AUTOMATICALLY GENERATED HIGH-PERFORMANCE CODE FOR DISCRETE WAVELET TRANSFORMS

Aca Gačić, Markus Püschel, and José M. F. Moura

Department of Electrical and Computer Engineering
Carnegie Mellon University,
Pittsburgh, U.S.A.

ABSTRACT

A growing number of performance-critical DSP application use the discrete wavelet transform (DWT), thus prompting the need for highly efficient DWT software implementations. Unfortunately, the rapid evolution of computing platforms and compiler technology makes carefully hand-tuned code obsolete almost as fast as it is written. In this paper we describe our work on the automatic generation of DWT implementations that are tuned to a given platform. Our approach captures the various DWT algorithms in a concise mathematical framework that enables the integration of DWTs into the SPIRAL code generation system. Experiments show the quality of our automatically generated code and provide interesting insights; for example, the fastest code differs between platforms and is usually based on a non-obvious combination of DWT algorithms.

1. INTRODUCTION

DSP applications typically require high-performance implementations. Software developers usually fine-tune their implementations to utilize the specific features of the target platform. However, hand-coding is tedious and time-consuming, and it requires expert knowledge of both algorithms and the computer architecture. Furthermore, the ever-changing hardware and compiler technologies require frequent re-implementation. This problem has led to a growing interest in automatic generation and tuning of code. There have been several efforts in this direction, mainly involving basic linear algebra and DSP kernels. For example, the SPIRAL system automatically generates platform-adapted code for several DSP transforms [1, 2]. Apart from common DSP transforms, increasingly more DSP applications use wavelet transforms; examples include compression, communications, and numerical analysis, just to name a few. Automatic code tuning for fast wavelet kernels is hence an important problem still waiting to be addressed.

In this paper we focus on the *automatic generation* of *high-performance* software implementations for the class of discrete wavelet transforms (DWTs). Previously, we extended the SPIRAL framework to generate efficient code for FIR filters [3], which serves as a foundation of the present work. In this paper we reformulate existing DWT algorithms into the mathematical form required for integration into the SPIRAL system, which we use to enable automatic code generation. Experimental results show the quality of our generated code and bring a few insights: the code depends on the underlying platform, and the choice of the algorithm depends on the size of the transform.

This work was supported by NSF awards 0234293 and 0310941.

2. SPIRAL

The SPIRAL system generates optimized platform-adapted implementations for a variety of important DSP transforms. SPIRAL's framework is based on four key concepts: *transforms, rules, rule trees,* and *formulas*.

Transforms. A *transform* of a finite discrete-time sequence is computed as a matrix-vector product. The transform is represented as a matrix, typically parameterized by its size. For example, the discrete Fourier transform (DFT) of size n is given by

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{k,\ell=0,\ldots,n-1}, \quad \omega_n = e^{-2\pi j/n}. \tag{1}$$

Rules. *Breakdown rules*, or simply *rules*, structurally decompose a transform into other transforms and/or primitive matrices. Rules are represented by a set of mathematical constructs such as the *Kronecker* or the *tensor* product $A \otimes B = [a_{k,\ell} \cdot B]$, where $A = [a_{k,\ell}]$ and the *direct sum* of matrices $A \oplus B = \left[\begin{smallmatrix} A & \\ & B \end{smallmatrix}\right]$, and with primitive symbols representing matrices that are not transforms, such as the stride permutation

$$\mathrm{L}_r^{rs} : \; j \leftarrow j \cdot r \mod (rs-1). \tag{2}$$

For example, we can write the rule for the class of mixed radix Cooley-Tukey algorithms for a DFT of size $n = pq$ [4],

$$\mathrm{DFT}_n = (\mathrm{DFT}_p \otimes \mathrm{I}_q)\, \mathrm{T}_q^n\, (\mathrm{I}_p \otimes \mathrm{DFT}_q)\, \mathrm{L}_p^n, \; n = p \cdot q, \tag{3}$$

where I_n is the $n \times n$ identity matrix, T_q^n is a diagonal matrix of complex roots of unity, and L_p^n is a stride permutation [4].

Rule trees. Rules such as (3) are applied recursively and combined with other rules as many times as possible. This gives rise to a tree structure called a *rule tree*. When no more rules are applicable then the base case is reached and the tree is said to be fully expanded. An algorithm in SPIRAL is uniquely represented by a fully expanded rule tree.

