve problems of one particular size. In FFTW, codelets are generated automatically by `genfft`, but it is possible for a user to add hand-written machine-specific codelets if desired.

We impose the restriction that $|\mathbf{V}| \leq 1$ because of engineering tradeoffs. Informally speaking, a codelet for $|\mathbf{V}| = 0$ consists of straight-line code, while a codelet for $|\mathbf{V}| = 1$ consists of a vector loop wrapped around straight-line code. Either codelets implement the loop or they don't—allowing for both possibilities would require the duplication of the whole set of codelets. In practice, $|\mathbf{V}| = 1$ is more common than $|\mathbf{V}| = 0$, and therefore FFTW takes the position that all direct problems have vector rank 1, converting the rank-0 I/O tensor $\{\}$ into the rank-1 I/O tensor $\{(1, 0, 0)\}$. We have not investigated the performance implications of codelets of higher vector rank. For now, FFTW handles the general vector-rank case via Section IV-C.5.

b) Cooley-Tukey plans: For problems of the form $\mathrm{dft}(\{(n, \iota, o)\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $n = rm$, FFTW generates a plan that implements a radix-r Cooley-Tukey algorithm (Section II). (FFTW generates a plan for each suitable value of r, possibly in addition to a direct plan. The planner then selects the fastest.)

Of the many known variants of the Cooley-Tukey algorithm, FFTW implements the following two, distinguished mainly by whether the codelets multiply their inputs or outputs by twiddle factors. (Again, if both apply, FFTW tries both.) As for direct plans, we restrict $|\mathbf{V}|$ to be ≤ 1 because of engineering tradeoffs. (In the following, we use n_1 and n_2 from Eq. (2).)

A *decimation in time* (*DIT*) plan uses a *radix* $r = n_2$ (and thus $m = n_1$): it first solves $\mathrm{dft}(\{(m, r \cdot \iota, o)\}, \mathbf{V} \cup \{(r, \iota, m \cdot o)\}, \mathbf{I}, \mathbf{O})$, then multiplies the output array $\mathbf{O}$ by the twiddle factors, and finally solves $\mathrm{dft}(\{(r, m \cdot o, m \cdot o)\}, \mathbf{V} \cup \{(m, o, o)\}, \mathbf{O}, \mathbf{O})$. For performance, the last two steps are not planned independently, but are fused together in a single "twiddle" codelet—a fragment of C code that multiplies its input by the twiddle factors and performs a DFT of size r, operating in-place on $\mathbf{O}$. FFTW contains one such codelet for each $r \in \{2, \ldots, 16, 32, 64\}$.

A *decimation in frequency* (*DIF*) plan uses $r = n_1$ (and thus $m = n_2$); it operates backwards with respect to a DIT plan. The plan first solves $\mathrm{dft}(\{(r, m \cdot \iota, m \cdot \iota)\}, \mathbf{V} \cup \{(m, \iota, \iota)\}, \mathbf{I}, \mathbf{I})$, then multiplies the input array $\mathbf{I}$ by the twiddle factors, and finally solves $\mathrm{dft}(\{(m, \iota, r \cdot o)\}, \mathbf{V} \cup \{(r, m \cdot \iota, o)\}, \mathbf{I}, \mathbf{O})$. Again, for performance, the first two steps are fused together in a single codelet. Because DIF plans destroy the input array, however, FFTW generates them only if $\mathbf{I} = \mathbf{O}$ or if the user explicitly indicates that the input can be destroyed. DIF plans that do not destroy the input could be devised, but we did not implement them because our main use of DIF plans is for in-place transforms (Section IV-D.3).

4) Plans for higher ranks: These plans reduce a multi-dimensional DFT problem to problems of lower rank, which are then solved recursively.

Formally, to solve $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, where $\mathbf{N} = \mathbf{N}_1 \cup \mathbf{N}_2$, $|\mathbf{N}_1| \geq 1$ and $|\mathbf{N}_2| \geq 1$, FFTW generates a plan that first solves $\mathrm{dft}(\mathbf{N}_1, \mathbf{V} \cup \mathbf{N}_2, \mathbf{I}, \mathbf{O})$, and then solves $\mathrm{dft}(\text{copy-o}(\mathbf{N}_2), \text{copy-o}(\mathbf{V} \cup \mathbf{N}_1), \mathbf{O}, \mathbf{O})$.

In principle, FFTW generates a plan for every suitable choice of the subsets $\mathbf{N}_1$ and $\mathbf{N}_2$, but in practice we impose certain restrictions on the possible choices in order to reduce the planning time. (See Section V-B.) A typical heuristic is to choose two sub-problems $\mathbf{N}_1$ and $\mathbf{N}_2$ of roughly equal rank, where each input stride in $\mathbf{N}_1$ is smaller than any input stride in $\mathbf{N}_2$.

5) Plans for higher vector ranks: These plans extract a vector loop to reduce a DFT problem to a problem of lower vector rank, which is then solved recursively.

Formally, to solve $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, where $\mathbf{V} = \{(n, \iota, o)\} \cup \mathbf{V}_1$, FFTW generates a loop that, for all k such that $0 \leq k < n$, invokes a plan for $\mathrm{dft}(\mathbf{N}, \mathbf{V}_1, \mathbf{I} + k \cdot \iota, \mathbf{O} + k \cdot o)$.

Any of the vector loops of $\mathbf{V}$ could be extracted in this way, leading to a number of possible plans. To reduce the loop permutations that the planner must consider, however, FFTW only considers the vector loop that has either the smallest or the largest ι; this often corresponds to the smallest or largest o as well, or commonly vice versa (which makes the best loop order nonobvious).

6) Indirect plans: Indirect plans transform a DFT problem that requires some data shuffling (or discontiguous operation) into a problem that requires no shuffling plus a rank-0 problem that performs the shuffling.

Formally, to solve $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $|\mathbf{N}| > 0$, FFTW generates a plan that first solves $\mathrm{dft}(\{\}, \mathbf{N} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$, and then solves $\mathrm{dft}(\text{copy-o}(\mathbf{N}), \text{copy-o}(\mathbf{V}), \mathbf{O}, \mathbf{O})$. This plan first rearranges the data, then solves the problem in place. If the problem is in-place or the user has indicated that the input can be destroyed, FFTW also generates a dual plan: first solve $\mathrm{dft}(\text{copy-i}(\mathbf{N}), \text{copy-i}(\mathbf{V}), \mathbf{I}, \mathbf{I})$, and then solve $\mathrm{dft}(\{\}, \mathbf{N} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$ (solve in place, then rearrange).

7) Other plans: For completeness, we now briefly mention the other kinds of plans that are implemented in FFTW.

Buffering plans solve a problem out-of-place to a temporary buffer and then copy the result to the output array. These plans serve two purposes. First, it may be inconvenient or impossible to solve a DFT problem without using extra memory space, and these plans provide the necessary support for these cases (e.g. in-place transforms). Second, if the input/output arrays are noncontiguous in memory, operating on a contiguous buffer might be faster because of better interaction with caches and the rest of the memory subsystem. Similarly, *buffered DIT* (or DIF) plans apply the twiddle codelets of Section IV-C.3.b by copying a batch of inputs to a contiguous buffer, executing the codelets, and copying back.

