Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW

Position Paper

Richard Vuduc[1] and James W. Demmel[2]

[1] Computer Science Division
University of California at Berkeley, Berkeley, CA 94720, USA
richie@cs.berkeley.edu
[2] Computer Science Division and Dept. of Mathematics
University of California at Berkeley, Berkeley, CA 94720, USA
demmel@cs.berkeley.edu

Abstract. Achieving peak performance in important numerical kernels such as dense matrix multiply or sparse-matrix vector multiplication usually requires extensive, machine-dependent tuning by hand. In response, a number automatic tuning systems have been developed which typically operate by (1) generating multiple implementations of a kernel, and (2) empirically selecting an optimal implementation. One such system is FFTW (Fastest Fourier Transform in the West) for the discrete Fourier transform. In this paper, we review FFTW's inner workings with an emphasis on its code generator, and report on our empirical evaluation of the system on two different hardware and compiler platforms. We then describe a number of our own extensions to the FFTW code generator that compute efficient discrete cosine transforms and show promising speed-ups over a vendor-tuned library. We also comment on current opportunities to develop tuning systems in the spirit of FFTW for other widely-used kernels.

1 Introduction

This paper presents a few findings of our exploratory work in the area of automatic software performance tuning. One specific aim of our study was to conduct a detailed empirical evaluation of the code generation component of FFTW (Fastest Fourier Transform in the West) [13, 14], a software system for automatically generating and performance tuning discrete Fourier transform code. To see some hint of the full generality of the FFTW approach, we also considered an extension to the system to generate efficient discrete cosine transform (DCT) [25] implementations. Finally, we hoped to discover and propose ideas that could be used to tune automatically other important algorithms.

In the remainder of this section, we describe the overall problem that automatic performance tuning systems address, as well as one basic solution. We include a short summary of recent work in the area. In Section 2 we give an

overview of FFTW, and perform an empirical evaluation in Section 3. The current state of our work toward extending FFTW for the discrete cosine transforms is given in Section 4.

1.1 Motivation

The modular design of software applications has been motivated by the need to improve development, testing, maintenance, and portability of code. In addition, such design can enable software developers to isolate performance within one or more key subroutines or libraries. Tuning these subroutines on many platforms facilitates the creation of applications with *portable performance*, provided the routines or libraries have been tuned on the hardware platforms of interest. In this report, we refer to these performance-critical library subroutines as *kernels*. Some examples of kernels and their applications are sparse matrix-vector multiply in the solution of linear systems, Fourier transforms in signal processing, discrete cosine transforms in JPEG image compression, and sorting in database applications.

One practical example of a widely-used kernel standard is the Basic Linear Algebra Subroutines (BLAS) [18, 11, 10, 6] standard. Highly tuned versions of the BLAS are typically available on most modern platforms. However, performance tuning by hand can be tedious, error-prone, and time-consuming. Moreover, each routine must be tuned for a particular hardware configuration, and modern general purpose microprocessors are diverse, constantly evolving, and difficult to compile for. Furthermore, compilers face the difficult task of optimizing for the general case, and cannot usually make use of domain- or kernel-specific knowledge. Therefore, we seek automated tuning methods that can be configured for a specific application.

1.2 A Scheme for Automatic Tuning

Several of the most successful attempts to automate the tuning process follow the following general scheme, first proposed and implemented in the PHiPAC project [4, 5] for matrix multiplication:

1. Generation: Instead of writing kernels, write kernel generators which output kernels in a portable, high-level source language. These kernels can be specialized according to some set of parameters chosen to capture characteristics of the input (e.g., problem size), or machine (e.g., registers, pipeline structure, cache properties). In addition, a generator can apply kernel-specific transformations (e.g., tiling for matrix multiply).
2. Evaluation: With a parameterized kernel generator in hand, explore the parameter space in search of some optimal implementation. The search might consist of generating an implementation, measuring its performance, selecting a new set of parameters, and repeating this process until no faster implementation can be discovered.[1] Each implementation could also be evaluated for specific workloads.

[1] This could be considered a form of profiling or feedback-directed compilation.

This process could be done either at compile-time (i.e., library or kernel installation time), at run-time, or through some hybrid scheme.

In this report, we focus on the generation process. In addition to PHiPAC and FFTW, other recent automatic code generation systems for specific kernels include Blitz++ [29, 30], ATLAS [33], the Matrix Template Library [26], all three of which target dense linear algebra kernels. More recently, the Sparsity system [17] has been developed for highly-optimized sparse matrix-vector multplication and includes automatic code generation as well. PHiPAC, ATLAS, FFTW, and Sparsity address both the generation and the evaluation problems together. The remainder explore only the generation problem.

Of the above systems, however, only FFTW explicitly maintains a high-level symbolic representation of the computation (kernel) itself. One goal of this study was to understand what benefits are available by doing so.

Note that Veldhuizen, et al. [31] have compiled a list of current approaches to library construction and automatic tuning. They also coined the term *active libraries* to describe the role that automatic kernel construction and optimization play in software development.

2 FFTW

The FFTW package is a system for producing highly-optimized discrete Fourier transform (DFT) kernels using the family of FFT algorithms. In the following subsection, we provide a brief overview of how the kernels generated by FFTW fare in practice, and summarize the overall software architecture for readers who may be unfamiliar with it. In section 2.2, we describe in more detail the elements of FFTW (specifically, its code generator) that are most relevant to this study.

Note that the material in this section is largely a summary of the material in the original papers on FFTW [13]. However, all empirical measurements presented in this report were made by the authors. All experiments were performed using FFTW version 2.1.3.

2.1 Overview of FFTW

Figure 1 shows two examples of its success in outperforming commonly used alternative routines on two platforms. In the figures, the "naive" algorithm is the FFT algorithm presented in Press, et al. [24]. FFTPACK [27] is a standard library for computing FFTs and related operations. The vendor library shown on the Sun Ultra-10 platform (right) is the highly tuned commercial numerical library available from Sun Microsystems. Note that the algorithms tested compute the complex DFT in double-precision.

