123

# The Fastest Fourier Transform in the South

*Anthony M. Blake, *Member, IEEE*, Ian H. Witten, *Member, IEEE*, and Michael J. Cree, *Senior Member, IEEE*

***Abstract*—This paper describes FFTS, a discrete Fourier transform (DFT) library that achieves state-of-the-art performance using a new cache-oblivious algorithm implemented with run-time specialization. During initialization the transform parameters, such as sign and direction, are fixed, allowing sequencing and data access information to be precomputed and specialized machine code to be generated. The resulting code may be executed any number of times on different input data. In contrast to FFTW, FFTS does not use large base cases at the leaves of recursion, nor a substantial library of codelets, and does not require time-consuming machine-specific calibration. The code presented in this paper has been benchmarked on recent Intel x86 and ARM machines, and is, in almost all cases, faster than self-tuning libraries such as FFTW, and even vendor-tuned libraries such as Intel IPP and Apple vDSP.**

***Index Terms*—FFT, Fourier transform, run-time specialization, dynamic code generation**

## I. Introduction

STATE-OF-THE-ART libraries for computing the discrete Fourier transform (DFT) divide into two categories: vendor-tuned libraries such as Intel Integrated Performance Primitives (IPP) and Apple vDSP, and self-tuning libraries such as FFTW ("The Fastest Fourier Transform in the West"), SPIRAL and UHFFT. The latter are intended as a response to the increasingly difficult problem of reasoning about the interaction between hardware and software; however, self-tuning libraries do not automatically produce good performance on arbitrary machines, and some programming effort is required before they can take advantage of machine-specific features and instructions (such as Altivec on PPC, SSE and AVX on x86, and NEON on ARM).

As well as automatically adapting to the hardware, self-tuning libraries are, in effect, applying the technique of *program specialization*. Instead of providing a single highly parameterized function for computing a range of DFTs, they use an initialization stage to set up a function that operates on one argument, the input data, from a general function of several arguments, such as the size of the input and the direction of the DFT. The idea of program specialization is far from new, and was formulated and proven as Kleene's s-m-n theorem more than 50 years ago [1].

State-of-the-art libraries that are referred to as being 'vendor-tuned' also employ program specialization to some extent. In an initialization stage, trigonometric coefficients specific to the size and direction (forward or backwards) of the requested transform are precomputed, and a specialized function can be set up based on the capabilities of the machine and the parameters of the transform.

In Blake's PhD thesis [34], a depth-first recursive implementation of the conjugate-pair algorithm is accelerated by iteratively computing blocks of code found at the leaves of recursion prior to computing the rest of the transform recursively. This work gives a succinct description of that algorithm and describes a more efficient implementation using run-time specialization. The resulting DFT library, called FFTS ("The Fastest Fourier Transform in the South") has been benchmarked on recent Intel x86 and ARM machines, and the measurements show that FFTS is, in almost all cases, faster than state-of-the-art vendor-tuned and self-tuning libraries. Additionally, FFTS manages to do this while avoiding the initialization delay and code-size overheads that plague self-tuning libraries.

## II. Related Work

FFTW [2]–[6] and UHFFT [7]–[12] are implementations of the discrete Fourier transform (DFT) that maximize performance by automatically adapting to the hardware at runtime. Given the parameters of a problem, such as size and direction, these libraries employ a *planner* to search the space of all possible factorizations of several highly parameterized FFT algorithms, and find a plan that has the smallest execution time.

Each plan is composed of blocks of straight-line code, called *codelets*, which are optimized at a low level. FFTW, for example, has a library of over 150 pregenerated codelets. FFTW and UHFFT compose plans using a wide range of parameterized FFT algorithms, including the Cooley-Tukey algorithm [13] and its derivatives: the split-radix [14], [15], conjugate-pair [5], [16] and mixed-radix algorithms. Radar's [17] and Bluestein's [18]–[20] algorithms are used for sizes that are prime, and the prime-factor algorithm [19], [21] for sizes that are factored by co-primes.

Because each plan is divided into subproblems, and because many of the subproblems considered during the search are essentially the same, FFTW and UHFFT apply dynamic programming [22] and evaluate the runtime of each subproblem only once. The first time a given subproblem is evaluated, its runtime is stored, or "memoized", and when the same subproblem is encountered subsequently, the previously measured runtime is retrieved from memory. Dynamic programming does not guarantee that the fastest plan will be found, but it is a good tradeoff between initialization time, where the search for the fastest plan occurs, and execution time [2].

The primary difference between FFTW and UHFFT is that UHFFT initializes two databases of execution times during

*A.M. Blake is with the Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton 3240, NEW ZEALAND e-mail: ablake@waikato.ac.nz phone: +64 7 838 4466 ext 8342 fax: +64 7 838 4155

I.H. Witten is with the Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton 3240, NEW ZEALAND e-mail: ihw@cs.waikato.ac.nz phone: +64 7 838 4246 fax: +64 7 858 5095

M.J. Cree is with the School of Engineering, University of Waikato, Private Bag 3105, Hamilton 3240, NEW ZEALAND e-mail: cree@waikato.ac.nz phone: +64 7 838 4301

installation. A codelet database is initialized with the execution times of codelets, and a transform database is initialized with execution times of some popular sizes of FFT (power-of-two and prime-factor algorithm sizes) [10].

In [2], Frigo and Johnson consider those plans that are chosen after searching the space of all possibilities, and conclude that one cannot predict the fastest plan. However, in earlier work they show that some algorithms should in fact have an advantage [24]. In particular, depth-first recursive algorithms have theoretical advantages arising from cache utilization, and these "cache-oblivious" algorithms have been shown to be asymptotically optimal [4], [6], [23], [24].

Kelefouras et al. [25] propose that the performance of an algorithm depends on its utilization of the memory hierarchy, and that it is possible to predict parameter values that will produce the fastest plan based on the characteristics of the underlying machine. However, many of the techniques in their work are specific to scalar microprocessors, and thus the results are not directly applicable to modern machines, which now invariably implement some form of SIMD functionality.

SPIRAL [26]–[31] performs automatic optimization for the Fourier transform and other signal processing functions. However, it differs from FFTW in that it performs the optimization at compile time, and thus the generated code is machine-dependent, even if it is compiled on a different machine. Another point of difference is that SPIRAL uses a wider range of search strategies that include ones based on machine learning [32].

In 1977 Morris [33] demonstrated Fortran programs which generate specialized Fortran FFT subroutines. More recently it was shown that static specialization of a cache-oblivious depth-first recursive algorithm can produce results that are, in many cases, faster than vendor-tuned and automatically optimized libraries on a range of modern machines [34]. Metaprogramming was used to statically elaborate a conjugate-pair algorithm with the size and sign of the transform, producing a specialized C program for those particular parameters. The disadvantage of this approach is that either the size and direction of the required transforms must be known at compile time, or a wide range of transforms has to be generated, resulting in a large binary. This paper addresses this limitation by generating specialized machine code at runtime instead.