Generic plans implement a naive $\Theta(n^2)$ algorithm to solve one-dimensional DFTs. Similarly, *Rader plans* implement the size-30 DFT, depth-first:

- loop 3
  - size-5 direct codelet, vector size 2
  - size-2 twiddle codelet, vector size 5
- size-3 twiddle codelet, vector size 10

size-30 DFT, breadth-first:

- loop 3
  - size-5 direct codelet, vector size 2
- loop 3
  - size-2 twiddle codelet, vector size 5
- size-3 twiddle codelet, vector size 10

Fig. 7. Two possible decompositions for a size-30 DFT, both for the arbitrary choice of DIT radices 3 then 2 then 5, and prime-size codelets. Items grouped by a "{" result from the plan for a single sub-problem. In the depth-first case, the vector rank was reduced to 0 as per Section IV-C.5 before decomposing sub-problems, and vice-versa in the breadth-first case.

algorithm from [28] to compute one-dimensional DFTs of prime size in $O(n \log n)$ time (with *Rader-DIT plans* for the twiddled DFTs of large prime factors). (A future release of FFTW also implements Bluestein's "chirp-z" algorithm [27], [29].)

Real/imaginary plans execute a vector loop of two specialized real-input DFT plans (Section VII) on the real and imaginary parts of the input, and then combine the results. This can be more efficient if, for example, the real and imaginary parts are stored by the user in separate arrays (a generalization of the storage format that we omitted above).

Parallel (multi-threaded) plans are achieved by a special variant of Section IV-C.5 that executes the vector loop in parallel, along with a couple of extra plans to execute twiddle-codelet loops in parallel. Although shared- and distributed-memory parallel versions of FFTW exist, we do not further describe them in this paper.

D. Discussion

Although it may not be immediately apparent, the combination of the recursive rules in Section IV-C can produce a number of useful algorithms. To illustrate these compositions, we discuss in particular three issues: depth- vs. breadth-first, loop reordering, and in-place transforms. More possibilities and explicit examples of plans that are "discovered" in practice are discussed in Section V-C.

1) Depth-first and breadth-first FFTs: If one views an FFT algorithm as a directed acyclic graph (dag) of data dependencies (e.g. the typical "butterfly" diagram), most traditional Cooley-Tukey FFT implementations traverse the tree in "breadth-first" fashion (Section II). In contrast, FFTW1 and FFTW2 traversed the dag in "depth-first" order, due to their explicitly recursive implementation. That is, they completely solved a single one-dimensional sub-DFT before moving on to the next. FFTW3 also evaluates its plans in an explicitly recursive fashion, but, because its problems now include arbitrary vector ranks, it is able to express both depth- and breadth-first traversal of the dag (as well as intermediate styles). This is illustrated by an example in Fig. 7 and discussed further below.

Depth-first traversal has theoretical advantages for cache utilization: eventually, the sub-DFT will fit into cache and (ideally) require no further cache misses [2], [3], [19], [35], regardless of the size of the cache. (Although we were initially motivated, in part, by these results, the point of FFTW's self-optimization is that we need not rely on this or any similar prediction.) Technically, the asymptotically optimal "cache-oblivious" recursive algorithm would use a radix of $\Theta(\sqrt{n})$ for a transform of size n, analogous to the "four-step" algorithm [18], [38], but we have found that a bounded radix generally works better in practice, except for at most a single step of radix-$\sqrt{n}$.

A depth-first style is also used for the multi-dimensional plans of Section IV-C.4, where in this case the planner can (and often does) choose the optimal cache-oblivious algorithm: it breaks the transform into sub-problems of roughly equal rank. In contrast, an iterative, "breadth-first" approach might perform all of the 1d transforms for the first dimension, then all of the 1d transforms for the second dimension, and so on, which has extremely poor cache performance compared to grouping the dimensions into smaller multi-dimensional transforms.

Because its sub-problems contain a vector loop that can be executed in a variety of orders, however, FFTW3 can also express breadth-first traversal. For example, if the rule of Section IV-C.4 were applied repeatedly to first reduce the rank to 1, and *then* the vector ranks were reduced by applying the loop rule of Section IV-C.5 to the sub-problems, the plan would implement the breadth-first multi-dimensional approach described above. Similarly, a 1d algorithm resembling the traditional breadth-first Cooley-Tukey would result from applying Section IV-C.3.b to completely factorize the problem size before applying the loop rule to reduce the vector ranks. As described in Section V-B, however, by default we limit the types of breadth-first-style plans considered in order to reduce planner time, since they appear to be suboptimal in practice as well as in theory.

Even with the breadth-first execution style described above, though, there is still an important difference between FFTW and traditional iterative FFTs: FFTW has no separate bit-reversal stage. For out-of-place transforms, the re-ordering occurs implicitly in the strides of Section IV-C.3.b (which are transferred to the strides of the nested vector loops in a recursive breadth-first plan); in any case, the "leaves" of the recursion (direct plans) transform the input directly to its correct location in the output, while the twiddle codelets operate in-place. This is an automatic benefit of a recursive implementation. (Another possibility would be a Stockham-style transform, from Section II, but this is not currently implemented in FFTW.)

2) Vector recursion: Another example of the effect of loop reordering is a style of plan that we sometimes call *vector recursion* (unrelated to "vector-radix" FFTs [16]). The basic idea is that, if you have a loop (vector-rank 1) of transforms, where the vector stride is smaller than the transform size, it is advantageous to push the loop towards the leaves of the transform decomposition, while otherwise maintaining recursive depth-first ordering, rather than looping "outside" the transform; i.e., apply the usual FFT to "vectors" rather than numbers. Limited forms of this idea have appeared for computing multiple FFTs on vector processors (where the loop in question maps directly to a hardware vector) [22], and in another restricted form as an undocumented feature of FFTW2. Such plans are among the many possible compositions of our recursive rules: one or more steps of the Cooley-Tukey decomposition (Section IV-C.3.b) can execute before the low-stride vector loop is extracted (Section IV-C.5), but with other loops still extracted before decomposition. The low-stride vector loop need not, however, be pushed all the way to the leaves of the decomposition, and it is not unusual for the loop to be executed at some intermediate level instead.

For example, low-stride vector loops appear in the decomposition of a typical multi-dimensional transform (Section IV-C.4): along some dimensions, the transforms are contiguous (stride 1) but the vector loop is not, while along other dimensions the vector stride is 1 but the transforms are discontiguous, and in this latter case vector recursion is often preferred. As another example, Cooley-Tukey itself produces a unit *input*-stride vector loop at the top-level DIT decomposition, but with a large *output* stride; this difference in strides makes it nonobvious whether vector recursion is advantageous for the sub-problem, but for large transforms we often observe the planner to choose this possibility.