Formulas. Formulas are mathematical expressions that represent rule trees. A formula is obtained by applying rules to a transform by replacing the transforms in formulas by the right-hand side of the applicable rules. We can, thus, expand (3) as

$$\begin{aligned} \mathrm{DFT}_8 = \; & (\mathrm{DFT}_2 \otimes \mathrm{I}_4)\, \mathrm{T}_4^8 \\ & \left((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}_2^4\, (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}_4^2\right) \mathrm{L}_8^2, \end{aligned} \tag{4}$$

Fully expanded formulas uniquely represent fully expanded rule trees and the corresponding algorithms.

SPIRAL system. The architecture of SPIRAL is displayed in Figure 1. For a given transform, the *formula generator* generates

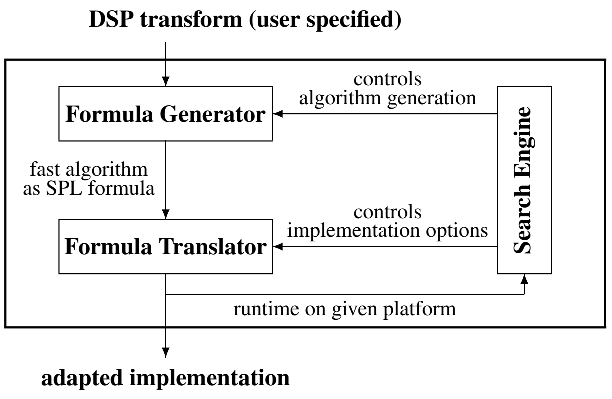

DSP transform (user specified)
Formula Generator
controls algorithm generation
Search Engine
fast algorithm as SPL formula
Formula Translator
controls implementation options
runtime on given platform
adapted implementation

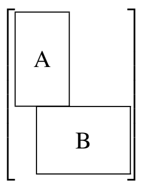
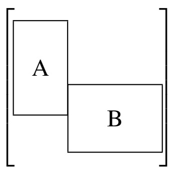
Fig. 1. The architecture of SPIRAL.

one or several out of many possible formulas, or algorithms, in the mathematical domain-specific language called SPL. The *formula translator* parses the SPL program and recursively matches the formulas with the set of predefined translation templates, which are combined to produce an actual C program for the formula. At this stage, various implementation choices are provided, such as the degree of loop unrolling. The produced code is measured and the runtime is used by the *search engine* to control the generation of new algorithms in the formula generator and possibly different implementation options in the formula translator. Thus, for a given transform, the search engine navigates through the space of possible algorithms and their implementations, and evaluates them to find the optimized solution suited to the target platform. SPIRAL employs various search techniques, including dynamic programming, random search, and evolutionary search [5].

Our goal is to capture DWTs in SPIRAL's framework to enable automatic code generation.

3. DWT RULES AND ALGORITHMS

To capture DWT algorithms in SPIRAL's rule framework, we first need the FIR filter transform $F_n(\mathbf{h})$ as a building block , where n is the number of inputs and $\mathbf{h}$ is the column vector of filter coefficients [3]. Second, we need two new mathematical operators, already introduced in [3], namely the *row overlapped direct sum* and the *column overlapped direct sum* of matrices A and B, respectively defined by

$A \oplus_k B =$  , $A \oplus^k B =$  , (5)

where the parameter k provides the number of overlapping columns or rows, respectively. In particular, $A \oplus_0 B = A \oplus^0 B = A \oplus B$. Also, $(A \oplus_k B)^T = A^T \oplus^k B^T$.

Further, we define the *row overlapped tensor product* by

$$\mathrm{I}_s \otimes_k A = \underbrace{A \oplus_k A \oplus_k \cdots \oplus_k A}_{s\text{-fold}}, \tag{6}$$

and the column overlapped tensor product analogously.

DWT definition. A two-band discrete-time wavelet transform (DWT) of the finite sequence $\{x_n\}$ of length N is defined using the following recursive relations, also known as *Mallat's algorithm* for the DWT [6]:

$$c_{j-1,n} = \sum_m h_{m-2n}\, c_{j,m}, \quad d_{j-1,n} = \sum_m g_{m-2n}\, c_{j,m}, \tag{7}$$

where $j = 1, \ldots, J$ is the *scale* or the *level* of the decomposition and $n = 0, \ldots, N-1$ is the *shift*.