## III. Conjugate-pair Algorithm

FFTS uses the conjugate-pair algorithm, or as it sometimes known, the "$-1$ exponent" algorithm. This is a variant of the standard split-radix algorithm that was initially proposed to reduce the minimum count of floating point operations required to compute the FFT [16], but was later shown to have an operation count identical to that of the ordinary split-radix algorithm [35]–[37]. The algorithm was relegated to obscurity for nearly 20 years, but re-emerged recently when van Buskirk used it as the basis for an algorithm that does actually reduce the operation count [38].

The conjugate-pair algorithm is derived from the DFT,

1: **function** $y_{k=0,\ldots,N-1}$ =FFT$(N, x_n)$
2: **if** $N = 1$ **then**
3: **return** $x_0$
4: **else if** $N = 2$ **then**
5: $y_0 \leftarrow x_0 + x_1$
6: $y_1 \leftarrow x_0 - x_1$
7: **return** $y_k$
8: **else**
9: $U_{k_2=0,\ldots,N/2-1} \leftarrow$FFT$(N/2, x_{2n_2})$
10: $Z_{k_4=0,\ldots,N/4-1} \leftarrow$FFT$(N/4, x_{4n_4+1})$
11: $Z'_{k_4=0,\ldots,N/4-1} \leftarrow$FFT$(N/4, x_{4n_4-1})$
12: **for** $k = 0$ to $N/4 - 1$ **do**
13: $y_k \leftarrow U_k + (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$
14: $y_{k+N/2} \leftarrow U_k - (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$
15: $y_{k+N/4} \leftarrow U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$
16: $y_{k+3N/4} \leftarrow U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$
17: **end for**
18: **return** $y_k$
19: **end if**
20: **end function**

Fig. 1. Depth-first recursive algorithm computing conjugate-pair split-radix FFT of length N (divisible by 4).

which is formally defined as [39]:

$$y_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n \tag{1}$$

where $k = 0, \ldots, N-1$ and $\omega_N$ is the primitive root-of-unity $\exp(-2\pi i/N)$. A decimation-in-time (DIT) decomposition of the split-radix algorithm divides the DFT in Equation 1 into three smaller DFTs over the terms $x_{2n_2}$, $x_{4n_4+1}$ and $x_{4n_4+3}$. However, the conjugate-pair algorithm rotates the indices of last sub-transform by $-4$, to obtain:

$$\begin{aligned} y_k \quad = \quad & \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2 k}\, x_{2n_2} + \omega_N^k \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k}\, x_{4n_4+1} \\ & + \omega_N^{-k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k}\, x_{4n_4-1} \end{aligned} \tag{2}$$

where $\omega_N^k$ and $\omega_N^{-k}$ are the conjugate-pair of coefficients, and the negative indices wrap around, i.e., $x_{-1} = x_{N-1}$ [5]. The advantage of this arrangement is that only one coefficient needs to be computed or loaded, instead of the two required by the ordinary split-radix algorithm.

Equation 2 becomes a *fast* Fourier transform when the three DFTs are themselves evaluated recursively with Equation 2, and thus the FFT easily maps to a depth-first recursive algorithm, as shown in Figure 1. Such an implementation makes optimal use of the memory hierarchy while being oblivious of its parameters, because when the sub-transforms of size $N/2$ and $N/4$ are combined into a larger transform of size $N$, they will still be in the closest level of the memory hierarchy in which they fit [24].

Although the FFT naturally maps quite well to a recursive implementation, most traditional implementations found in

```
1: procedure FFT-NoLeaves(N, y_{k=0,...,N-1})
2:   if N > 2 then
3:     FFT-NOLEAVES(N/2, y_{N_2})
4:     FFT-NOLEAVES(N/4, y_{N_4+N/2})
5:     FFT-NOLEAVES(N/4, y_{N_4+3N/4})
6:     U_{k_2=0,...,N/2-1} ← y_{N_2}
7:     Z_{k_4=0,...,N/4-1} ← y_{N_4+N/2}
8:     Z'_{k_4=0,...,N/4-1} ← y_{N_4+3N/4}
9:     for k = 0 to N/4 − 1 do
10:      y_k ← U_k + (ω_N^k Z_k + ω_N^{-k} Z'_k)
11:      y_{k+N/2} ← U_k − (ω_N^k Z_k + ω_N^{-k} Z'_k)
12:      y_{k+N/4} ← U_{k+N/4} − i(ω_N^k Z_k − ω_N^{-k} Z'_k)
13:      y_{k+3N/4} ← U_{k+N/4} + i(ω_N^k Z_k − ω_N^{-k} Z'_k)
14:    end for
15:  end if
16: end procedure
17: procedure FFT-SOUTH(x_{n=0,...,N-1}, y_{k=0,...,N-1})
18:  j ← 0
19:  for k = 0 to N/N_l/3 do
20:    y_{δ_j} ← x_j + x_{j+N/2}
21:    y_{δ_j+1} ← x_j − x_{j+N/2}
22:    j ← j + 1
23:  end for
24:  for k = 0 to N/N_l/3 + (log_2 N and 1 xor 1) − 1 do
25:    y_{δ_j} ← x_j
26:    y_{δ_j+1} ← x_{j+N/2}
27:    j ← j + 1
28:  end for
29:  for k = 0 to N/N_l/3 − 1 do
30:    y_{δ_j} ← x_{j+N/2} + x_j
31:    y_{δ_j+1} ← x_{j+N/2} − x_j
32:    j ← j + 1
33:  end for
34:  FFT-NOLEAVES(N, y_k)
35: end procedure
```

Fig. 2. FFT-SOUTH first computes the base cases iteratively, by order of access to the input array, in the three loops at lines 19, 24 and 29, before computing the rest of the transform with a depth-first post-order traversal. Computing the base cases in this way results in the desirable memory access pattern shown in Figure 4b.

```
1: function δ_{k=0,...,N/N_l-1} =ELAB(N_l, N, O_i, O_o, S, E)
2:   if (E and N == N_l) or (not E and N ≤ N_l) then
3:     δ_0 ← ⟨O_i × 2, O_o⟩
4:     return δ_k
5:   else if N ≥ 2 then
6:     T_0 ←ELAB(N_l, N/2, O_i, O_o, S + 1, E)
7:     T_1 ←ELAB(N_l, N/4, O_i + (1 ≪ S), O_o + N/2, S + 2, 0)
8:     if N/4 ≥ N_l then
9:       T_2 ←ELAB(N_l, N/4, O_i − (1 ≪ S), O_o + 3N/4, S + 2, 0)
10:    else
11:      T_2 ← ⟨⟩
12:    end if
13:    δ_k ← T_0 ‖ T_1 ‖ T_2
14:    return δ_k
15:  end if
16: end function
17: function δ_{k=0,...,N/N_l-1} =INIT-OFFSETS(N_l, N)
18:  δ_k ←ELAB(N_l, N, 0, 0, 1, 1)
19:  for i = 0 → |δ_k| − 1 do
20:    if (δ_i)_0 < 0 then
21:      (δ_i)_0 ← (δ_i)_0 + N
22:    end if
23:  end for
24:  sort(δ_k)
25:  for i = 0 → |δ_k| − 1 do
26:    δ_i ← (δ_i)_1
27:  end for
28:  return δ_k
29: end function
```

Fig. 3. Initialization functions for the precomputed offsets $\delta_k$ used by FFT-SOUTH in Figure 2.

textbooks traverse the tree iteratively [5]. A radix-2 algorithm, for example, will compute $N/2$ size-2 sub-transforms, and then $N/4$ size-4 sub-transforms, and so on, until there have been approximately $\log_2 N$ passes over the data, at which point the transform is complete. An iterative implementation of the split-radix algorithm is not quite so simple, but several efficient schemes have been described [40], [41].