3) In-place plans: In-place 1d transforms can be obtained by two routes from the possibilities described in Section IV-C: via combination of DIT and DIF plans (Section IV-C.3.b) with transposes (Section IV-C.2), or via buffering (Section IV-C.7).

The transpose-based strategy for an in-place transform of size pqm is outlined as follows. First, the transform is decomposed via a radix-p DIT plan into a vector of p transforms of size qm, then these are decomposed in turn by a radix-q DIF plan into a vector (rank 2) of $p \times q$ transforms of size m. These transforms of size m have input and output at different places/strides in the original array, and so cannot be solved independently. Instead, an indirect plan (Section IV-C.6) is used to express the sub-problem as pq in-place transforms of size m, followed or preceded by an $m \times p \times q$ rank-0 transform. The latter sub-problem is easily seen to be m in-place $p \times q$ transposes (ideally square, i.e. $p = q$). Related strategies for in-place transforms based on small transposes were described in [23]–[26]; alternating DIT/DIF, without concern for in-place operation, was also considered in [39], [40].

As an optimization, we include *DIF-transpose codelets* that combine the radix-q DIF twiddle codelet (in a loop of length p) with the $p \times q$ transpose, for $p = q \in \{2, 3, 4, 5, 6, 8\}$. (DIF-transpose is to DIF + transpose roughly as [24] is to [25].) Another common special case is where $m = 1$, in which a size-q direct plan (Section IV-C.3.a), not a DIF codelet, is required (the twiddle factors are unity), and the transposes are performed at the leaves of the plan.

Since the size-m transforms must be performed in-place, if they are too large for a direct plan the transpose scheme can be used recursively *or* a buffered plan can be used for this sub-problem. That is, a mixture of these two strategies can be employed. We emphasize that all of these algorithms are "discovered" automatically by the planner simply by composing the rules of Section IV-C.

E. The FFTW planner

In this section, we discuss the implementation and operation of the FFTW planner.

The FFTW planner is a modular piece of code independent of the specific problems and plans supported by the system. In this way, we can reuse the same planner for complex DFTs, real-data DFTs, and other transforms. The separation between planner and plans is achieved by means of ancillary entities called *solvers*, which can be viewed as the portion of the planner that is problem- and plan-specific. The choreography of the planner, solvers, and plans is arranged as follows.

The planner is first initialized with a list of solvers. Given a problem, the planner calls each solver in sequence, requesting a plan for the problem. Each solver returns either a pointer to a plan or a null pointer, which indicates that the solver cannot create a plan for that problem. The planner selects the fastest plan (by performing explicit time measurements) and returns it to the user. The user calls the plan to compute Fourier transforms as desired.

A solver can generate a certain class of plans. (Approximately, one solver exists for each item in the classification of plans from Section IV-C.) When invoked by the planner, a solver creates the plan for the given problem (if possible) and it initializes any auxiliary data required by the plan (such as twiddle factors). In many cases, creating a plan requires that a plan for one or more sub-problems be available. For example, Cooley-Tukey plans require a plan for a smaller DFT. In these cases, the solver obtains the sub-plans by invoking the planner recursively.

By construction, the FFTW planner uses *dynamic programming* [12, chapter 16]: it optimizes each sub-problem locally, independently of the larger context. Dynamic programming is not guaranteed to find the fastest plan, because the performance of plans is context-dependent on real machines: this is another engineering tradeoff that we make for the sake of planning speed. The representation of problems discussed in Section IV-A is well suited to dynamic programming, because a problem encodes all the information required to solve it—no reference to a larger context is necessary.

Like most dynamic-programming algorithms, the planner potentially evaluates the same sub-problem multiple times. To avoid this duplication of work, the FFTW planner uses the standard solution of *memoization*: it keeps a table of plans for already computed problems and it returns the solution from the table whenever possible. Memoization is accomplished by FFTW in a slightly unorthodox fashion, however. The memoization table, which maps problems into plans, contains neither problems nor plans, because these data structures can be large and we wish to conserve memory. Instead, the planner stores a 128-bit hash of the problem and a pointer to the solver that generated the plan in the first place. When the hash of a problem matches a hash key in the table, the planner invokes the corresponding solver to obtain a plan. For hashing, we

```
fftw_plan plan;
fftw_complex in[n], out[n];

/* plan a 1d forward DFT: */
plan = fftw_plan_dft_1d(n, in, out,
          FFTW_FORWARD, FFTW_PATIENT);
```

Initialize `in[]` *with some data...*

```
fftw_execute(plan); // compute DFT
```

Write some new data to `in[]` ...

```
fftw_execute(plan); // reuse plan
```

Fig. 8. Example of FFTW's use. The user must first create a plan, which can be then used for many transforms of the same size.

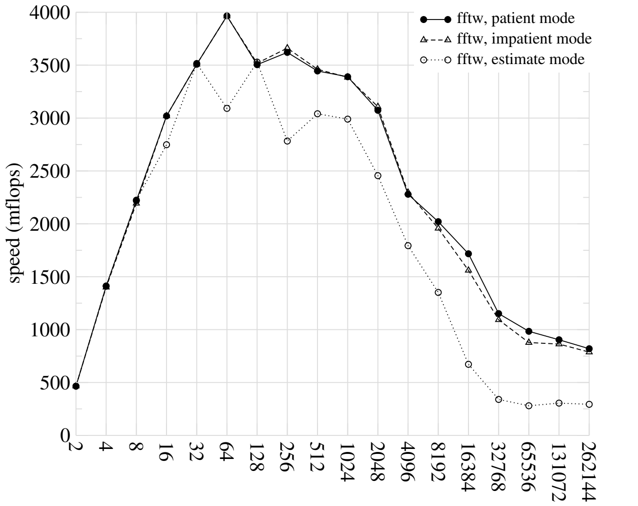


Fig. 9. Effect of planner tradeoffs: comparison of patient, impatient, and estimate modes in FFTW for double-precision 1d complex DFTs, power-of-two sizes, on a 2 GHz PowerPC 970 (G5). Compiler and flags as in Fig. 4.

use the cryptographically strong MD5 algorithm [41]. In the extremely unlikely event of a hash collision, the planner would still return a valid plan, because the solver returned by the table lookup would either construct a valid plan or fail, and in the latter case the planner would continue the search as usual.

V. FFTW3 IN PRACTICE

In this section, we discuss some of our practical experiences with FFTW, from user-interface design, to planning time/optimality tradeoffs, to interesting planner choices that are experimentally observed.

A. User interface

The internal complexity of FFTW is not exposed to the user, who only needs to specify her problem for the planner and then, once a plan is generated, use it to compute any number of transforms of that size. (See Fig. 8.)

Although the user can optionally specify a problem by its full representation as defined in Section IV, this level of generality is often only necessary internally to FFTW. Instead, we provide a set of interfaces that are totally ordered by increasing generality, from a single (vector-rank 0) 1d unit-stride complex transform (as in Fig. 8), to multi-dimensional transforms, to vector-rank 1 transforms, all the way up to the general case. (An alternate proposal has been to modify an FFT/data "descriptor" with a set of subroutines, one per degree of freedom, before planning [42].)