The starting coefficients at the highest level $J = \log_2 N$ represent the input sequence, i.e., $\{x_n\} = \{c_{J,n}\}$, and the output of the DWT is $\{y_n\} = \{\{d_{J-1,k}\}, \{d_{J-2,k}\}, \ldots, \{d_{0,k}\}, \{c_{0,k}\}\}$. The coefficients $\{c_{j,n}\}$ and $\{d_{j,n}\}$ are known as the *scaling* and the *wavelet* coefficients at scale j, respectively, whereas $\{h_n\}$ and $\{g_n\}$ are the scaling and the wavelet function coefficients. The relation in (7) can be seen as the filter bank shown in Figure 2.

We capture the Mallat's algorithm in the following rule

$$\begin{aligned}\mathrm{DWT}_n(\mathbf{h},\mathbf{g}) &= \left(\mathrm{DWT}_{n/2}(\mathbf{h},\mathbf{g}) \oplus \mathrm{I}_{n/2}\right) \cdot (\downarrow \mathbf{2}_n) \cdot \\ &\quad \left(F_{N/2}(\mathbf{h}) \oplus_{N+k-1} F_{N/2}(\mathbf{g})\right) \cdot \mathrm{E}^*_{n,l,r},\end{aligned}$$

where $(\downarrow \mathbf{2}_n) = (\mathrm{I}_n \oplus^n \mathbf{0}_n)\, \mathrm{L}^{2n}_2$ is the downsample operator, $\{h_0, \ldots, h_{k-1}\}$ are scaling function coefficients, $\{g_0, \ldots, g_{k-1}\}$ are wavelet function coefficients from (7), and $E^*_{n,l,r}$ is a type "$*$" boundary extension operator that extends the input by l points to the left and r points to the right. The DWT is decomposed into a smaller DWT and two FIR filter transforms. The downsampler throws away half of the results, so we fuse it into the filters to obtain *fused Mallat* rule.

$$\mathrm{DWT}_n(\mathbf{h},\mathbf{g}) = \left(\mathrm{DWT}_{n/2}(\mathbf{h},\mathbf{g}) \oplus \mathrm{I}_{n/2}\right) \cdot H_n(\mathbf{h},\mathbf{g}) \cdot \mathrm{E}^*_{n,l,r}, \tag{8}$$

where the matrix H_n represents one stage of the wavelet filter bank

$$H_n(\mathbf{h},\mathbf{g}) = \begin{bmatrix} \mathrm{I}_{n/2} \otimes_{k-2} [\, h_0 \;\; h_1 \;\; \ldots \;\; h_{k-1} \,] \\ \mathrm{I}_{n/2} \otimes_{k-2} [\, g_0 \;\; g_1 \;\; \ldots \;\; g_{k-1} \,] \end{bmatrix}. \tag{9}$$

Downsampling can be done before filtering to save operations. This is achieved through the polyphase representation. The signal is split into even and odd samples. Each sequence is filtered by subsampled filters, which results in the reduction of arithmetic cost by two (see Figure 2).

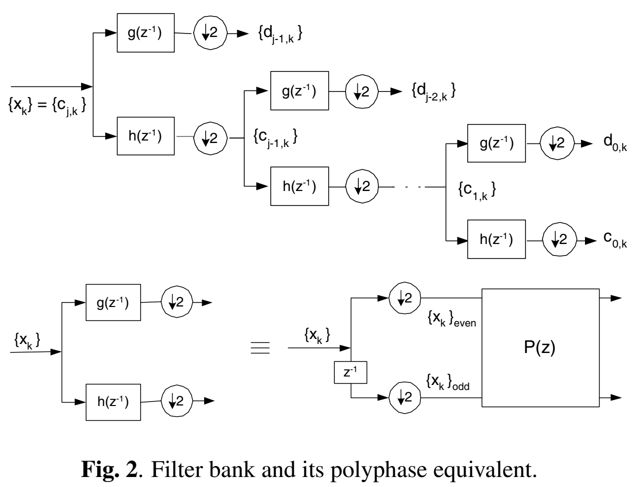


Fig. 2. Filter bank and its polyphase equivalent.