One key to the success of FFTW is the recursive decomposition of FFT algorithms. For instance, one way of expressing this decomposition is via the Cooley-Tukey recursive formulation [8] for a 1-dimensional N-point FFT, when

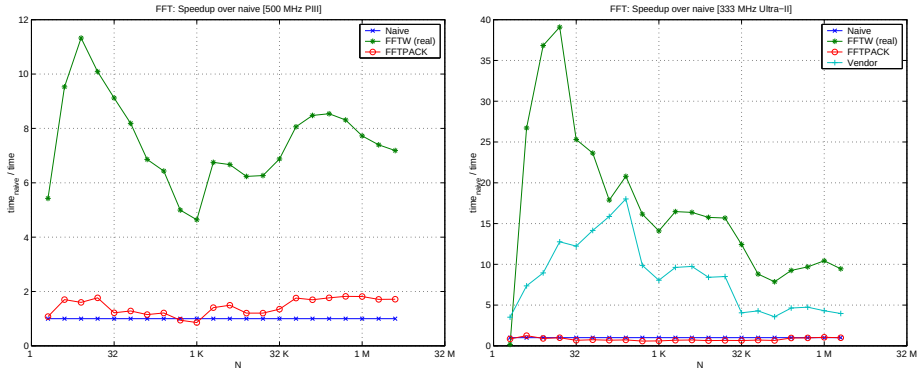

Fig. 1. (*Left*) Performance of the FFTW on a Millennium node. (*Right*) Performance of FFTW on a Sun Ultra-10 workstation (configuration details given in Section 3).

N factors as $N = N_1 N_2$:

$$y[k_1 + k_2 N_1] = \sum_{n_2=0}^{N_2-1} \left[\left(\sum_{n_1=0}^{N_1-1} x[n_1 N_2 + n_2] \omega_{N_1}^{-k_1 n_1} \right) \omega_N^{-k_1 n_2} \right] \omega_{N_2}^{-k_2 n_2} \qquad (1)$$

where $x[n]$ is an element of the input sequence and $y[k]$ is an element of the Fourier transform, with $0 \leq n \leq N-1$, and $\omega_N = e^{2\pi\sqrt{-1}/N}$. The inner sum represents a N_1-point FFT, and the outermost sum represents an N_2-point FFT.

Thus, for any non-prime value of N, we have the freedom to choose the factors N_1 and N_2. Of course, we can also continue to apply the recursion for resulting sub-problems. The FFTW system makes use of this recursive decomposition by (1) generating highly tuned transforms for use at the lowest level (base case) of the recursion, and (2) selecting the decomposition based on empirical performance measurements. Specifically, the system carries out this process through its three components:

Codelet Generator: Creates a symbolic representation of the FFT for a specific size (usually small, say, less than 128 elements), optimizes this high-level representation, and outputs efficient C code. The authors of FFTW refer to these generated implementations of the FFT as *codelets*.

Planner: At run-time, the user gives the planner a length N, and the planner factors N in various ways to get several possible decompositions, or *plans*, of the FFT. The planner then measures the speed of the various codelets to determine which plan is likely to be the fastest. Note that the plan only has to be computed once for a given value of N.[2] To reduce the complexity of exploring the various plans, the planner assumes an optimal substructure property (i.e., a plan for an N_1-length FFT will also be fast even if it is

[2] FFTW also contains a mechanism, referred to as *wisdom*, for storing these plans so they do not have to be recomputed in later runs.

called as a sub-problem of a larger FFT. This is not necessarily true, for instance, due to differences in the state of cache data placement). Also note that FFTW has a mode in which it uses hard-wired heuristics to guess or estimate a plan instead if the overhead of planning cannot be tolerated in a particular application.

Executor: The executor actually computes the FFT, given input data and an optimal plan.

We revisit the run-time overhead in Section 3, and omit any additional discussion of the planner and executor phases otherwise. In the remainder of this section, we describe the codelet generator.

2.2 Codelet generator

The FFTW approach is unique among automatic code generation systems in that it maintains an explicit, high-level symbolic representation of the computation being performed. The FFTW codelet generator, written in Objective Caml, takes a length N as input and performs the following phases in order:

Generate: Generate a high-level, symbolic representation of the FFT computation, referred to as an *expression DAG*. This is like an abstract syntax tree except that it is a directed acyclic graph and not a tree. The DAG structure exposes common subexpressions. Each node is either a load or store of some input variable, or an arithmetic operation whose operands are the predecessors of the node. The data type for a node given in Objective Caml is shown in Figure 2. A visual example of an expression DAG is shown in Figure 3.

Simplify: Visit each node of the DAG in topological order, using rule-based pattern matching to perform a series of "simplifications." These simplifications are essentially local transformations to the DAG that correspond to algebraic simplifications. We describe these in more detail below.

Transpose: Tranpose the DAG. The expressions represented by FFT DAGs corresponds to a mathematical object called a linear network, which is well-known in the signal processing literature [9]. Briefly, a linear network is a graph whose nodes correspond to variables, additions, or subtractions, and whose edges correspond to multiplications. The DAG shown in Figure 3 is shown as a linear network in Figure 4. One property of a linear network is that its transpose, which is the same network with the edges reversed, corresponds to a different, albeit equivalent, dual representation of the original. In the transpose representation, opportunities for simplification are exposed that might not have been otherwise (we will examine this more closely in Section 3).

Iterate: Simplify the transposed DAG, and transpose the DAG again to return to the primal representation. This process of simplifying and transposing can be iterated if desired, although in practice there does not appear to be an appreciable benefit to doing so.

Schedule: Once all transformations have been completed, translate the DAG into a C implementation. This requires scheduling the DAG. The FFTW algorithm for scheduling is asymptotically provably optimal with respect to register or cache usage when N is a power of 2, even though the generator does not know how many registers or how much cache are available [13].

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

Fig. 2. Objective Caml code defining the node data type. "number" represents a floating point constant, and "variable" an input datum, i.e., an element of x or y.

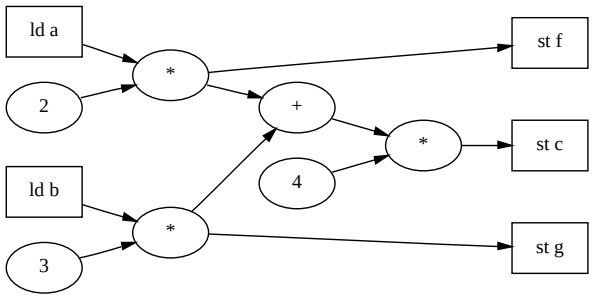

Fig. 3. An expression DAG corresponding to the set of symbolic equations $\{f = 2a, c = 4(2a + 3b), g = 3b\}$.

The simplify phase performs a small number of relatively simple transformations. These include removing additions and multiplications by $0, \pm 1$ and constant folding. It also applies the distributive property, i.e., $ax + ay = a(x + y)$.[3] The simplifier also identifies and creates common subexpressions (in the DAG, these are represented by nodes with multiple outgoing edges). Finally, it performs one DFT-specific optimization, which is to store only positive constants and propagate the minus sign on negative constants as needed. This is because

[3] While this could destroy common subexpressions ax and ay, the FFTW authors claim that this did not occur, although they do not know why.