With the more advanced interfaces, which allow the user to specify vector loops and even I/O tensors, it is possible for the user to define nonsensical problems with DFTs of overlapping outputs (Section IV-B). The behavior of FFTW is undefined in such a case; this is rarely a problem, in practice, because only more sophisticated users exploit these interfaces, and such users are naturally capable of describing sensible transforms to perform.

As one additional feature, the user may control tradeoffs in planning speed versus plan optimality by a flag argument (e.g. `FFTW_PATIENT` in Fig. 8). These tradeoffs are discussed below.

B. Planning-time tradeoffs

Depending upon the application, it is not always worthwhile to wait for the planner to produce an optimal plan, even under the dynamic-programming approximation discussed in Section IV-E, so FFTW provides several other possibilities. One option is to load from a file the memoization hash table of Section IV-E), so that the planner need not recompute it. For problems that have not been planned in advance, various time-saving approximations can be made in the planner itself.

In *patient mode* (used for the benchmarks in Section III), the planner tries essentially all combinations of the possible plans, with dynamic programming.

Alternatively, the planner can operate in an *impatient mode* that reduces the space of plans by eliminating some possibilities that appear to inordinately increase planner time relative to their observed benefits. Most significantly, only one way to decompose multi-dimensional $\mathbf{N}$ or $\mathbf{V}$ (Sections IV-C.4 and IV-C.5) is considered, and vector recursion is disabled (Section IV-D.2). Furthermore, the planner makes an approximation: the time to execute a vector loop of ℓ transforms is taken to be ℓ multiplied by the time for one transform. Altogether, impatient mode often requires a factor of 10 less time to produce a plan than the full planner.

Finally, there is an *estimate mode* that performs no measurements whatsoever, but instead minimizes a heuristic cost function: the number of floating-point operations plus the number of "extraneous" loads/stores (such as for copying to buffers). This can reduce the planner time by several orders of magnitude, but with a significant penalty observed in plan efficiency (see below). This penalty reinforces a conclusion of [3]: there is no longer any clear connection between operation counts and FFT speed, thanks to the complexity of modern computers. (Because this connection was stronger in the past, however, past work has often used the count of arithmetic operations as a metric for comparing $O(n \log n)$ FFT algorithms, and great effort has been expended to prove and achieve arithmetic lower bounds [16].)

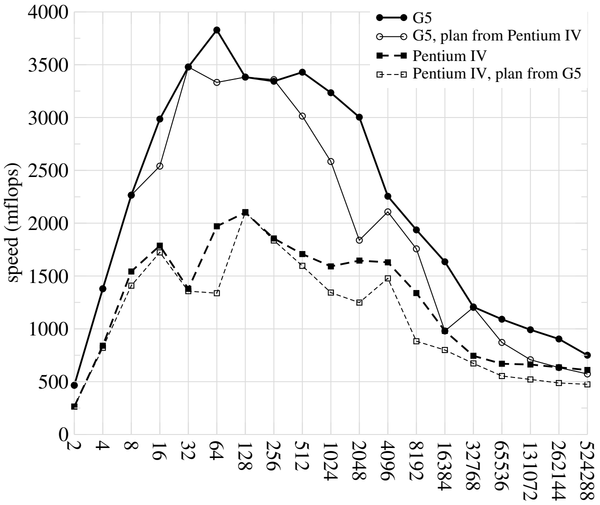


Fig. 10. Effects of tuning FFTW on one machine and running it on another. The graph shows the performance of one-dimensional DFTs on two machines: a 2 GHz PowerPC 970 (G5), and a 2.8 GHz Pentium IV. For each machine, we report both the speed of FFTW tuned to that machine and the speed tuned to the *other* machine.

The relative performance of the 1d complex-data plans created in patient, impatient, and estimate modes are shown in Fig. 9 for the PowerPC G5 from Section III. In this case, estimate mode imposes median and maximum speed penalties of 20% and 72%, respectively, while impatient mode imposes a maximum penalty of 11%. In other cases, however, the penalty from impatient mode can be larger; for example, it has a 47% penalty for a 1024×1024 2d complex-data transform on the same machine, since vector recursion proves important there for the discontiguous (row) dimension of the transform.

It is critical to create a new plan for each architecture—there is a substantial performance penalty if plans from one machine are re-used on another machine. To illustrate this point, Fig. 10 displays the effects of using the optimal plan from one machine on another machine. In particular, it plots the speed of FFTW for one-dimensional complex transforms on the G5 and the Pentium IV. In addition to the optimal plan chosen by the planner on the same machine, we plot the speed on the G5 using the optimal plan from the Pentium IV and vice versa. In both cases, using the wrong machine's plan imposes a speed penalty of 20% or more for at least $1/3$ of the cases benchmarked, up to a 40% or 34% penalty for the G5 or Pentium IV, respectively.

C. Planner choices

It is interesting to consider examples of the sometimes unexpected plans that are actually chosen in practice by the planner.

For example, consider an out-of-place DFT of size $65536 = 2^{16}$. On our Pentium IV, the plan has the overall structure: DIT of radices 32 then 8 then 16, followed by a direct codelet of size 16. However, the first step actually uses buffered DIT, and its size-32 vector loop is pushed down to the direct codelet "leaves" by vector recursion (Section IV-D.2). Moreover, the size-16 direct codelet would normally have discontiguous input and contiguous output; instead, an indirect plan is used to first copy input to output, and then the codelet is executed in-place on contiguous values. The same size on the G5 yields the plan: radix-4 DIT; followed by an indirect plan to copy and work with a contiguous size-16384 in-place sub-plan on the output. The sub-plan is: radix-32 DIT; vector-recursion of the size-32 loop through radix-16 DIT; followed by another indirect plan to perform 16 transposes of size 32×32, and then 512 size-32 direct codelets. The sub-plan's usage of indirect plans fulfills their original purpose of in-place transforms (Section IV-D.3); indirect plans for large out-of-place DFTs were initially a surprise (and often boosted speed by 20% or more).

Another surprise was that, whenever possible, the transposes for in-place DFTs are almost always used at the leaves with a direct codelet, as for the size-16384 sub-plan of the G5 plan above; our preconception was that the transpose would be grouped at an intermediate point with an explicit DIF step (as for the DIF-transpose codelets). As another example, an in-place size-65536 plan on the Pentium IV uses: radix-4 DIT, radix-4 DIF-transpose, two radix-16 DIT steps, and finally an indirect plan that first performs 16×16 transposes and then uses a size-16 direct codelet.