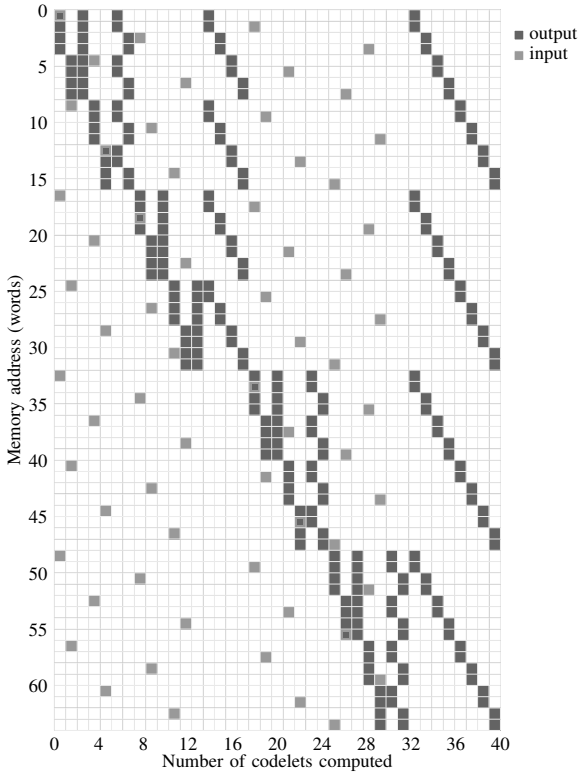
Generally, iterative implementations do not use the memory hierarchy efficiently. However, as mentioned earlier, a recently proposed approach by Kelefouras et al. [25] addresses this problem by partitioning the transform according to the number of levels of the data cache hierarchy, and then choosing sizes for the sub-transforms according to the size of each level. Sets of sub-transforms are computed iteratively within each level.

Most iterative implementations perform a separate pass which reorders either the input or the output data, depending on whether a decimation-in-time (DIT) or decimation-in-frequency (DIF) decomposition is used. Although the reordering pass only requires $O(N)$ time, it can often account for a non-negligible fraction of the overall runtime [2]. In most cases the reordering is a bit-reversal permutation, but the conjugate-pair algorithm has a more complex permutation because of the rotation in the $x_{4n_4-1}$ terms. Recursive implementations, on the other hand, have another distinct advantage in that the permutation can implicitly be performed at the leaves of the computation for an out-of-place transform (ibid.).

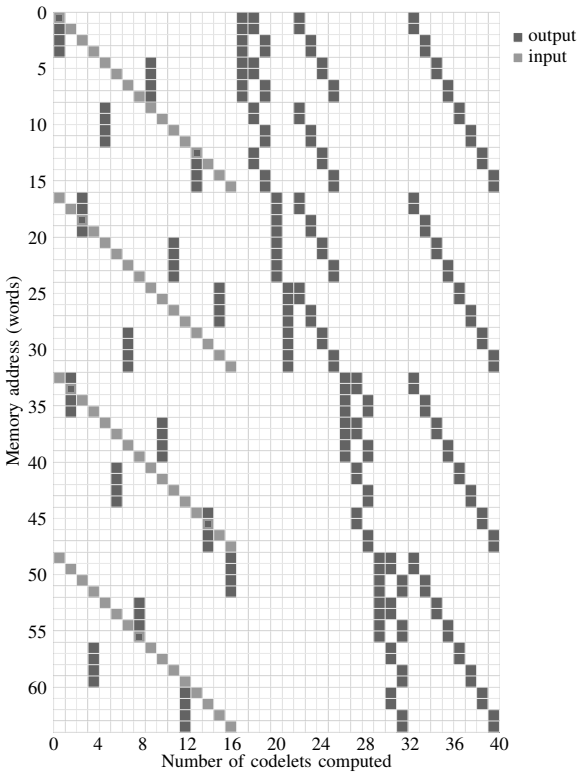
## IV. New FFT Algorithm

The new FFT algorithm, shown in Figure 2, differs from a standard depth-first recursive implementation of the conjugate-pair algorithm in that the base cases, which correspond to lines 2–7 of Figure 1, are first computed iteratively using three loops, in lines 18–33 of Figure 2. Following computation of the base cases, the rest of the transform is computed using a recursive procedure that has no base cases, in lines 1–16 of Figure 2.

A standard implementation of the conjugate-pair algorithm shown in Figure 1 handles two different sizes of base case at lines 2 and 4. In contrast, the new algorithm exploits the fact that smaller base cases are always decomposed in pairs, as in




(a) Standard depth-first recursive implementation, using the algorithm in Figure 1.




(b) Modified implementation, using the algorithm in Figure 2. In the first 16 columns, the base cases are computed iteratively in an order that optimizes spatial locality, and the remainder of the transform is computed recursively.

Fig. 4. Memory access patterns of a size 64 decimation-in-time conjugate-pair FFT operating out-of-place. The 64 rows each correspond to a memory address in either the input or the output array. The arrays have been aliased to conserve space, and access to either array is distinguished by shade. Each column represents the memory accesses of a codelet, starting with the first codelet in the leftmost column, and as time advances, more codelets are computed, moving right along the horizontal axis.

lines 10 and 11 of Figure 1, and so they can be recombined into a codelet that is the same size as the other base case.

The codelet in the first loop of base cases at line 19 of Figure 2 implements a standard $N = 2$ base case, while the second loop at line 24 implements two $N = 1$ base cases in parallel. The third loop implements $N = 2$ base cases, but with a permutation due to the conjugate-pair index rotation [34].

Figure 4a shows the memory access pattern of a standard size 64 depth-first recursive implementation of the conjugate-pair FFT, while Figure 4b depicts the memory access pattern of the new algorithm computing the same transform.

The memory access patterns have 64 rows, each corresponding to an address in either the input or the output array. Each column represents the memory accesses of one codelet, starting with the first codelet in the leftmost column. As time goes on, more codelets are computed, moving right along the horizontal axis. In order to to conserve space in the figures, the input and output data arrays have been aliased and access to either array is distinguished by shade, and both of the memory access patterns depict implementations with size 4 base cases and vectorized main codelets.

As can be seen from the figures, the new algorithm improves spatial locality and avoids the decimated access to the input data that plagues a standard implementation.