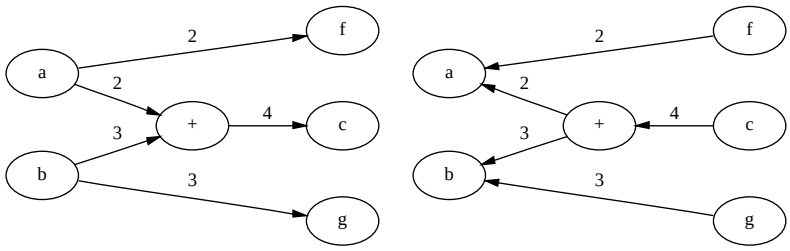

Fig. 4. The DAG of Figure 3 appears here as a linear network (*left*) and its transpose (*right*). Note that the transpose corresponds to the set of equations $\{a = 2f+2(4c), b = 3(4c) + 3g\}$.

many compilers force loads of both positive and negative constants separately, and $+k$ and $-k$ pairs appear very frequently in FFT computations.

The Objective Caml code (roughly) which generates the symbolic form of the FFT as shown in Equation (1) is shown in Figure 5. The correspondence between the Caml code and the equation can be easily seen.

```
let rec cooley_tukey N1 N2 input sign =
  let tmp1 n2 = fftgen N1
    (fun n1 → input (n1 * N2 + n2)) sign in
  let tmp2 k1 n2 =
    exp N (sign * k1 * k2) @* tmp1 n2 k1 in
  let tmp3 k1 = fftgen N2 (tmp2 k1) sign
  in
    fun k → tmp3 (k mod N1) (k / N1)
```

Fig. 5. Roughly, the Objective Caml code that generates a function to compute a symbolic expression for $y[k]$ by Equation (1). Note that @* is an infix operator corresponding to complex multiplication, and that *exp* n k computes $e^{2\pi ik/n}$ where $i = \sqrt{-1}$.

In Figure 5, note that access to the array x being transformed is encapsulated by a function *input*. As we will see later, this helps reduce the complexity of array indexing typical in FFT and related computations. Also note that Equation (1) is only one of many possible expressions for the DFT; the function *fftgen* actually chooses one from among many such algorithms (including *cooley_tukey*) depending on the input size. When either *fftgen* or *cooley_tukey* are evaluated, each returns a Complex.expr which is a pair of nodes (corresponding to the real and imaginary parts of a complex expression). More concretely, *fftgen* has type

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

Note that Figures 2 and 5 were reproduced from [13] with modifications.

We will see how we use this machinery to develop our extensions for the discrete cosine transform in Section 4.

3 Evaluating FFTW

In this section, we show empirically the effect of two of the three major phases in the codelet generator: simplification and DAG transposition.

Evaluation studies were conducted on two platforms. The first was a Sun Ultra-10 workstation with a 333 MHz Ultra-II microprocessor and 2 MB L2 cache running Solaris 2.7. Programs on the Ultra-10 were compiled using Sun's cc v5.0 compiler with the following flags: `-dalign -xtarget=native -xO5 -xarch=v8plusa`. The vendor library used in this study was the Sun Performance Library.

The second platform was a node in the Millennium cluster [1] (a 500 MHz Pentium-III microprocessor and 512-KB L2 cache) running Linux. Programs were compiled using egcs/gcc 2.95 with the following flags: `-O3 -mcpu=pentiumpro -fschedule-insns2`. The vendor library used was the latest version of Greg Henry's tuned numerical libraries for Linux [16] which were developed as part of the ASCI Red Project. Note that due to bugs in the FFT subroutines, we do not report vendor times when input sizes exceed 32K elements.

All timing measurements were made by executing the code of interest for a minimum of 5 times or 5 seconds, whichever was longer, and reporting the average.

3.1 Simplification and DAG Transposition

The authors of FFTW observed reductions in the number of multiplications of between 5% and 25% in certain cases with DAG transposition and subsequent simplification. To see how this can occur, consider the linear network shown in Figure 4 (left). If we simplify the expression $c = 4(2a + 3b)$ to $c = 8a + 12b$, we save a multiplication. However, we cannot apply this simplification haphazardly since it is not beneficial if $2a$ and $3b$ are common subexpressions. In that case, we lose the common subexpressions $2a$ and $3b$, and may increase the overall operation count by one.

However, if we transpose the graph as shown in Figure 4 (right), we see that the simplifier will apply the distributive rule and effectively "discover" the common subexpressions. This results in the network shown in Figure 6. Notice that when we transpose again to return to the primal network, the opportunity to simplify the expression for c by "undistributing" the constant 4 does not exist anymore.

Note that if $2a$ and $3b$ had not been common subexpressions, then the constant 4 would have been propagated when simplifying the transpose, saving an operation. It seems that more complicated pattern matching could identify the difference between the two cases in which the operands either are or are not common subexpressions, obviating the need for transposition. However, such rules might be tedious or complex to develop. Thus, we believe the true benefit of the

transpose is that we may use a relatively small, simple number of patterns in the simplifier. This works because the linear network representation is a suitable representation of an FFT computation.

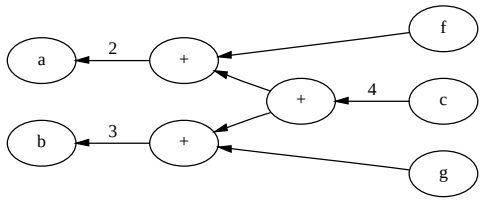

Fig. 6. The distributive rule applied by the simplifier will "discover" the common subexpressions shown in figure 4, resulting in this network.

There is a dramatic way to visualize the global effect of DAG transposition, shown in Figure 7. We modified FFTW's C unparser to output `dot`[4] code. While one could argue that the apparent structural differences are artifacts of the graph layout software. Nevertheless, the bottom network does appear to have discovered additional common sub-expressions, represented by the shift in the grouping structure of the graph to the network inputs.

However, one may rightly wonder what impact these transformations have on the performance realized in practice. For instance, constant folding and common sub-expression elimination are routinely implemented in optimizing compilers. Thus, one might expect these simplifications to duplicate the compiler effort.

We compared the performance of FFTW to itself on the Millennium node in three scenarios: (1) no simplification, (2) algebraic transformations, but no DAG transposition, (3) all optimizations. The results are shown in Figure 8 (*left*), normalized to the full optimization case. It appears that omitting the seemingly simple algebraic optimizations resulted in code that ran as much as two times slower than the implementation generated with full optimizations.