The polyphase matrix consists of four filters, the even and the odd subsampled versions of the lowpass and the highpass filters.

$$P(z^{-1}) = \begin{bmatrix} h_e(z^{-1}) & h_o(z^{-1}) \\ g_e(z^{-1}) & g_o(z^{-1}) \end{bmatrix}, \quad (10)$$

The polyphase decomposition is represented by the following rule

$$\begin{aligned} H_n(\mathbf{h},\mathbf{g}) &= \Big[(F^{\mathrm{T}}_{n/2}(\mathbf{h}_o) \oplus^{n/2} F^{\mathrm{T}}_{n/2}(\mathbf{h}_e)) \oplus_{n+k-2} \\ &\quad (F^{\mathrm{T}}_{n/2}(\mathbf{g}_o) \oplus^{n/2} F^{\mathrm{T}}_{n/2}(\mathbf{g}_e))\Big] \cdot \mathrm{L}_2^{n+k-2} \end{aligned} \quad (11)$$

$$\mathbf{h}_e = [h_0, h_2, \ldots, h_{k-2}]^{\mathrm{T}} \quad \mathbf{h}_o = [h_1, h_3, \ldots, h_{k-1}]^{\mathrm{T}}$$
$$\mathbf{g}_e = [g_0, g_2, \ldots, g_{k-2}]^{\mathrm{T}} \quad \mathbf{g}_o = [g_1, g_3, \ldots, g_{k-1}]^{\mathrm{T}}$$

The polyphase rule is a gateway to implementations in the frequency domain through the DFT-based rules for the four FIR filter transforms [3].

Arithmetic cost can be further reduced by using the *lifting scheme* [7]. The method is based on the factorization of the polyphase matrix into the product of upper and lower triangular polyphase matrices with short filters $s_i(z)$ and $t_i(z)$. These matrices are known as the primal (predict) and the dual (update) lifting steps. The lifting scheme reduces the cost for computing the polyphase matrix asymptotically by 50%. It is used, for example, in JPEG2000 standard.

We define a rule that captures this decomposition:

$$\begin{aligned} H_n(\mathbf{h},\mathbf{g}) &= (K \cdot \mathrm{I}_{n/2} \oplus 1/K \cdot \mathrm{I}_{n/2}) \\ &\quad \underbrace{\Big((\mathrm{I}_{n/2} \oplus^{n/2} F^T_{n/2}(\mathbf{s}_i) \oplus_{n+|\mathbf{s}_i|} \mathrm{I}_{n/2}\Big)}_{S_i} \\ \cdots &\quad \underbrace{\Big(\mathrm{I}_{n/2} \oplus_{n+|\mathbf{t}_0|} (F^T_{n/2}(\mathbf{t}_0) \oplus^{n/2} \mathrm{I}_{n/2})\Big)}_{T_0} \mathrm{L}_2^{n+k-2}, \end{aligned} \quad (12)$$

where $\mathbf{s}_i$ and $\mathbf{t}_i$ represent the primal and dual filter coefficients obtained by decomposing $\mathbf{h}$ and $\mathbf{g}$ using the Euclidean algorithm, and $|\mathbf{x}|$ is the length of $\mathbf{x}$.

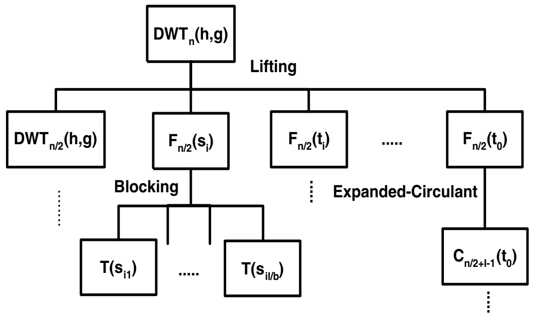


Fig. 3. A rule tree generated from DWT and FIR rules

Combined application of rules (8)-(12), together with the rules for FIR filters, provide a very large space of different DWT algorithms that SPIRAL uses to search for an optimized implementation. An example is the algorithm represented by the rule tree shown in Figure 3. The lifting rule (12) is applied to decompose H_n into smaller filters. Different lifting step filters are in turn decomposed either into blocks using the blocking rule or into circulant matrices that are decomposed further (e.g, using DFT-based rules) [3].