Each base case codelet writes a sequential block of data to the output array, and the offset for the output of each codelet is precomputed and stored in $\delta_k$. Recursive code for initializing $\delta_k$ is given in Figure 3, where $N$ is the size of the transform and $N_l$ is the size of the base cases. $O_i$ denotes input array offset, $O_o$ output array offset, $S$ stride, and $E$ even.

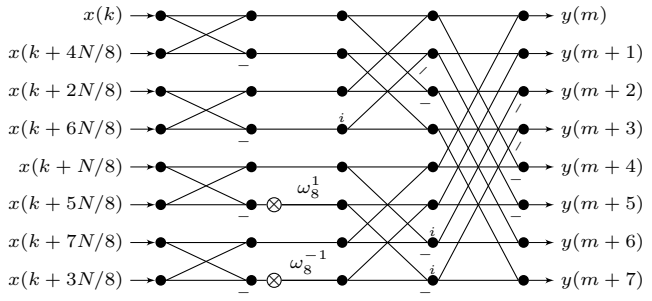
In line 18, a sequence of input and output offset pairs are elaborated in the standard depth-first post-order traversal. The conjugate-pair algorithm rotates some indices such that $x_{-1} = x_{N-1}$, and these indices are adjusted in lines 19–23 before the sequence is sorted by input offset in line 24. Finally, because the input offsets are now ordered from $0, \ldots, N/N_l - 1$, they can be implicitly determined from the position in the sequence, and thus the input offsets are discarded in lines 25–27 before returning the sequence of output offsets.

In line 8 of Figure 3, the recursion for the second $N/4$ subtransform is prevented if the subtransform is a base case, because the second base case will be computed in parallel with the first base case at line 7.

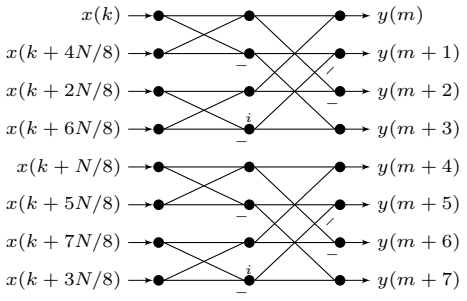
## V. Implementation Details

FFTS uses pregenerated straight-line blocks of code for computing transforms where $N \leq 16$, and as with most other implementations, trigonometric coefficients are precomputed.

Because implementations of SSE, AVX and NEON have 16 vector registers, a size-8 subtransform is the largest that can

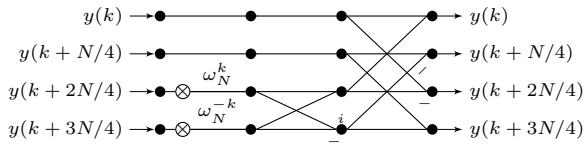



(a) Size-8 base case

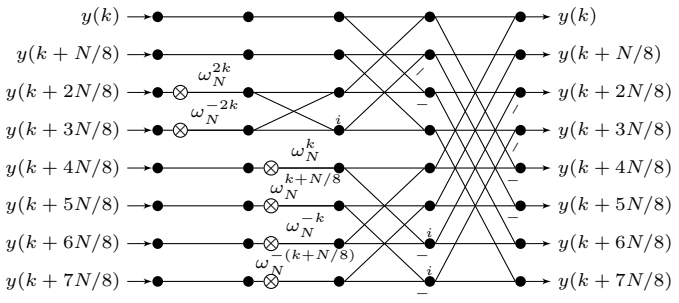



(b) Pair of size-4 base cases in parallel

Fig. 5. Signal-flow graphs of base case codelets.




(a) Standard size-4 codelet




(b) Size-8 codelet composed of three size-4 codelets

Fig. 6. Signal-flow graphs of non-terminal codelets.

be easily computed without spilling registers onto the stack. Thus FFTS uses size-8 base cases, as shown above. The signal-flow graph for the size-8 base case is shown in Figure 5a, and the signal-flow graph for two size-4 base cases computed in parallel is shown in Figure 5b. For the same reasons, the size of the main codelets is increased to size 8, where possible, by subsuming three size-4 codelets into a size-8 codelet, as shown in Figure 6b. The residual codelets that cannot be subsumed into a size-8 codelet are computed using the ordinary size-4 codelet shown in Figure 6a.

The loop of main codelets at line 9 of Figure 2 is trivial to vectorize, but vectorization of the base case loops poses a subtle challenge: two of the loops will have an odd number of iterations. FFTS deals with this problem by automatically generating special codelets to compute the final iteration(s) of one loop in parallel with the first iteration(s) of the next loop (exactly how many iterations from each loop depends on the vector length in the underlying SIMD implementation). The automatic generation of these special codelets is described in [34].

The signal-flow graphs in the above figures are mapped to assembly code fairly directly. For example, the size-4 codelet in Figure 6a is implemented for ARM NEON with the assembly code in Figure 7.

There are, however, the following differences between the codelets in the SSE and NEON implementations.

- On NEON, vector scatter/gather instructions are utilized to load complex data stored in interleaved format and store the intermediate results in split format, before finally converting back to interleaved format in the final pass. On SSE, however, exploratory experiments suggested that it was more efficient to use interleaved format for the entire transform, probably because of the separate shuffle pipe in most SSE implementations.
- The SSE codelets were implemented in C with intrinsics, and were further optimized by hand after compilation. The NEON codelets were entirely hand coded in assembly because there are no intrinsics for instructions such as VSWP. The largest codelet is a size-8 subtransform, and thus small enough that hand optimization is practical.
- The x86 ISA allows the displacement of a memory operation to be encoded as an immediate in the instruction, and so the SSE implementation modifies the displacement at runtime, reducing the number of pointers from eight to one for a size-8 codelet.
- In the NEON implementation, the sign of the trigonometric coefficients is implicitly absorbed into the computation, and thus the data *and* the instructions for an inverse transform are different. However, rather than assemble and link two variants of each codelet, an inverse codelet is obtained from a forwards codelet at runtime using the method described in Section VI. In the SSE implementation, the sign is stored as data and the code for a forward and an inverse transform is the same. Again, this difference is probably attributable to the the separate shuffle pipe in most SSE implementations.