DAG transposition resulted in savings of up to 10% or slightly more in several cases, mostly for prime or odd sizes. Most cases in which omitting DAG transposition appears to be better than using it are generally within the timing noise threshold (we estimate this to be at most about 5% error) and therefore not significant. In those instances in which noise does not explain the finding, we believe that a change in the instruction mix may be the cause. This is because, as the FFTW authors observed, the effect of using DAG transposition is to reduce the number of multiplications. There are generally fewer multiplications than additions, so reducing the number of multiplications only exacerbates the imbalance on architectures in which balancing adds and multiplies leads to better performance.

[4] `dot` is an automatic graph layout application, available from AT&T research: `www.research.att.com/sw/tools/graphviz`

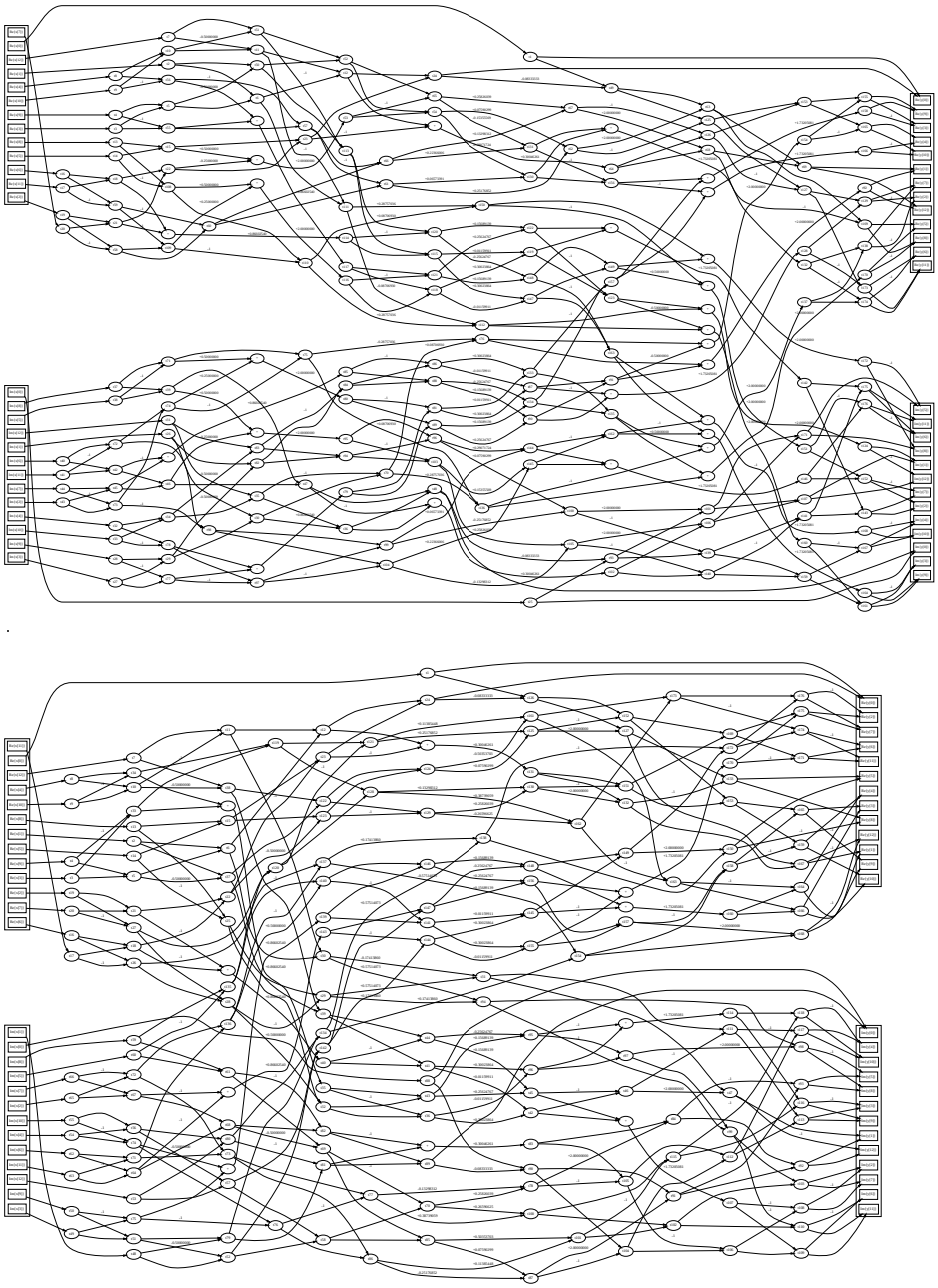

Fig. 7. (*Top*) The linear network representing the complex 13-point FFT without DAG transposition. (*Bottom*) The result when DAG transposition follwed by an additional pass of simplification is applied to the linear network above. To reduce apparent structure which might result as an artifact of the drawing program, real inputs, imaginary inputs, real outputs, and imaginary ouputs have been grouped (shown as boxes). The boxes correspond between the top and bottom figures; however, within the boxes, the order of the nodes may not match between the top and bottom.

On the Sun Ultra-10 platform, we can observe similar trends, as shown in Figure 8 (*right*) with respect to the importance of the algebraic simplifications. However, one startling difference is that for a handful of relatively small sizes, omitting DAG transposition would have led to up to a 35% speed improvement.

Note that in both experiments, we allowed the FFTW planner to do a full search for an optimal decomposition.

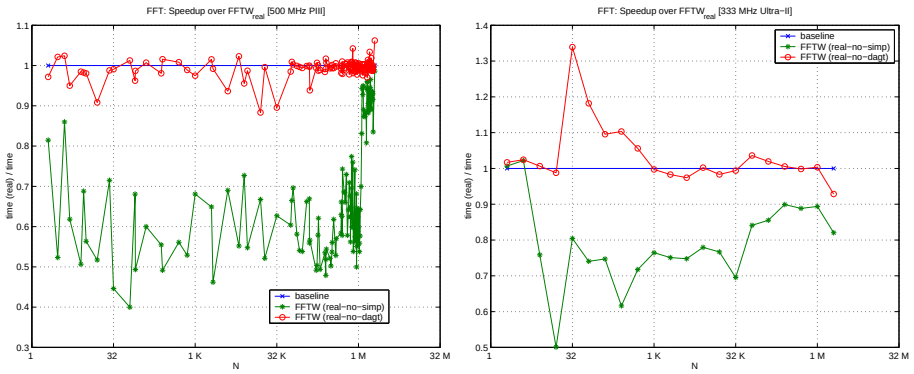

Fig. 8. (*left*) Comparison between (1) no simplification, (2) algebraic simplifications but no DAG transposition, and (3) full optimizations. Powers of 2, 3, 5, and 7 are included, as well as uniformly randomly distributed sizes between 32,000 elements and 1 million. Measurements shown for the Millennium node. (*right*) The same experiment repeated on the Sun Ultra-10 platform, (only powers of 2 shown).