Regarding vector recursion, we had first guessed that a low-stride vector loop would always be pushed all the way to the leaves of the recursion, and an early implementation enforced this constraint. It turns out that this is often not the case, however, and the loop is only pushed one or two levels down, as in the G5 plan above. Indirect plans add another level of complexity, because often the copy (rank-0) sub-plan executes its loops in a different order than the transform sub-plan. This happens, for example, when the (discontiguous) columns of a 1024×1024 array are transformed in-place on the G5, whose resulting plan uses contiguous buffer storing 8 columns at a time, a radix-16 DIT step, an indirect plan that first copies to the buffer than transforms in-place with a size-64 direct codelet, and then copies back. Because the vector loop over the columns is stride-1, it is best to push that loop to the leaves of the copy operations; on the other hand, the direct codelet operates on contiguous buffers so it prefers to have the size-16 vector loop innermost. (A similar effect, with different radices, occurs in the Pentium IV plan for this problem.)

While "explanations" can usually be fabricated in hindsight, we do not really understand the planner's choices because we cannot predict what plans will be produced. Indeed, this is the whole point of implementing a planner.

VI. THE GENFFT CODELET GENERATOR

The base cases of FFTW's recursive plans are its "codelets," and these form a critical component of FFTW's performance. They consist of long blocks of highly optimized, straight-line code, implementing many special cases of the DFT that give the planner a large space of plans in which to optimize. Not only was it impractical to write numerous codelets by hand, but we also needed to rewrite them many times in order to explore different algorithms and optimizations. Thus, we designed a special-purpose "FFT compiler" called `genfft` that produces the codelets automatically from an abstract description. `genfft` is summarized in this section and described in more detail by [2].

As discussed in Section IV, FFTW uses many kinds of codelets: "direct" codelets (Section IV-C.3.a), "twiddle" codelets in the DIT and DIF variants (Section IV-C.3.b), and the more exotic "DIF-transpose codelets" (Section IV-D.3). (Additional kinds of codelets will be presented in Sections VII and VIII.)

In principle, all codelets implement some combination of the Cooley-Tukey algorithm from Eq. (2) and/or some other DFT algorithm expressed by a similarly compact formula. However, a high performance implementation of the DFT must address many more concerns than Eq. (2) alone suggests. For example, Eq. (2) contains multiplications by 1 that are more efficient to omit. Eq. (2) entails a run-time factorization of n, which can be precomputed if n is known in advance. Eq. (2) operates on complex numbers, but breaking the complex-number abstraction into real and imaginary components turns out to expose certain non-obvious optimizations. Additionally, to exploit the long pipelines in current processors, the recursion implicit in Eq. (2) should be unrolled and re-ordered to a significant degree. Many further optimizations are possible if the complex input is known in advance to be purely real (or imaginary). Our design goal for `genfft` was to keep the expression of the DFT algorithm independent of such concerns. This separation allowed us to experiment with various DFT algorithms and implementation strategies independently and without (much) tedious rewriting.

`genfft` is structured as a compiler whose input consists of the kind and size of the desired codelet, and whose output is C code. `genfft` operates in four phases: creation, simplification, scheduling, and unparsing.

In the *creation* phase, `genfft` produces a representation of the codelet in the form of a directed acyclic graph (dag). The dag is produced according to well-known DFT algorithms: Cooley-Tukey (Eq. (2)), prime-factor [27, page 619], split-radix [16], and Rader [28]. Each algorithm is expressed in a straightforward math-like notation, using complex numbers, with no attempt at optimization.

In the *simplification* phase, `genfft` applies local rewriting rules to each node of the dag in order to simplify it. This phase performs algebraic transformations (such as eliminating multiplications by 1), common-subexpression elimination, and a few DFT-specific transformations. These simplifications are sufficiently powerful to derive DFT algorithms specialized for real and/or symmetric data automatically from the complex algorithms. We take advantage of this property to implement real-data DFTs (Section VII), to exploit machine-specific "SIMD" instructions (Section IX), and to generate codelets for the discrete cosine (DCT) and sine (DST) transforms (Section VIII).

In the *scheduling* phase, `genfft` produces a topological sort of the dag (a "schedule"). The goal of this phase is to find a schedule such that a C compiler can subsequently perform a good register allocation. The scheduling algorithm used by `genfft` offers certain theoretical guarantees because it has its foundations in the theory of cache-oblivious algorithms [35] (here, the registers are viewed as a form of cache). As a practical matter, one consequence of this scheduler is that FFTW's machine-independent codelets are no slower than machine-specific codelets generated by SPIRAL [43, Figure 3].

In the stock `genfft` implementation, the schedule is finally unparsed to C. A variation from [44] implements the rest of a compiler backend and outputs assembly code.

VII. Real-data transforms

In this section, we briefly outline how FFTW computes DFTs of real data (a *real DFT*), and we give a new $O(n \log n)$-time algorithm to compute the one-dimensional DFT of real arrays of prime length n.

As is well known, the DFT Y of a real array of length n has the *Hermitian symmetry*

$$Y[n-k] = Y^*[k] \ , \tag{3}$$

where $Y^*[k]$ denotes the complex conjugate of $Y[k]$. (A similar symmetry holds for multi-dimensional arrays as well.) By exploiting this symmetry, one can save roughly a factor of two in storage and, by eliminating redundant operations within the FFT, roughly a factor of two in time as well [45].

The implementation of real-data DFTs in FFTW parallels that of complex DFTs discussed in Section IV. For direct plans, we use optimized codelets generated by `genfft`, which automatically derives specialized real-data algorithms from the corresponding complex algorithm (Section VI). For Cooley-Tukey plans, we use a mixed-radix generalization of [45], which works by eliminating the redundant computations in a standard Cooley-Tukey algorithm applied to real data [22], [46], [47].

When the transform length is a prime number, FFTW uses an adaptation of Rader's algorithm [28] that reduces the storage and time requirements roughly by a factor of two with respect to the complex case. The remainder of this section describes this algorithm, which to our knowledge has not been published before.

The algorithm first reduces the real DFT to the discrete Hartley transform (DHT) by means of the well-known reduction of [48], and then it executes a DHT variant of Rader's algorithm. The DHT was originally proposed by [48] as a faster alternative to the real DFT, but [45] argued that a well-implemented real DFT is always more efficient than an algorithm that reduces the DFT to the DHT. For prime sizes, however, no real-data variant of Rader's algorithm appears to be known, and for this case we propose that a DHT is useful.

To compute DHTs of prime size, recall the definition of DHT:

$$Y[k] = \sum_{j=0}^{n-1} X[j] \operatorname{cas}\left(\frac{2\pi jk}{n}\right) \ , \tag{4}$$

where $\operatorname{cas}(x) = \cos(x) + \sin(x)$. If n is prime, then there exists a generator g of the multiplicative group modulo n: for all $j \in \{1, 2, \ldots, n-1\}$, there exists a unique integer $p \in \{0, 1, \ldots, n-2\}$ such that that $j = g^p \pmod{n}$. Similarly, one can write $k = g^{-q} \pmod{n}$ if $k \neq 0$. For nonzero k, we can thus rewrite Eq. (4) as follows.

$$Y[g^{-q}] = X[0] + \sum_{p=0}^{n-2} X[g^p] \operatorname{cas}\left(\frac{2\pi g^{-(q-p)}}{n}\right) , \quad (5)$$

where the summation is a cyclic convolution of a permutation of the input array with a fixed real sequence. This cyclic convolution can be computed by means of two real DFTs, in which case the algorithm takes $O(n \log n)$ time, or by any other method [49]. (FFTW computes convolutions via DFTs.) The output element $Y[0]$, which is the sum of all input elements, cannot be computed via Eq. (5) and must be calculated separately.