## VI. Run-time specialization

To describe the generation of specialized functions, we first define some notation. If `f` is a function, then $[\![\texttt{f}]\!]$ denotes its

```
1  vld1.32 {q8,q9}, [r0, :128]
2  add r4, r0, r1, lsl #1
3  vld1.32 {q10,q11}, [r4, :128]
4  add r5, r0, r1, lsl #2
5  vld1.32 {q12,q13}, [r5, :128]
6  add r6, r4, r1, lsl #2
7  vld1.32 {q14,q15}, [r6, :128]
8  vld1.32 {q2,q3}, [r2, :128]
9
10 vmul.f32  q0, q13, q3
11 vmul.f32  q5, q12, q2
12 vmul.f32  q1, q14, q2
13 vmul.f32  q4, q14, q3
14 vmul.f32  q14, q12, q3
15 vmul.f32  q13, q13, q2
16 vmul.f32  q12, q15, q3
17 vmul.f32  q2, q15, q2
18 vsub.f32  q0, q5, q0
19 vadd.f32  q13, q13, q14
20 vadd.f32  q12, q12, q1
21 vsub.f32  q1, q2, q4
22 vadd.f32  q15, q0, q12
23 vsub.f32  q12, q0, q12
24 vadd.f32  q14, q13, q1
25 vsub.f32  q13, q13, q1
26 vadd.f32  q0, q8, q15
27 vadd.f32  q1, q9, q14
28 vadd.f32  q2, q10, q13  @ Change to sub for IFFT
29 vsub.f32  q4, q8, q15
30 vsub.f32  q3, q11, q12  @ Change to add for IFFT
31 vst1.32 {q0,q1}, [r0, :128]
32 vsub.f32  q5, q9, q14
33 vsub.f32  q6, q10, q13  @ Change to add for IFFT
34 vadd.f32  q7, q11, q12  @ Change to sub for IFFT
35 vst1.32 {q2,q3}, [r4, :128]
36 vst1.32 {q4,q5}, [r5, :128]
37 vst1.32 {q6,q7}, [r6, :128]
38 bx lr
```

Fig. 7. ARM NEON code for the size-4 codelet in Figure 6a. In terms of lines of code, the other size-8 codelets are about three times larger, but still small enough that hand optimization is practical.

meaning, usually an input/output function. Thus for $n \geq 0$,

$$\texttt{output} = [\![\texttt{f}]\!]\,[\texttt{in}_0, \texttt{in}_1, ..., \texttt{in}_{n-1}]$$

results from running `f` on input values $\texttt{in}_0, \texttt{in}_1, ..., \texttt{in}_{n-1}$, and `output` is undefined if `f` enters an infinite loop [42].

A function that computes an FFT, given parameters $N$, $sign$ and $x_n$, is described by

$$y_{k=0,...,N-1} = [\![\texttt{fft}]\!]\,[N, sign, x_{n=0,...,N-1}]$$

Suppose now that the parameters $N$ and $sign$ are known at initialization time, and the transform will be computed many times on different data. Specialization is applied by performing the calculations that depend on $N$ and $sign$ during initialization, and generating code to compute the remainder of the calculations, which depend on $x_n$. In this case, computation is performed in two stages, described by

$$\begin{aligned} \texttt{fft}_{N,sign} &= [\![\texttt{genfft}]\!]\,[N, sign] \\ y_{k=0,...,N-1} &= [\![\texttt{fft}_{N,sign}]\!]\,[x_{n=0,...,N-1}] \end{aligned}$$

where $\texttt{fft}_{N,sign}$ is a specialized function with fixed parameters $N$ and $sign$. Combining these two yields an equational definition of `genfft`:

$$[\![\texttt{fft}]\!]\,[N, sign, x_n] = [\![\underbrace{[\![\texttt{genfft}]\!]\,[N, sign]}_{\text{specialized program}}]\!]\,[x_n]$$

where if one side of the equation is defined, the other is also defined and has the same value [42].

```
1  push {r4, r5, r6, r7, r8, r9, r10, r11, lr}
2  vstmdb sp!, {d8-d15}
3  add r3, r1, #0
4  add r7, r1, #128
5  add r5, r1, #256
6  add r10, r7, #256
7  add r4, r5, #256
8  add r8, r10, #256
9  add r6, r4, #256
10 add r9, r8, #256
11 ldr r12, [r0]
12 add r1, r0, #0
13 add r0, r2, #0
14 ldr r2, [r1, #128]
15 mov r11, #3
16
17 @ << inlined loop of leaf butterfly codelets >>
18
19 @ << inlined loop of leaf butterfly pair codelets >>
20
21 @ << inlined loop of leaf butterfly codelets >>
22
23 ldr r2, [r1, #32]
24 mov r1, #32
25 add r2, r2, #32
26 bl  neon_x4
27 add r0, r0, #256
28 sub r1, r1, #16
29 sub r2, r2, #32
30 bl  neon_x8
31 add r0, r0, #128
32 bl  neon_x8
33 add r0, r0, #128
34 add r1, r1, #16
35 add r2, r2, #32
36 bl  neon_x4
37 add r0, r0, #256
38 bl  neon_x4
39
40 @ << inlined final loop of neon_x8 >>
41
42 vldmia sp!, {d8-d15}
43 pop {r4, r5, r6, r7, r8, r9, r10, r11, pc}
```

Fig. 8. Generated ARM code for $N = 128$ transform.

The `genfft` function operates in three stages:

1) The parameters $N$ and $sign$ are used to precompute the trigonometric coefficients;
2) If $N \geq 32$ the output offsets are precomputed using the functions in Figure 2;
3) If $N \geq 32$ specialized machine code is generated, otherwise a pointer to a hard-coded transform is returned.

The code in Figure 8 is generated when initializing a transform for $N = 128$ on ARM, and all generated code is of this form, even on x86. Before generating the main function in Figure 8, the codelets for neon_x4 and neon_x8 are copied into place and modified. As previously mentioned, computation of an inverse transform on ARM requires that some instructions be adjusted.

The stack is only used for saving and restoring registers in the prologue and epilogue, corresponding to lines 1, 2, 42 and 43 in Figure 8. Following the prologue, pointers and parameters for the first three loops are setup in lines 3–15, and the three loops are inlined in lines 17–21. In the SSE implementation, only one pointer is used for the base case codelets, and offsets are encoded into the immediate field of the instructions at runtime. Next, the depth-first recursive structure of the non-terminal parts of the transform are elaborated, and for each subtransform a call is emitted, resulting in lines 23–38 of Figure 8. The final non-terminal function is inlined.

## VII. Results

FFTS has been evaluated against other state-of-the-art libraries on recent Intel x86 and ARM machines. The benchmark methods are described in Section VII-A, and speed, initialization time and accuracy results are presented in the following three subsections. Data relating to code size is presented in Section VII-E.

The scope of this work has been limited to single-precision, single-threaded, 1D transforms on power-of-two sizes of inputs, and the benchmarks on Intel x86 machines have been computed with SSE, even when AVX is available (later versions of FFTS will support AVX).

### A. Benchmark Methods

The benchmarks were performed with BenchFFT [44], a collection of libraries and benchmarking software assembled by Frigo and Johnson, the authors of FFTW.

The benchmarks measure speed, initialization time and accuracy. Initialization and run time of an FFT are measured separately. The former is measured once, while the latter is measured more accurately by executing several runs and choosing the minimum time, where each run consists of a large number of FFTs.

The minimum time for a transform is then used to calculate a scaled inverse time measurement based on the asymptotic number of floating-point operations required for computing the radix-2 Cooley-Tukey algorithm. This measurement is sometimes referred to as Cooley-Tukey equivalent GigaFLOPS (CTGs), but in this work MFLOPS is used, which are defined for a transform of size $N$ as $(5N \log_2 N)/t$, where $t$ is the time in microseconds for one transform, excluding initialization time [2]. In this case the measurement is not an actual FLOP count, but rather a rough measure of a particular implementation's efficiency relative to the radix-2 algorithm and the clock speed of the machine.