3.2 Run-time Overhead

Most FFT implementations require users to call a pre-initialization routine in which the so-called constant *twiddle factors* (the ω factors in equation (1)) are pre-computed. In practice, users perform many FFTs of the same length on different data, so it is reasonable to perform initialization in advance whose cost can be amortized over many uses. It is during this initialization phase that FFTW executes its planner, and in this section we quantify the run-time overhead incurred.

The results are shown in Figure 9 for the Millennium and the Sun Ultra-10 platforms. In both cases for each implementation, we show how many computations of the naive algorithm are equivalent to the initialization cost. Recall that FFTW has two planning modes: a "real" mode in which the algorithm tries to accurately determine a plan, and an "estimate" mode in which heuristics are applied to select a good plan. Estimation mode costs essentially the same in initialization time as that of conventional algorithms.

The slowdowns incurred from using the estimated plans are shown for the two platforms in Figure 10 for the Millennium and Sun Ultra-10 platforms, re-

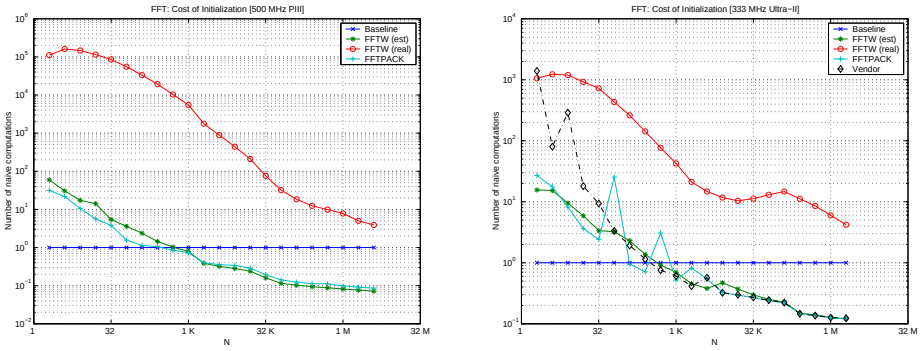

Fig. 9. Planner overhead on the Millennium node (*left*) and on the Sun Ultra-10 (*right*).

spectively. Note that on the Ultra-10 (for which the FFTW authors note that they implicitly tuned the heuristic algorithm), the difference between using the real vs. estimated plan is unclear. In fact, sometimes the estimated plan resulted in better performance. This is not surprising in the sense that the "real" mode does not actually find the optimal plan since it uses the optimal sub-plan heuristic to prune the search space. However, on the Millennium node, real mode does tend to find better plans, leading to a speedup of 15% to 30% for large powers of two.

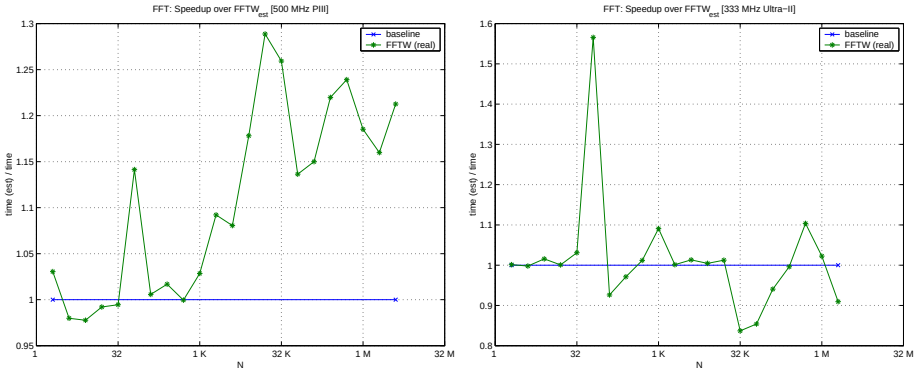

Fig. 10. Resulting performance difference between real vs. estimated run-time planning modes on the Millennium node (*left*) and the Sun Ultra-10 (*right*).

4 Fast Discrete Cosine Transforms

The results of the evaluation indicated that the generator, which includes special built-in transformations specific to the FFT, was effective in generating efficient

code. Before considering generalizations to completely different kernels, it seemed natural to consider extending the system to compute a very similar transform: the discrete cosine transform (DCT).

The version of FFTW available at the time of this writing included code to generate DCT codelets, but did not have a complete DCT planner and executor implementation. However, since there are many ways to generate DCT codelets, only one of which was available in FFTW, we considered a variety of alternatives. We have also begun implementation of a DCT planner and executor. In this section, we describe the relationship between the DCT and FFT and the current status of our effort to extend FFTW.

4.1 Overview

The DCT has numerous applications in speech coding, image compression, and solving partial differential equations. For instance, both the current JPEG image file standard and the upcoming HDTV standard rely on the DCT as a fundamental kernel. There are a variety of formal definitions of the DCT and in this report we use the formulation known as DCT-II, defined as follows. The 1-D DCT $y[k]$ of a real-valued input signal $x[n]$ is given by

$$y[k] = 2 \sum_{n=0}^{N-1} x[n] \cos\left(\pi \frac{k(2n+1)}{2N}\right) \tag{2}$$

where $0 \leq k, n < N$, and the finite signal N is assumed to be even and symmetric about $n = N - \frac{1}{2}$, with period $2N$. An example of a signal and its extension is shown in Figure 11. Note that direct computation of equation (2) for all k has an arithmetic complexity of $O(N^2)$.

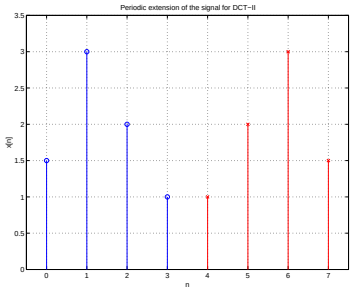

Fig. 11. The original signal (blue) of length 4 is extended (red) to be even and symmetric about $n = 3.5$ with period 8.

There are many ways to compute the DCT with an arithmetic complexity of $O(N \log N)$, as with the FFT. We considered two classes of methods: (1) reduction of the DCT to an equivalent FFT computation and (2) purely real recursive

formulations. The former allowed us to make the most use of pre-existing FFTW code generation infrastructure, while the latter should in principle lead to implementations with fewer arithmetic operations. We describe the algorithms we tried and their results below.