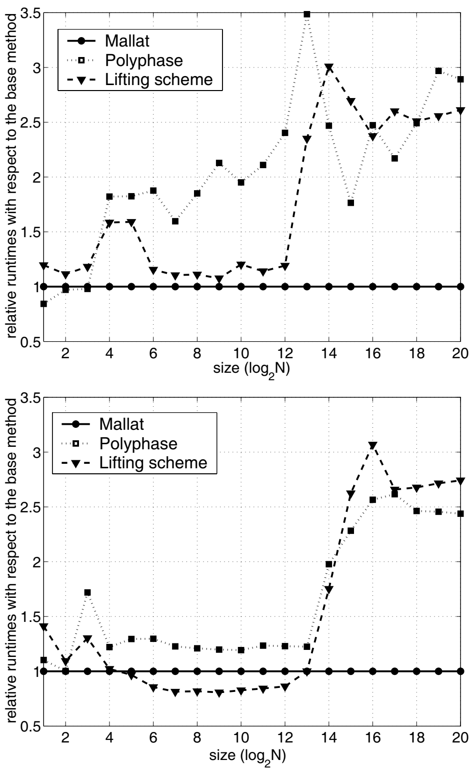


Fig. 5. Comparison of best found implementations using 3 methods for Daubechies D_4 (top) and D_{9-7} (bottom).

4. EXPERIMENTAL RESULTS

We included the framework for DWTs in SPIRAL and performed experiments with a large number of generated algorithms across two platforms. We designed the experiments so as to investigate the efficiency of certain types of algorithms and their implementations on different machines. Besides the algorithmic degrees of freedom, we included the degree of loop unrolling and code blocking strategies in the search.

We ran experiments on two computer platforms: Intel Xeon (1.7 GHz, Linux) and AMD Athlon XP (2.1 GHz, Linux), referred to as Xeon and Athlon, respectively, using gcc 3.2.1 under Linux with compiler flags set to "–O6 –fomit-frame-pointer –malign-double –fstrict-aliasing –mcpu=pentiumpro". We considered single-level DWTs with orthogonal Daubechies D_4 and D_{30} wavelets, and the biorthogonal D_{9-7} used in the JPEG2000 standard [8], all with symmetric signal extensions. SPIRAL automatically generated all algorithms and the corresponding code, both found by the dynamic programming search provided by SPIRAL.

We compared three classes of algorithms determined by the choice of the top-level rule (8), (11), or (12). For each class SPIRAL generates and searches a large number of algorithms. Based on the choice of the top-level rule, we have:

- *Direct methods*. We use the Fused-Mallat rule (8) to compute $H_n(\mathbf{h},\mathbf{g})$. The matrix $H_n(\mathbf{h},\mathbf{g})$ is implemented using the best found blocking strategy for a matrix-vector multiplication.
- *Polyphase methods*. We use the Polyphase rule (11) to decompose $H_n(\mathbf{h},\mathbf{g})$ into four shorter FIR filters. This method provides a way to implement $H_n(\mathbf{h},\mathbf{g})$ in the frequency domain.

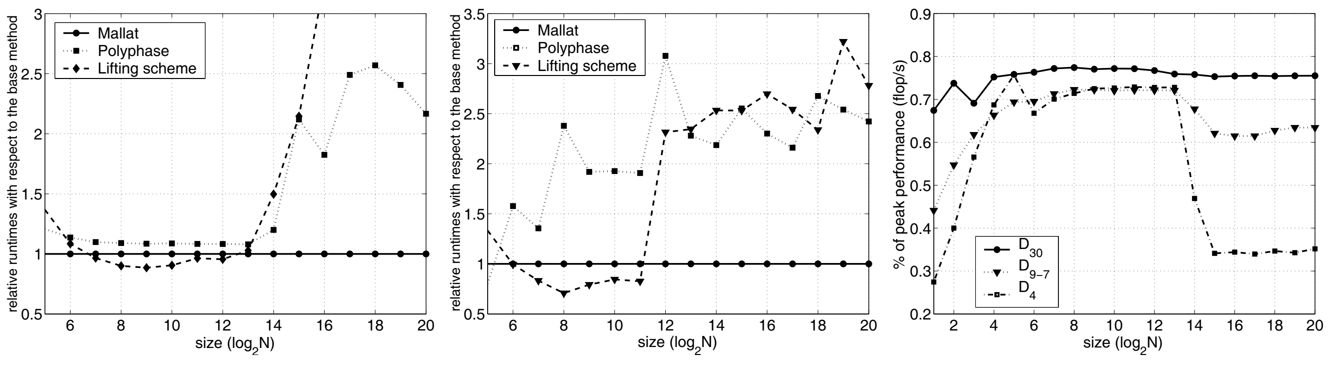


Fig. 4. Comparison of best found implementations for Daubechies D_{30} on Xeon (left) and Athlon (center). Flop/s performance for different DWTs on Xeon (right).