To measure the accuracy of a transform, the output is compared with an arbitrary-precision FFT computed on the same inputs, and relative RMS error is plotted. The inputs are pseudo-random in the range $[0.5, 0.5)$ and the arbitrary-precision FFT has over 40 decimal places of accuracy. When a transform has several variants, such as direction or radix, the accuracy is reported as being the worst of the results.

### B. Speed

Figure 9 shows the speed of benchmarks run on a 3.8GHz Intel "Sandy-bridge" Core i7-2600 running Linux. Figure 10 plots the results of benchmarks on a MacBook Pro 10,1 running OS X 10.8.1, equipped with a 3.3GHz Intel "Ivy-bridge" Core i7-3615QM. Figure 11 plots the results of benchmarks run on 3.4GHz Intel "Sandy-bridge" Core i5-2400 running Linux. All code in the above benchmarks was compiled with the latest available release of the Intel compilers.

In addition to Intel x86 machines, two ARM machines running iOS were benchmarked. Figure 12 shows the results of benchmarks run on an Apple A4 System-on-Chip (SoC), which implements the ARM Cortex-A8 core running at 1GHz.

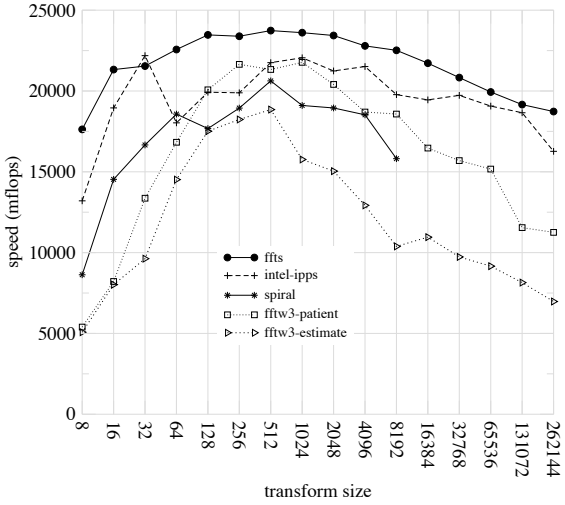



Fig. 9. Speed of single-precision power-of-two FFTs running on an Intel Core i7-2600 CPU with 8MB of cache running at 3.8GHz. Compiler: icc 12.1.5

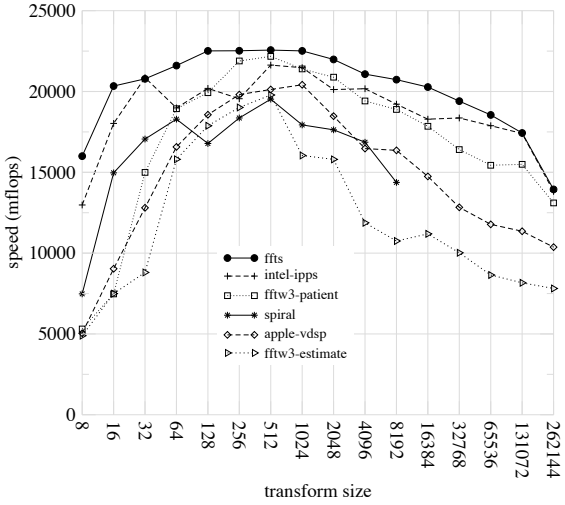



Fig. 10. Speed of single-precision power-of-two FFTs running on an Intel Core i7-3615QM CPU with 6MB of cache running at 3.3GHz. Compiler: icc 12.1.5

Figure 13 plots the results of benchmarks run on an Apple A5 SoC, which implements the ARM Cortex-A9 core running at 1GHz. All code in the ARM benchmarks was compiled with Apple clang 4.0.

The performance of all implementations drops for large transforms, illustrating the effects of the cache hierarchy.

There are some instances where FFTS is slower than other libraries. In many cases the 32-point transform is slower than Intel IPP, and on the Apple A5 the 8 and 16-point transforms are slower than FFTW. Because the smaller transforms are not widely used, they have not yet been highly optimized in FFTS. For $N < 32$, FFTS only uses scalar arithmetic, and for $N = 32$, and direct implementation would likely be faster (at the cost of code size). Furthermore, in [34] it was shown that the ordinary split-radix algorithm is marginally faster when the data fits in cache (very approximately, where $N < 16384$ or so for most machines), and this would likely result in FFTS being faster than FFTW running in patient mode for the size

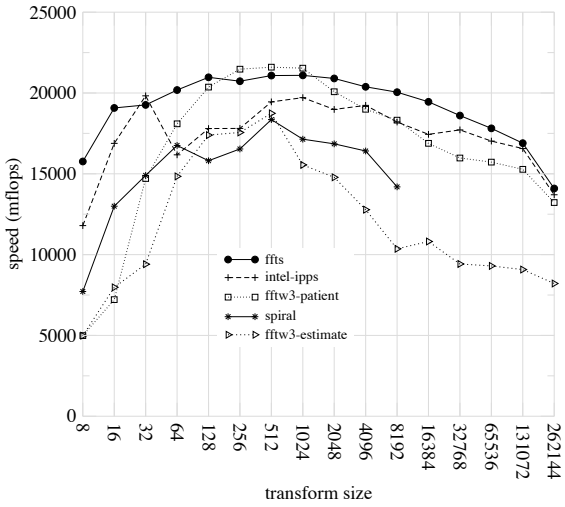



Fig. 11. Speed of single-precision power-of-two FFTs running on an Intel Core i5-2400 CPU with 6MB of cache running at 3.4GHz. Compiler: icc 12.1.5

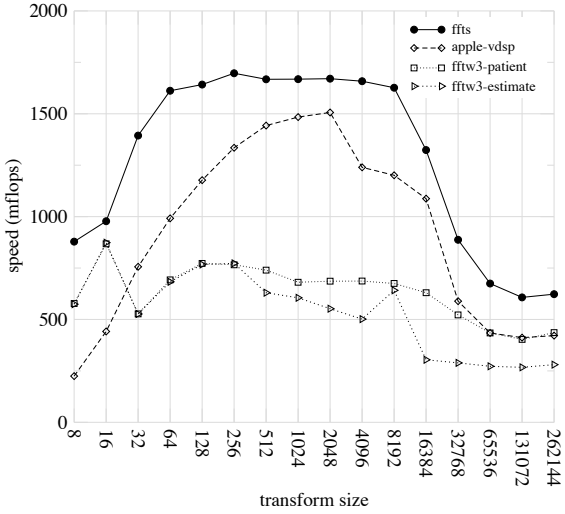



Fig. 12. Speed of single-precision power-of-two FFTs running on an Apple A4 SoC, which uses an ARM Cortex A8 with 512kB of cache, and clocked at 1GHz. Compiler: Apple clang 4.0

256, 512 and 1024 transforms in Figure 11.

### C. Initialization Time

Figure 14 shows the initialization times for codes running on a MacBook Pro 10,1 with an Intel i7-3615QM. On this machine, all libraries initialize within one second, with the exception of FFTW running in patient mode, which begins to take more than one second for transforms larger than 4096.