4.2 DCT-II via the FFT

We considered four algorithms which all work by reducing the DCT to an equivalent FFT computation:

1. Reduction to a $2N$-point real-to-complex FFT: This is the "textbook" method [23]. Let $V[k]$ be the $2N$-point FFT of one period of the extended signal. Then the DCT-II $y[k]$ of $x[n]$ is given by

$$y[k] = 2\mathfrak{Re}\left\{V[k]\omega_{4N}^{-k}\right\} \tag{3}$$

2. Reduction to one real-to-complex N-point FFT: This requires permuting the input so that the even elements $x[2n]$ appear first, followed by the odd elements $x[2n+1]$ in reverse order. If $V[k]$ is the real-to-complex FFT of this N-length permuted signal, then the DCT-II $y[k]$ is given by equation (3).
3. Reduction to one complex $N/2$-point FFT: We create a new complex signal $v[n]$ of length $\frac{N}{2}$ whose real and imaginary parts are the even and odd elements of the original signal respectively.
4. Reduction to a $4N$-point real-to-complex FFT: Consider padding the period $2N$ extended signal with zeros between every other element to obtain a length $4N$ signal. Then the first N elements of this transformed signal will be exactly the DCT-II of the original signal. This is the algorithm used in FFTW to generate DCT codelets.[5]

We might consider implementing the DCT by not changing the generator at all, and instead by just calling the pre-built FFT routines as in any of the above formulations. However, as shown in Figure 12, this is not nearly as efficient as the vendor and FFTPACK implementations on the Sun Ultra-10 for input sizes under 128K elements. Thus, we considered generating DCT-II codelets directly.

As with the FFT, each of the algorithms described above is simple to express in the code generator. Specifically, since all four algorithms involve a rearrangement of the input, we really only need to ensure that the appropriate *input* function (as in Figure 5) is supplied.

Observe that only the $4N$-point transform does not require explicit multiplication by the extra twiddle factor ω_{4N}^{-k} as in equation (3). Furthermore, the redundancy in the $4N$ element extension exposes more common sub-expressions than the more compact reductions. Irrelevant parts of the computation are effectively pruned by both simplification and the fact that the generator only stores

[5] We actually implemented this as well, although we didn't realize our formulation was the same as the pre-existing one at the time.

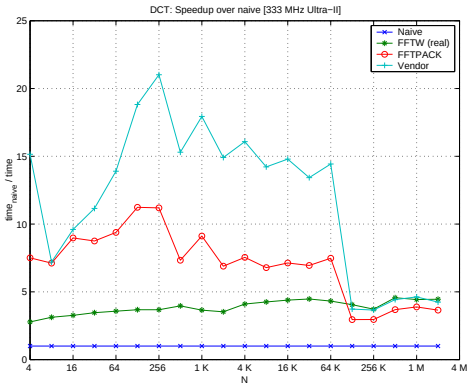

Fig. 12. Performance of the DCT-II implemented by calling the pre-built FFTW routine (i.e., without using specialized DCT-II codelets) on the Sun Ultra-10 platform.

expressions that go to one of the desired outputs. This led to a DCT implementation that required about 25% fewer multiplications for non-power of two sizes than the more compact representations. (Power of 2 sizes produced identical implementations.)

4.3 Purely Real DCTs

Instead of reducing the DCT to an equivalent FFT computation, we can also express the DCT recursively in terms of smaller DCTs. Below, we list the three variants of these so-called purely real recursive formulations that we tried. In all cases, we assume N is a power of 2.

1. Decimation in frequency (DIF) algorithm (Yip and Rao [34]): Given an N-length sequence $x[n]$, define two new sequences

$$g[n] = x[n] + x[N-n-1]$$
$$h[n] = \frac{1}{2}(x[n] - x[N-n-1])$$

   where $0 \leq n < \frac{N}{2}$. Let $G[m], H[m]$ be the DCT-II transforms of g, h respectively. Then the even and odd elements of the DCT-II $y[k]$ of $x[n]$ are

$$y[2m] = G[m]$$
$$y[2m+1] = H[m] + V_o[m+1] \tag{4}$$

   where $0 \leq m < \frac{N}{2}$ and $H[\frac{N}{2}] = 0$.
2. Zhijin's algorithm [35]: Define g, h and G, H as above. Let $b[n] = 2h[n]\cos\left(\pi\frac{2n+1}{2N}\right)$, and let $B[k]$ be its DCT-II. Then the even and odd components of the DCT-II $y[k]$ of $x[n]$ are

$$y[2m] = G[m]$$

$$y[1] = B[0]/2$$
$$y[2m+1] = B[m] - y[2m-1] \tag{5}$$

3. Restructured recursive DCT (Lee and Huang [19]): Define a recursive function $\hat{C}(x)$, which takes a sequence of x of length N and returns another length-N sequence, as follows. When the input sequence x has length 2, $\hat{C}(x)$ is the DCT-II of x. Otherwise, define two new sequences

$$x_1[k] = x[k] + x[k + \frac{N}{2}]$$
$$x_2[k] = x[k] - x[k + \frac{N}{2}]$$

where $0 \leq k < \frac{N}{2}$. Then $X[k]$ be a new sequence such that

$$X[2k] = \{\hat{C}(x_1)\}[k]$$
$$X[2k+1] = \{\hat{C}(x_2)\}[k] \cdot 2\cos\left(\pi \frac{4k+1}{2N}\right) - X[2k-1]$$
$$X[-1] = 0$$

Then $\hat{C}(x) = X[0..N-1]$. To obtain the DCT-II of x, define $\hat{x}$ to be the following permutation of x:

$$\hat{x}[2k] = x[k]$$
$$\hat{x}[2k+1] = x[N-1-k]$$

Then the DCT-II of x is $\hat{C}(\hat{x})$.

As with the FFT, these can be implemented in a straightforward way in FFTW's codelet generator, thus benefitting from the simplifier/scheduler structure of FFTW's code generator. We give an example of how the first algorithm can be expressed in Appendix A.

The three variations appear similar on the surface but lead to implementations with different numbers of additions and multiplications once they have been processed by the codelet generator's optimizer, as shown in Figure 13. The execution times on both platforms are comparable for $N < 16$; for $N \geq 16$, FFT-algorithm 4 either matched or outperformed the real formulations. In comparing the three purely real codelets, we see that operation count alone does not explain the execution times. A slight difference in just one addition and multiplication on the Millennium node for $N = 16$, for instance, caused the DCT algorithm 3 to be about 15% slower than DCT algorithm 1.