- *Lifting scheme*. The Rule (12) is used to decompose $H_n(\mathbf{h}, \mathbf{g})$ into lifting steps, which reduces the arithmetic cost.

An example of the lifting scheme class algorithm is the one shown in Figure 3.

Different top-level methods (rules). In the first experiment, we generated code for the DWT using the D_4 and D_{9-7} wavelets for input sizes $2^1 \leq N \leq 2^{20}$. The runtimes in Figure 5 are shown relative to Mallat's method (higher is worse), which serves as baseline. We observe that for D_4 the lifting method is inferior for all sizes, even though its arithmetic cost is only about 65% of the direct methods. A longer critical path and more involved data access patterns of the lifting scheme are the likely reasons for this discrepancy. For D_{9-7}, the lifting proves to be better for the range $2^4 \leq N \leq 2^{13}$ offering a speed-up of up to 20%. For smaller sizes there is an extension overhead since the input size is smaller than the filter length, the case rarely found in practice. For sizes above 2^{13}, in both cases, the temporary vectors used in the lifting scheme deteriorate cache performance. For both wavelets, there is no advantage of using polyphase methods since the filter lengths are too short for the frequency-based techniques to be efficient [3].

Different platforms. In the second experiment, we generated code for D_{30} wavelets for two different platforms. Figure 4 shows again the relative runtimes obtained for the three methods on Xeon (left) and Athlon (center). We emphasize the difference in the two graphs. The lifting scheme is preferable for sizes 2^7 to 2^{13} on Xeon, and from 2^6 to 2^{11} on Athlon. The difference arises from considerably different architectures and compiler behavior on the two platforms. This shows that the performance is not portable and that the search is necessary to automatically adjust implementations to the new environment.

Performance. In the third experiment, run on Xeon, we measured the percentage of the theoretical peak performance for our generated code; the performance is measured in floating point operations per second (flops/s). The right plot in Figure 4 shows the result for the direct methods for three wavelet transforms implemented on Xeon. The performance is highest (about 75% of the peak performance) for the longest wavelet D_{30} and does not deteriorate for large sizes. In contrast, the short wavelets experience a performance drop at the cache boundary, more pronounced for the shortest D_4 wavelet. It is interesting to note that the lifting scheme has lower flops/s (not shown) but also lower computational cost, which may lead to a faster runtime.

Conclusion. We designed a new framework that efficiently captures the existing DWT algorithms in a concise notation suitable for machine representation and code generation, and integrated it into SPIRAL. We used SPIRAL to automatically generate high quality code that is adapted to the target platform. The best found code 1) is not obvious for a human programmer since it involves different algorithmic methods and variable degrees of code blocking and unrolling that are platform dependent; 2) can not be reliably predicted from the arithmetic cost and the flops/s performance; and 3) is considerably different across different platforms. It is therefore very hard to draw conclusions that apply in general to different applications and platforms, which justifies our approach of automatic search for the best suited solution.

5. REFERENCES

[1] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, "SPIRAL: Automatic Library Generation and Platform-Adaptation for DSP Algorithms," 1998, http://www.spiral.net.

[2] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," to appear in *Journal of High Performance Computing and Applications*, 2004.

[3] A. Gačić, M. Püschel, and J. M. F. Moura, "Fast Automatic Implementation of FIR Filters," in *IEEE Proc. ICASSP*, April 2003, vol. 2, pp. 541–544.

[4] R. Tolimieri, M. An, and C. Lu, *Algorithms for discrete Fourier transforms and convolution*, Springer, 2nd edition, 1997.

[5] B. Singer and M. Veloso, "Automating the Modeling and Optimization of the Performance of Signal Transforms," *IEEE Trans. Signal Proc.*, vol. 5, no. 8, pp. 2003–2014, 2002.

[6] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. on Pattern Recognition and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, July 1989.

[7] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, 1998.

[8] I. Daubechies, *Ten Lectures on Wavelets*, vol. 61 of *CBMS-NSF Regional Conference Series in Applied Mathematics*, SIAM, Philadelphia, 1992.