Graphs for other machines are similar.

### D. Accuracy

Figure 15 shows the accuracy of codes running on a MacBook Pro 10,1. The accuracy of all codes is within an acceptable range for single-precision arithmetic, and graphs for other machines are similar.

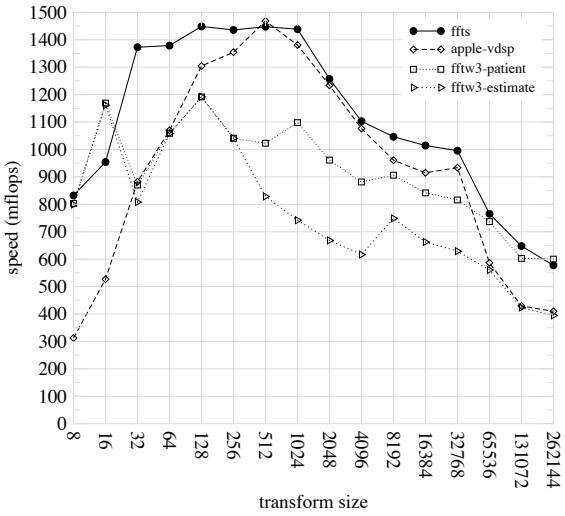



Fig. 13. Speed of single-precision power-of-two FFTs running on an Apple A5 SoC, which uses an ARM Cortex A9 with 1MiB of cache, and clocked at 1GHz. Compiler: Apple clang 4.0

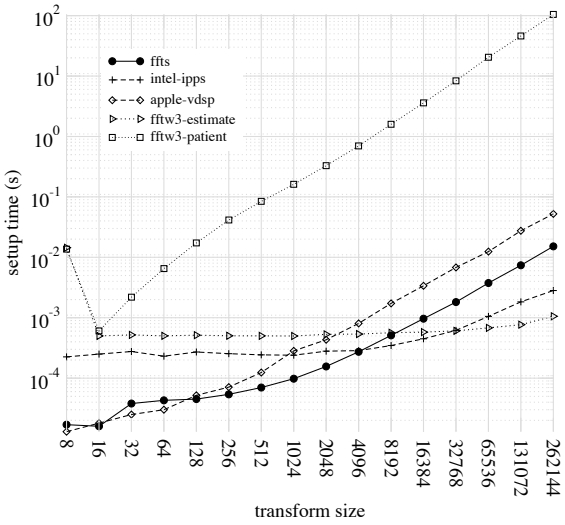



Fig. 14. Setup time of single-precision power-of-two FFTs running on an Intel Core i7-3615QM CPU running at 3.3GHz. Compiler: icc 12.5

TABLE I
CODE SIZE FOR A STATICALLY LINKED TEST PROGRAM.

| Name of FFT library | Size (kilobytes) |
|---|---|
| **FFTS** | **33.28** |
| Intel IPP | 510.70 |
| FFTW3 | 1733.16 |

### E. Code Size

Table I shows the size of a test program when statically linked with different FFT libraries, on OS X 10.8.1 with the Intel compilers. The test program computes a DFT, and does little else. Because FFTS dynamically generates code, the binary is small, however it should be noted that FFTS currently only computes a subset of the transforms supported by FFTW.

The size of the dynamically generated code varies according

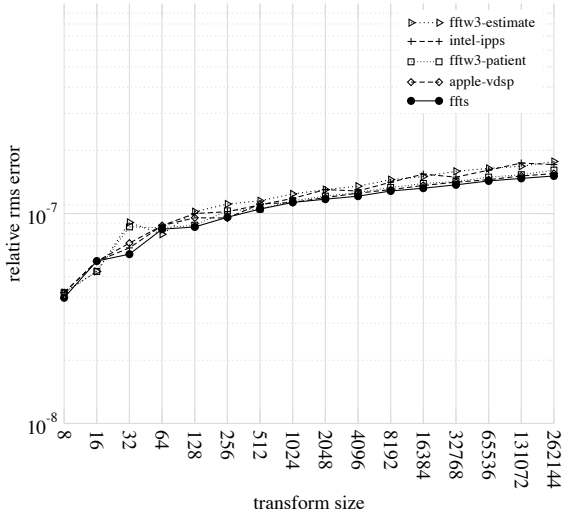



Fig. 15. Accuracy of single-precision power-of-two FFTs running on an Intel Core i7-3615QM CPU running at 3.3GHz. Compiler: icc 12.5

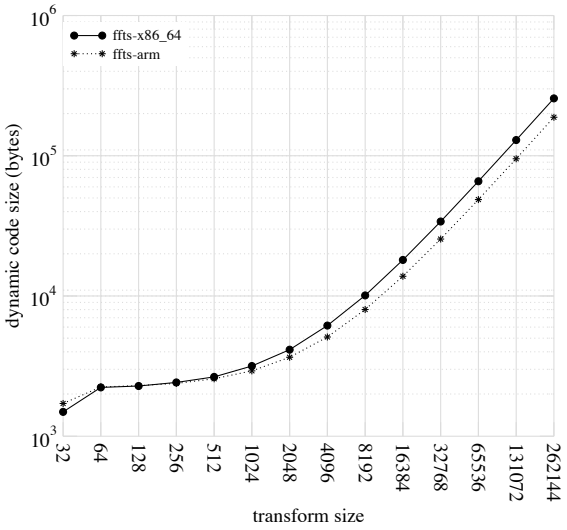



Fig. 16. Dynamic code size on ARM and x86-64.

to the size of the transform and the underlying machine, as shown in Figure 16.

## VIII. Conclusion and Future Work

This paper described FFTS, a DFT library which implements a new cache-oblivious FFT algorithm with runtime specialization. When evaluated with benchmarks run on recent Intel x86 and ARM machines, FFTS was faster than self-tuning libraries and even vendor tuned libraries in almost all cases, without requiring any machine-specific calibration or hardware parameters (due to the use of a cache oblivious algorithm) or an extensive library of codelets.

Program specialization is not a new technique, and has already been applied to the FFT in various different forms [3], [45]. However, FFTS is unique in that it takes specialization to the limit by generating specialized machine code at runtime rather than statically generating or interpreting specialized code.

Future work will apply these techniques to multi-threaded and multi-dimensional transforms, and add support for non-power-of-two sizes. Multi-dimensional transforms have already been implemented, but as there are special considerations for such transforms, that work falls outside the scope of this paper. Support for AVX and AVX2 will also be implemented in future.

FFTS has been released as open source code under a permissive license,[1] and runs on Linux, OS X and iOS, on the Intel x86 and ARM architectures. Support for other architectures and operating systems will be added in the future.

## Acknowledgment

We are grateful to Associate Professor J.A. Perrone for the many helpful discussions relating to this work, which was partly supported by a Marsden grant awarded to him by the Royal Society of New Zealand.