Figure 14 shows the raw codelet performance on the Sun Ultra-10 and the Millennium node. We show only the codelets based on FFT-algorithm 4 in both cases. Note that performance is normalized to the FFTW-based DCT-II implementation, and not a naive implementation. The drop-off at 128 and 256 is expected, since by then the fully unrolled codelet begins to exert significant register pressure on the code, leading to frequent spills.

		Real DCT-IIs		
N	4N-FFT	DIF	Zhijin	Lee
2	2+2	2+2	2+2	2+2
	(4)	(4)	(4)	(4)
	.14	.14	.14	.14
	.10	.10	.10	.10
4	8+6	9+5	8+6	8+6
	(14)	(14)	(14)	(14)
	.17	.18	.17	.17
	.14	.14	.14	.14
8	26+16	29+14	27+15	28+15
	(42)	(43)	(42)	(43)
	.23	.25	.24	.25
	.26	.25	.24	.25
16	72+42	81+37	77+37	80+38
	(114)	(118)	(114)	(118)
	.44	.43	.44	.45
	.52	.50	.51	.62
32	186+104	209+92	201+89	208+93
	(290)	(301)	(290)	(301)
	.86	.95	1.01	1.11
	1.22	1.32	1.33	1.27
64	456+250	513+219	497+209	512+220
	(706)	(732)	(706)	(732)
	1.87	2.68	3.25	3.10
	2.84	3.15	3.12	3.24

Fig. 13. Comparison between the 4N-point FFT-based algorithm and the three purely real recrusive DCT-II algorithms. The four rows correspond to (1) the number of real additions and multiplications (shown as $a + m$), (2) total arithmetic operations, and execution times (in μs) on (3) the Sun Ultra-10 platform and (4) the Millennium Node. The approximate timing uncertainties on the Sun Ultra-10 platform is $\pm.01$ μs, and on the Millennium node it is $\pm.02$ μs. The full set of FFTW code generation optimizations were applied in all implementations.

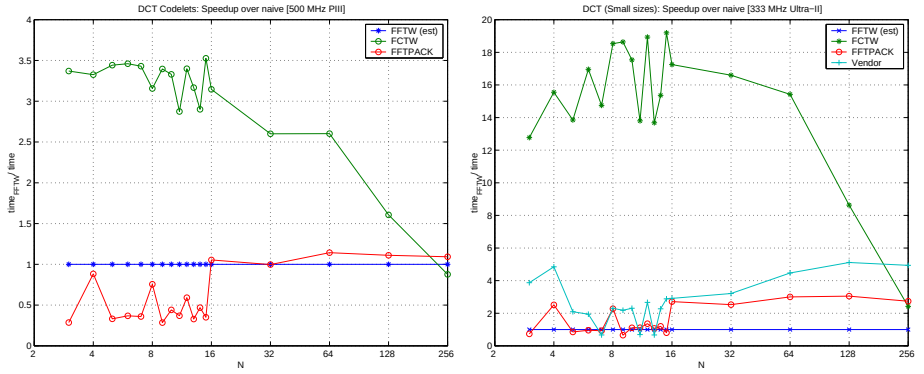

Fig. 14. DCT-II specialized codelet performance on a Millennium node (*left*) and the Sun Ultra-10 (*right*). All codelets of size less than 16 are shown, with the remaining points at powers of 2.

While this appears promising, the DCT-II codelets are probably not optimal in terms of the number of operations, since they derive from complex-valued FFTs instead of a purely real-valued formulation. In the case of the 8-point DCT-II, the code included in the freely available JPEG reference implementation [2, 21] only requires 29 additions and 11 multiplications. This compares to 26 additions and 16 multiplications for the 8-point DCT-II codelet. The actual running times of the two implementations are identical to within the timing noise resolution, however, reflecting the differences in structure, pipeline usage, and scheduling of the two codes. Nevertheless, this suggests that a more efficient formulation for the DCT-II codelets is possible. We have not yet had a chance to investigate this further, but plan to do so in the near term.

The important point is that within the code generator's framework, we were able to experiment relatively quickly and simply with a large variety of implementations. The multitude of other recursive DCT-II formulations [3, 7, 20, 22, 25, 32], some of which handle non-power of 2 cases, are all characterized by complex indexing schemes (unlike the FFT case), which we expect will be relatively easy to handle in the FFTW framework.

However, this work is not yet complete because we have not yet finished implementation of the planner and executor for the DCT. We also expect to complete this in the near term. Finally, we should point out that we did not examine or introduce any new simplification rules to the simplifier that would benefit the purely real formulations. Nevertheless, we believe that the visualization component we introduced will facilitate the identification of such rules.

5 Summary and Future Work

We have presented data that gives some idea of the relative merits of the various techniques employed in the FFTW codelet generator. In particular, simple

FFT-specific transformations led to significantly improved final code. Moreover, for odd and prime sizes, transposition followed by additional simplification was beneficial as well. However, we saw that in a number of cases on the Ultra-10, for instance, in which DAG transposition and additional simplification reduced the efficiency of the final code. Therefore, there are a number of additional scheduling and compiler interaction issues that remain unresolved.

Nevertheless, the choice of implementation of the FFTW codelet generator was simple to extend to compute a similar transform, the DCT, and led to extremely fast codelets.

Furthermore, explicitly maintaining the symbolic representation and having multiple ways of "unparsing" it (in our case, we added the `dot` code output lead to an interesting visualization. This could be extremely useful in future code generation systems.

There are a number of future directions in which to take this work, which can be classified as FFTW-specific improvements, future kernel generators, and future compiler research. We make a few suggestions below.

5.1 FFTW-specific Improvements

In addition to completing the DCT part of FFTW by implementing a planner and generator, the kernel generator could be extended to handle 2-D transforms as well. This is because specialized 2-D kernels (specifically in the DCT case) are widely used and could benefit from additional symmetries in the 2-D case.

Also, we did not discuss the codelet generator's scheduler in any detail. Briefly, the scheduler is "cache-oblivious" meaning there are no specific references or parameters that might directly relate to the number of available registers or size of the cache. An asymptotically optimal scheduling does exist, although it is not explicitly parameterized by the cache size (or equivalently the number of available registers). It would be useful to see if a cache-aware scheduler can lead to improved performance.

Finally, we did not discuss numerical stability issues. This problem is generic to any floating point kernel generator since we are now "deriving" new algorithms that may have completely different floating point behavior.