An adaptation of Bluestein's prime-size algorithm to the DHT also exists [50], but the known method does not exhibit asymptotic savings over the complex-data algorithm.

VIII. Trigonometric transforms

Along with the DHT, there exist a number of other useful transforms of real inputs to real outputs: namely, DFTs of real-symmetric (or anti-symmetric) data, otherwise known as the discrete cosine and sine transforms (DCTs and DSTs), types I–VIII [27], [51]–[53]. We collectively refer to these transforms as *trigonometric transforms*. Types I–IV are equivalent to ($\sim$ double-length) DFTs of even size with the different possible half-sample shifts in the input and/or output. Types V–VIII [52] are similar, except that their "logical" DFTs are of *odd* size; these four types seem to see little practical use, so we do not implement them. (In order to make the transforms unitary, additional factors of $\sqrt{2}$ multiplying some terms are required, beyond an overall normalizaton of $1/\sqrt{n}$. Some authors include these factors, breaking the direct equivalence with the DFT.)

Each type of symmetric DFT has two kinds of plans in FFTW: direct plans (using specialized codelets generated by `genfft`), and general-length plans that re-express a rank-1 transform of length n in terms of a real-input DFT plus pre/post-processing. (Here, n denotes the number of non-redundant real inputs.)

In the rest of this section, we show how `genfft` generates the codelets required by trigonometric direct plans (Section VIII-A), and we discuss how FFTW implements trigonometric transforms in the general case (Section VIII-B).

A. Automatic generation of trigonometric-transform codelets

`genfft` does not employ any special trigonometric-transform algorithm. Instead, it takes the position that all these transforms are just DFTs in disguise. For example, a DCT-IV can be reduced to a DFT as follows. Consider the definition of the DCT-IV:

$$Y[k] = 2 \sum_{j=0}^{n-1} X[j] \cos\left(\frac{\pi\left(j+\frac{1}{2}\right)\left(k+\frac{1}{2}\right)}{n}\right)$$

This definition can be rewritten in this way:

$$\begin{aligned} Y[k] = & \sum_{j=0}^{n-1} X[j] e^{2\pi i\,(2j+1)(2k+1)/(8n)} \\ & + \sum_{j=0}^{n-1} X[j] e^{-2\pi i\,(2j+1)(2k+1)/(8n)} \ . \end{aligned}$$

In other words, the outputs of a DCT-IV of length n are just a subset of the outputs of a DFT of length $8n$ whose inputs have been made suitably symmetric and interleaved with zeros. Similar reductions apply to all other kinds of trigonometric transforms.

Consequently, to generate code for a trigonometric transform, `genfft` first reduces it to a DFT and then it generates a dag for the DFT, imposing the necessary symmetries, setting the appropriate inputs to 0, and pruning the dag to the appropriate subset of the outputs. The symbolic simplications performed by `genfft` are powerful enough to eliminate all redundant computations, thus producing a specialized DCT/DST algorithm. This strategy requires no prior knowledge of trigonometric-transform algorithms and is exceptionally easy to implement.

Historically, the generator of FFTW2 (1999) implemented experimental, undocumented support for the DCT/DST I and II in this way. Vuduc and Demmel independently rediscovered that `genfft` could derive trigonometric transforms from the complex DFT while implementing ideas similar to those described in this section [54].

B. General trigonometric transforms

Type II and III trigonometric transforms of length n are computed using a trick from [22], [55] to re-express them in terms of a size-n real-input DFT. Types I and IV are more difficult, because we have observed that convenient algorithms to embed them in an equal-length real-input DFT have poor numerical properties: the type-I algorithm from [22], [31] and the type-IV algorithm from [56] both have L_2 (root mean square) relative errors that seem to grow as $O(\sqrt{n})$. We have not performed a detailed error analysis, but we believe the problem is due to the fact that both of these methods multiply the data by a bare cosine (as opposed to a unit-magnitude twiddle factor), with a resulting loss of relative precision near the cosine zero. Instead, to compute a type-IV trigonometric transform, we use one of two algorithms: for even n, we use the method from [57] to express it as pair of type-III problems of size $n/2$, which are solved as above; for odd n, we use a method from [58] to re-express the type-IV problem as a size-n real-input DFT (with a complicated re-indexing that requires no twiddle factors at all). For the type-I DCT/DST, however, we could not find any accurate algorithm to re-express the transform in terms of an equal-length real-input DFT, and thus we resort to the "slow" method of embedding it in a real-input DFT of length $2n$. All of our methods are observed to achieve the same $O(\sqrt{\log n})$ L_2 error as the Cooley-Tukey FFT [59].

One can also compute symmetric DFTs by directly specializing the Cooley-Tukey algorithm, removing redundant operations as we did for real inputs, to decompose the transform into smaller symmetric transforms [53], [56], [57]. Such a recursive strategy, however, would require eight new sets of codelets to handle the different types of DCT and DST, and we judged the cost in code size to be unacceptable.

IX. How FFTW3 uses SIMD

This section discusses how FFTW exploits special *SIMD* (Single-Instruction Multiple Data) instructions, which perform the same operation in parallel on a data vector. These instructions are implemented by many recent microprocessors, such as the Intel Pentium III (SSE) and IV (SSE2), the AMD K6 and successors (3DNow!), and some PowerPC models (AltiVec). The design of FFTW3 allowed us to efficiently support such instructions simply by plugging in new types of codelets, without disturbing the overall structure.

SIMD instructions are superficially similar to "vector processors", which are designed to perform the same operation in parallel on an all elements of a data array (a "vector"). The performance of "traditional" vector processors was best for long vectors that are stored in contiguous memory locations, and special algorithms were developed to implement the DFT efficiently on this kind of hardware [22], [26]. Unlike in vector processors, however, the SIMD vector length is small and fixed (usually 2 or 4). Because microprocessors depend on caches for performance, one cannot naively use SIMD instructions to simulate a long-vector algorithm: while on vector machines long vectors generally yield better performance, the performance of a microprocessor drops as soon as the data vectors exceed the capacity of the cache. Consequently, SIMD instructions are better seen as a restricted form of instruction-level parallelism than as a degenerate flavor of vector parallelism, and different DFT algorithms are required.

In FFTW, we experimented with two new schemes to implement SIMD DFTs. The first scheme, initially developed by S. Kral, involves a variant of `genfft` that automatically extracts SIMD parallelism from a sequential DFT program [44]. The major problem with this compiler is that it is machine-specific: it outputs assembly code, exploiting the peculiarities of the target instruction set.