## References

[1] S. Kleene, N. de Bruijn, J. de Groot, and A. Zaanen, "Introduction to metamathematics," 1952.
[2] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
[3] S. G. Johnson and M. Frigo, "Implementing FFTs in practice," in *Fast Fourier Transforms*, ser. Connexions, C. S. Burrus, Ed. Houston TX: Rice University, September 2008, ch. 11.
[4] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.
[5] S. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *Signal Processing, IEEE Transactions on*, vol. 55, no. 1, pp. 111–119, 2006.
[6] M. Frigo, "A fast Fourier transform compiler," in *ACM SIGPLAN Notices*, vol. 34, no. 5. ACM, 1999, pp. 169–180.
[7] A. Ali and L. Johnsson, "UHFFT: A high performance DFT framework," 2006.
[8] A. Ali, L. Johnsson, and J. Subhlok, "Scheduling FFT computation on SMP and multicore systems," in *Proceedings of the 21st annual international conference on Supercomputing*. ACM, 2007, pp. 293–301.
[9] A. Ali, L. Johnsson, and D. Mirkovic, "Empirical auto-tuning code generator for FFT and trignometric transforms," in *ODES: 5th Workshop on Optimizations for DSP and Embedded Systems, in conjunction with International Symposium on Code Generation and Optimization (CGO)*, 2007.
[10] D. Mirkovic and L. Johnsson, "Automatic performance tuning in the UHFFT library," *Lecture notes in computer science*, pp. I–71, 2001.
[11] D. Mirkovic, R. Mahasoom, and L. Johnsson, "Adaptive software library for fast Fourier transforms," in *2000 International Conference on Supercomputing*, 2000, pp. 215–224.
[12] A. Ali, L. Johnsson, and J. Subhlok, "Adaptive computation of self sorting in-place FFTs on hierarchical memory architectures," *High Performance Computing and Communications*, pp. 372–383, 2007.
[13] J. Cooley and J. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.
[14] P. Duhamel and H. Hollmann, "Split radix FFT algorithm," *Electronics Letters*, vol. 20, no. 1, pp. 14–16, 1984.
[15] R. Yavne, "An economical method for calculating the discrete Fourier transform," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 115–125.
[16] I. Kamar and Y. Elcherif, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 25, p. 324, 1989.
[17] C. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968.

[1] Available at `http://www.cs.waikato.ac.nz/~ablake/ffts`

[18] L. Bluestein, "A linear filtering approach to the computation of discrete fourier transform," *Audio and Electroacoustics, IEEE Transactions on*, vol. 18, no. 4, pp. 451–455, 1970.
[19] A. Oppenheim, R. Schafer, J. Buck *et al.*, *Discrete-time signal processing*. Prentice hall Upper Saddle Riverˆ eN. JNJ, 1989, vol. 2.
[20] L. Rabiner, R. Schafer, and C. Rader, "The chirp z-transform algorithm," *Audio and Electroacoustics, IEEE Transactions on*, vol. 17, no. 2, pp. 86–92, 1969.
[21] C. Temperton, "Note on prime factor FFT algorithms," *J. Comput. Phys.*, vol. 1, 1983.
[22] R. Rivest and C. Leiserson, "Introduction to algorithms," 1990.
[23] R. C. Singleton, "On computing the fast Fourier transform," *Commun. ACM*, vol. 10, pp. 647–654, October 1967.
[24] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.
[25] V. I. Kelefouras, G. Athanasiou, N. Alachiotis, H. E. Michail, A. Kritikakou, and C. E. Goutis, "A methodology for speeding up fast Fourier transform focusing on memory architecture utilization," *IEEE Transactions on Signal Processing*, vol. 59, no. 12, pp. 6217–6226, 2011.
[26] F. Franchetti and M. Puschel, "A SIMD vectorizing compiler for digital signal processing algorithms," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. IEEE, 2002, pp. 20–26.
[27] F. Franchetti, Y. Voronenko, and M. Puschel, "FFT program generation for shared memory: SMP and multicore," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 51–51.
[28] F. Franchetti, Y. Voronenko, and M. Püschel, "A rewriting system for the vectorization of signal transforms," *High Performance Computing for Computational Science-VECPAR 2006*, pp. 363–377, 2007.
[29] S. Kral, F. Franchetti, J. Lorenz, C. Ueberhuber, and P. Wurzinger, "FFT compiler techniques," in *Compiler Construction*. Springer, 2004, pp. 2725–2725.
[30] M. Puschel, J. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. Johnson, "SPIRAL: A generator for platform-adapted libraries of signal processing alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, p. 21, 2004.
[31] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.
[32] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *The Journal of Machine Learning Research*, vol. 3, pp. 887–919, 2003.
[33] L. Morris, "Automatic generation of time efficient digital signal processing software," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 25, no. 1, pp. 74–79, 1977.
[34] A. Blake, "Computing the fast Fourier transform on SIMD microprocessors," Ph.D. dissertation, University of Waikato, New Zealand, 2012.
[35] R. A. Gopinath, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 25, p. 1084, 1989.
[36] A. M. Krot and H. B. Minervina, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 28, p. 1143, 1992.
[37] H.-S. Qian and Z.-J. Zhao, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 26, p. 541, 1990.
[38] D. Bernstein, "The tangent FFT," *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pp. 291–300, 2007.
[39] C. Burrus and T. Parks, *DFT/FFT and Convolution Algorithms: theory and Implementation*. John Wiley & Sons, Inc., 1991.
[40] H. Sorensen, M. Heideman, and C. Burrus, "On computing the split-radix FFT," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, no. 1, pp. 152–156, 1986.
[41] A. Skodras and A. Constantinides, "Efficient computation of the split-radix FFT," *IEE proceedings. Part F. Radar and signal processing*, vol. 139, no. 1, pp. 56–60, 1992.
[42] N. Jones, C. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.
[43] D. Takahashi, "An extended split-radix FFT algorithm," *Signal processing letters, IEEE*, vol. 8, no. 5, pp. 145–147, 2001.
[44] "BenchFFT, a program to compare the performance and accuracy of many different FFT implementations." URL: http://www.fftw.org/benchfft/, November 2011.
[45] F. Franchetti and M. Puschel, "Short vector code generation and adaptation for DSP algorithms," in *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, vol. 2. IEEE, 2003, pp. II–537.

**Anthony M. Blake** received his PhD in computer science at the University of Waikato, New Zealand in 2012. He is currently a post-doctoral Research Fellow in the Department of Computer Science at the University of Waikato. His research interests include high-performance computing, computer architecture and signal processing.

**Ian H. Witten** is Professor of Computer Science at the University of Waikato in New Zealand. His research interests include language learning, information retrieval, and machine learning. He has published widely, including several books, such as Managing Gigabytes (1999), Web Dragons (2007), How to Build a Digital Library (2009), and Data Mining (2011). He is a Fellow of the ACM and of the Royal Society of New Zealand.

**Michael J. Cree** received his PhD in electrical and electronic engineering at the University of Canterbury, New Zealand in 1994. After almost three years as a researcher at the University of Aberdeen, Scotland, he took up a position at the University of Waikato, New Zealand, where he is now Senior Lecturer in the School of Engineering. His research interests include medical imaging, range imaging, and image processing.