5.2 Other Kernels

This project was originally motivated by a desire to produce a specialized kernel generator for the fast wavelet transform (FWT). My initial research revealed that the FWT could benefit from a similar approach if small wavelet kernels were needed in applications. This is because wavelet transforms have similar symmetries that lead to many common sub-expressions. However, the FWT has *linear* arithmetic and I/O complexity with respect to the input size, and its recursive decomposition calls only one smaller transform at each level of the recursion. This implies that a highly tuned "kernel" for small sized transforms might not be as beneficial since each small transform is called only once. In

contrast, the FFT can potentially call a small-sized transform many times at each level of the recursion (depending on how the input length factors), making the need for highly-tuned kernels for small-sized transforms much more important.

One important kernel that could benefit from FFTW codelet generator techniques is sparse matrix-vector multiplication, when the structure of the sparse matrix is known. In finite element computations, for instance, the sparse matrices often have a known block structure, where the blocks are small and symmetric. In these cases, it may be possible to generate dense matrix-vector multiply "codelets" for these blocks.[6] For example, such codelets could be specialized verisons of the "register-blocked routines" generated in the Sparsity system [17].

An important set of kernels that, like the FFT, have recursive structure are the recursive formulations of dense matrix multiply, Cholesky and LU factorization [28, 15], and QR factorization [12]. Researchers have identified the importance of small kernel routines in these cases. However, we do not currently know what the right high-level representation and scheduling decisions should be.

5.3 Implications for Compilers

We do not believe that the full implications of this work in kernel generation for compiler writers will be known until more kernel generators have been written. It is clear, however, that for the time being, the flow of technology transfer will be from compiler work to kernel generation. In this sense, we feel that one important area for development in kernel generators is to find representations convenient for expressing loops (instead of just recursion as in FFTW) and associated loop transformations like tiling, unrolling, and software pipelining; many kernels may be more naturally expressed in this way.

References

1. Millennium: An hierarchical campus-wide cluster of clusters (home page). `www.millennium.berkeley.edu`.
2. Independent JPEG Group Reference Implementation, May 2000. `www.ijg.org`.
3. G. Bi. DCT algorithms for composite sequence lengths. *IEEE Transactions on Signal Processing*, 46(3):554–562, March 1998.
4. J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing, Vienna, Austria*, July 1997.
5. J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C. Chin. The PHiPAC WWW home page. `www.icsi.berkeley.edu/~bilmes/phipac`.
6. S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. W. von Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) standard: Blas technical forum. `www.netlib.org/cgi-bin/checkout/blast/blast.pl`.

[6] Indeed, the DFT is simply a matrix-vector multiply operation where the matrix is fixed and highly symmetric.

7. L.-P. Chau and W.-C. Siu. Recursive algorithm for the discrete cosine transform with general lengths. *Electronic Letters*, 30(3):197–198, February 1994.
8. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, April 1965.
9. R. E. Crochiere and A. V. Oppenheim. Analysis of digital linear networks. In *Proceedings of the IEEE*, volume 63, pages 581–595, April 1975.
10. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.
11. J. Dongarra, J. D. Croz, I. Duff, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.
12. E. Elmroth and F. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development*, 44(1), January 2000. `www.cs.umu.se/ elmroth/papers/eg99.ps`.
13. M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.
14. M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 1998.
15. F. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development*, 41(6), November 1997. `www.research.ibm.com/journal/rd/416/gustavson.html`.
16. G. Henry. Linux libraries for 32-bit Intel Architectures, March 2000. `www.cs.utk.edu/ ghenry/distrib`.
17. E.-J. Im and K. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
18. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.
19. P. Z. Lee. Restructured recursive DCT and DST algorithms. *IEEE Transactions on Signal Processing*, 42(7), July 1994.
20. P. Z. Lee and F.-Y. Huang. An efficient prime-factor algorithm for the discrete cosine transform and its hardware implementation. *IEEE Transactions on Signal Processing*, 42(8):1996–2005, August 1994.
21. C. Loeffler, A. Ligtenberg, and G. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 988–991, May 1989.
22. D. P. K. Lun. On efficient software realization of the prime factor discrete cosine transform. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 465–468, April 1994.
23. A. Oppenheim and R. Schafer. *Discrete-time Signal Processing.* Prentice-Hall, 1999.
24. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992. `www.nr.com`.
25. K. R. Rao and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications.* Academic Press, Inc., 1992.
26. J. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, December 1998.

27. P. Swarztrauber. FFTPACK User's Guide, April 1985. `www.netlib.org/fftpack`.
28. S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4), 1997. www.math.tau.il/ sivan/pubs/029774.ps.gz.
29. T. Veldhuizen. The Blitz++ home page. `oonumerics.org/blitz`.
30. T. Veldhuizen. Using C++ template metaprograms. *C++ Report Magazine*, 7(4):36–43, May 1995.
31. T. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.
32. Z. Wang. Recursive algorithms for the forward and inverse discrete cosine transform with arbitrary lengths. *IEEE Signal Processing Letters*, 1(7):101–102, July 1994.
33. R. C. Whaley and J. Dongarra. The ATLAS WWW home page. `www.netlib.org/atlas/`.
34. P. Yip and K. R. Rao. The decimation-in-frequency algorithms for a family of discrete sine and cosine transforms. *Circuits, Systems, and Signal Processing*, pages 4–19, 1988.
35. Z. Zhijin and Q. Huisheng. Recursive algorithms for the discrete cosine transform. In *Proceedings of the IEEE International Conference on Signal Processing*, volume 1, pages 115–118, October 1996.

A DCT-II DIF Algorithm in Objective Caml

The purely real recursive algorithm can be expressed in just a dozen lines of Caml code. The resulting expression then benefits from all of the same transformations and infrastructure available to FFTW.

```
let rec dct2 n input =
    let g = let g_input = fun i ->
          (input i) @+ (input (n-i-1))
       in fun m -> dct2 (n/2) g_input m
    and h = let h_input = fun i ->
          let twiddle = (sec (4*n) (2*i+1)) @* complex_half
          in ((input i) @- (input (n-i-1))) @* twiddle
       in fun m -> dct2 (n/2) h_input m
    in fun m -> (* dct2 n input m *)
       let m2 = m / 2
       in if n == 1 then complex_two @* input 0 (* 1 element input *)
          else if (m mod 2) == 0 then (* m even *)
             g m2
          else (* m odd *)
             if m == (n-1) then (* last element *)
                h m2
             else
                (h m2) @+ (h (m2+1))
```

Note that the *input* function (not shown) provided to *dct2* is defined to only load a real element.