The second scheme relies on an abstraction layer consisting of `C` macros in the style of [60], and it is therefore semi-portable (the `C` compiler must support SIMD extensions in order for this scheme to work). To understand this SIMD scheme, consider first a machine with length-2 vectors, such as the Pentium IV using the SSE2 instruction set (which can perform arithmetic on pairs of double-precision floating-point numbers). We view a *complex* DFT as a pair of *real* DFTs:

$$\mathrm{DFT}(A + i \cdot B) = \mathrm{DFT}(A) + i \cdot \mathrm{DFT}(B) \ , \qquad (6)$$

where A and B are two real arrays. Our algorithm computes the two real DFTs in parallel using SIMD instructions, and then it combines the two outputs according to Eq. (6).

This SIMD algorithm has two important properties. First, if the data is stored as an array of complex numbers, as opposed to two separate real and imaginary arrays, the SIMD loads and stores always operate on correctly-aligned contiguous locations, even if the the complex numbers themselves have a non-unit stride. Second, because the algorithm finds two-way parallelism in the real and imaginary parts of a single DFT (as opposed to performing two DFTs in parallel), we can completely parallelize DFTs of any size, not just even sizes or powers of 2.

This SIMD algorithm is implemented in the codelets: FFTW contains SIMD versions of both direct and twiddle codelets (as defined in Section IV-C.3). It may seem strange to implement the complex DFT in terms of the real DFT, which requires much more involved algorithms. Our codelet generator `genfft`, however, derives real codelets automatically from complex algorithms, so this is not a problem for us.

On machines that support vectors of length 4, we view SIMD data as vectors of two complex numbers, and each codelet executes two iterations of its loop in parallel. (A similar strategy of codelets that operate on 2-vectors was argued in [11] to have benefits even without SIMD.) The source of this 2-way parallelism is the codelet loop, which can arise from the Cooley-Tukey decomposition of a single 1d DFT, the decomposition of a multi-dimensional DFT, or a user-specified vector loop. Four-way SIMD instructions are problematic, because the input or the output are not generally stride-1, and arbitrary-stride SIMD memory operations are more expensive than stride-1 operations. Rather than relying on special algorithms that preserve unit stride, however, FFTW relies on the planner to find plans that minimize the number of arbitrary-stride memory accesses.

Although compilers that perform some degree of automatic vectorization are common for SIMD architectures, these typically require simple loop-based code, and we are not aware of any that is effective at vectorizing FFTW, nor indeed of any automatically vectorized code that is competitive on these 2-way and 4-way SIMD architectures.

X. Concluding remarks

For many years, research on FFT algorithms focused on the question of finding the best single algorithm, or the best strategy for implementing an algorithm such as Cooley-Tukey. Unfortunately, because computer hardware is continually changing, the answer to this question has been continually changing as well. Instead, we believe that a more stable answer may be possible by changing the question: instead of asking what is the best algorithm, one should ask what is the smallest collection of simple algorithmic fragments whose composition spans the optimal algorithm on as many computer architectures as possible.

FFTW is a step in that direction, but is not the ultimate answer; several open problems remain. Besides the obvious point that many possible algorithmic choices remain to be explored, we do not believe our existing algorithmc fragments to be as simple or as general as they should. The key to almost every FFT algorithm lies in two elements: strides (re-indexing) and twiddle factors. We believe that our current formalism for problems expresses strides well, but we do not know how to express twiddle factors properly. Because of this limitation, we are currently forced to distinguish between decimation-in-time and decimation-in-frequency Cooley-Tukey, which causes redundant coding. Our ultimate goal (for version 2π) is to eliminate this redundancy so that we can express many possible re-arrangements of the twiddle factors.

ACKNOWLEDGMENTS

We are indebted to F. Franchetti and S. Kral for their efforts in developing experimental SIMD versions of FFTW. Thanks to Greg Allen and the University of Texas for providing access to a PowerPC 970. SGJ is grateful to J. D. Joannopoulos for his unfailing encouragement of this project. Finally, we are indebted to the anonymous reviewers for helpful suggestions that improved the quality of this paper.

REFERENCES

[1] M. Frigo and S. G. Johnson, "The FFTW web page," `http://www.fftw.org/`, 2004.

[2] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, vol. 34, no. 5. Atlanta, Georgia: ACM, May 1999, pp. 169–180.

[3] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, vol. 3, Seattle, WA, May 1998, pp. 1381–1384.

[4] G. Jayasumana, "Searching for the best Cooley-Tukey FFT algorithms," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, vol. 4, 1987, pp. 2408–2411.

[5] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proc. 2nd Int'l Conf. Architectural Support for Programming Languages and Operating System (ASPLOS)*, 1987, pp. 122–127.

[6] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," in *Proc. Int'l Conf. Supercomputing*, Vienna, Austria, July 1997.

[7] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," Computer Science Department, Univ. Tennessee, Knoxville, TN, Tech. Rep. CS-97-366, 1997.

[8] S. K. S. Gupta, C. Huang, P. Sadayappan, and R. W. Johnson, "A framework for generating distributed-memory parallel programs for block recursive algorithms," *J. Parallel and Distributed Computing*, vol. 34, no. 2, pp. 137–153, May 1996.

[9] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. M. Veloso, and R. W. Johnson, "SPIRAL: A generator for platform-adapted libraries of signal processing algorithms," *Journal of High Performance Computing and Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[10] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[11] K. S. Gatlin, "Portable high performance programming via architecture-cognizant divide-and-conquer algorithms," Ph.D. dissertation, University of California, San Diego, 2000.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 1990.

[13] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *J. Machine Learning Research*, vol. 3, pp. 887–919, 2002, special issue on the Eighteenth Int'l Conf. Machine Learning (ICML 2001).

[14] J. W. Cooley and J. W. Tukey, "An algorithm for the machine computation of the complex Fourier series," *Math. Computation*, vol. 19, pp. 297–301, Apr. 1965.

[15] M. T. Heideman, D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform," *IEEE ASSP Magazine*, vol. 1, no. 4, pp. 14–21, 1984.

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

[17] C. van Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia: SIAM, 1992.

[18] D. H. Bailey, "FFTs in external or hierarchical memory," *J. Supercomputing*, vol. 4, no. 1, pp. 23–35, May 1990.

[19] R. C. Singleton, "On computing the fast Fourier transform," *Comm. ACM*, vol. 10, pp. 647–654, 1967.

[20] A. H. Karp, "Bit reversal on uniprocessors," *SIAM Rev.*, vol. 38, no. 1, pp. 1–26, 1996.

[21] T. G. Stockham, "High speed convolution and correlation," *Proc. AFIPS Spring Joint Computer Conference*, vol. 28, pp. 229–233, 1966.

[22] P. N. Swarztrauber, "Vectorizing the FFTs," in *Parallel Computations*, G. Rodrigue, Ed. New York: Academic Press, 1982, pp. 51–83.

[23] H. W. Johnson and C. S. Burrus, "An in-place in-order radix-2 FFT," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, 1984, pp. 28A.2.1–4.

[24] C. Temperton, "Self-sorting in-place fast Fourier transforms," *SIAM J. Scientific and Statistical Computing*, vol. 12, no. 4, pp. 808–823, 1991.

[25] Z. Qian, C. Lu, M. An, and R. Tolimieri, "Self-sorting in-place FFT algorithm with minimum working space," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 42, no. 10, pp. 2835–2836, 1994.

[26] M. Hegland, "A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing," *Numerische Mathematik*, vol. 68, no. 4, pp. 507–547, 1994.

[27] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1999.

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

[29] L. I. Bluestein, "A linear filtering approach to the computation of the discrete Fourier transform," *Northeast Electronics Research and Eng. Meeting Record*, vol. 10, pp. 218–219, 1968.

[30] S. Winograd, "On computing the discrete Fourier transform," *Math. Computation*, vol. 32, no. 1, pp. 175–199, Jan. 1978.

[31] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. New York, NY: Cambridge Univ. Press, 1992.

[32] R. C. Singleton, "An algorithm for computing the mixed radix fast Fourier transform," *IEEE Trans. Audio and Electroacoustics*, vol. AU-17, no. 2, pp. 93–103, June 1969.

[33] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, "On computing the split-radix FFT," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 34, no. 1, pp. 152–156, Feb. 1986.

[34] D. Takahashi, "A blocking algorithm for FFT on cache-based processors," in *Proc. 9th Int'l Conf. High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science, vol. 2110. Amsterdam, The Netherlands: Springer-Verlag, 2001, pp. 551–554.

[35] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th Ann. Symp. Foundations of Computer Science (FOCS '99)*, New York, USA, Oct. 1999.

[36] M. Dow, "Transposing a matrix on a vector computer," *Parallel Computing*, vol. 21, no. 12, pp. 1997–2005, 1995.

[37] E. G. Cate and D. W. Twigg, "Algorithm 513: Analysis of in-situ transposition," *ACM Trans. Math. Software (TOMS)*, vol. 3, no. 1, pp. 104–110, 1977.

[38] W. M. Gentleman and G. Sande, "Fast Fourier transforms—for fun and profit," *Proc. AFIPS Fall Joint Computer Conference*, vol. 29, pp. 563–578, 1966.

[39] K. Nakayama, "An improved fast Fourier transform algorithm using mixed frequency and time decimations," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 36, no. 2, pp. 290–292, 1988.

[40] A. Saidi, "Decimation-in-time-frequency FFT algorithm," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, vol. 3, 1994, pp. 453–456.

[41] R. Rivest, "The MD5 message-digest algorithm," Network Working Group Request for Comments (RFC) 1321, Apr. 1992.

[42] P. T. P. Tang, "A comprehensive DFT API for scientific computing," in *The Architecture of Scientific Software*, ser. IFIP Conference Proceedings, R. F. Boisvert and P. T. P. Tang, Eds., vol. 188. Ottawa, Canada: Kluwer, Oct. 2001, pp. 235–256.

[43] J. Xiong, D. Padua, and J. Johnson, "SPL: A language and compiler for DSP algorithms," in *Proc. ACM SIGPLAN'01 Conf. Programming Language Design and Implementation (PLDI)*, 2001, pp. 298–308.

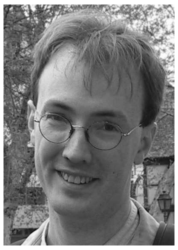
[44] F. Franchetti, S. Kral, J. Lorenz, and C. Ueberhuber, "Efficient utilization of SIMD extensions," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 409–425, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[45] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus, "Real-valued fast Fourier transform algorithms," *IEEE Trans. Acoustics,*

Speech, and Signal Processing, vol. ASSP-35, no. 6, pp. 849–863, June 1987.
[46] C. Temperton, "Fast mixed-radix real Fourier transforms," *Journal of Computational Physics*, vol. 52, pp. 340–350, 1983.
[47] G. D. Bergland, "A fast Fourier transform algorithm for real-valued series," *Comm. ACM*, vol. 11, no. 10, pp. 703–710, 1968.
[48] R. N. Bracewell, *The Hartley Transform*. New York: Oxford Univ. Press, 1986.
[49] H. J. Nussbaumer, *Fast Fourier transform and convolution algorithms*, 2nd ed. Springer-Verlag, 1982.
[50] J.-I. Guo, "An efficient design for one-dimensional discrete Hartley transform using parallel additions," *IEEE Trans. Signal Processing*, vol. 48, no. 10, pp. 2806–2813, Oct. 2000.
[51] Z. Wang, "Fast algorithms for the discrete W transform and for the discrete Fourier transform," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 32, no. 4, pp. 803–816, 1984.
[52] S. A. Martucci, "Symmetric convolution and the discrete sine and cosine transforms," *IEEE Trans. Signal Processing*, vol. 42, pp. 1038–1051, 1994.
[53] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Boston, MA: Academic Press, 1990.
[54] R. Vuduc and J. Demmel, "Code generators for automatic tuning of numerical kernels: experiences with FFTW," in *Proc. Semantics, Application, and Implementation of Code Generators Workshop*, Montreal, Sept. 2000.
[55] J. Makhoul, "A fast cosine transform in one and two dimensions," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 28, no. 1, pp. 27–34, 1980.
[56] S. C. Chan and K. L. Ho, "Direct methods for computing discrete sinusoidal transforms," *IEE Proc. F*, vol. 137, no. 6, pp. 433–442, 1990.
[57] Z. Wang, "On computing the discrete Fourier and cosine transforms," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 33, no. 4, pp. 1341–1344, 1985.
[58] S. C. Chan and K. L. Ho, "Fast algorithms for computing the discrete cosine transform," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 3, pp. 185–190, 1992.
[59] J. C. Schatzman, "Accuracy of the discrete Fourier transform and the fast Fourier transform," *SIAM J. Scientific Computing*, vol. 17, no. 5, pp. 1150–1166, 1996.
[60] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber, "Architecture independent short vector FFTs," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, vol. 2, 2001, pp. 1109–1112.

Matteo Frigo received his Ph. D. in 1999 from the Dept. of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology (MIT). Besides FFTW, his research interests include the theory and implementation of Cilk (a multi-threaded system for parallel programming), cache-oblivious algorithms, and software radios. In addition, he has implemented a gas analyzer that is used for clinical tests on lungs.

Joint recipient, with Steven G. Johnson, of the 1999 J. H. Wilkinson Prize for Numerical Software, in recognition of their work on FFTW.



Steven G. Johnson joined the faculty of Applied Mathematics at MIT in 2004. He received his Ph. D. in 2001 from the Dept. of Physics at MIT, where he also received undergraduate degrees in computer science and mathematics. His recent work, besides FFTW, has focused on the theory of photonic crystals: electromagnetism in nano-structured media. This has ranged from general research in semi-analytical and numerical methods for electromagnetism, to the design of integrated optical devices, to the developement of optical fibers that guide light within an air core to circumvent limits of solid materials. His Ph. D. thesis was published as a book, *Photonic Crystals: The Road from Theory to Practice*, in 2002.