FAST AUTOMATIC SOFTWARE IMPLEMENTATIONS OF FIR FILTERS

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

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, U.S.A.
{agacic, pueschel, moura}@ece.cmu.edu

ABSTRACT

SPIRAL is a generator for platform-adapted libraries of DSP transform algorithms. SPIRAL represents and automatically generates fast algorithms as mathematical formulas and translates them into programs. Adaptation is achieved by searching in the space of algorithmic and coding alternatives for the fastest implementation. In this paper we extend SPIRAL to generate platform-adapted implementations of FIR filters. First, we present various filter algorithms and introduce the mathematical constructs needed to include them into SPIRAL's architecture. Then we use SPIRAL to find fast filter implementations. The results show runtime improvements to a standard loop implementation of up to 70% using different blocking techniques. Further, we show that the usefulness of frequency-domain methods is not determined by the number of operations.

1. INTRODUCTION

Designers of fast digital signal processing (DSP) algorithms are usually concerned with reducing their arithmetic cost. However, it is well known that the actual runtime of a software implementation of these algorithms is critically dependent on the architecture, in particular on the memory hierarchy, of the computing platform, and on the data flow pattern of the algorithm. Choosing the right algorithm is a difficult problem that requires extensive testing. Usually, the resulting code is obtained by hand-tuning to the target platform. In many cases, the same code does not yield high performance when used on a different machine.

SPIRAL, [1, 2], is a generator of libraries of fast software implementations for DSP transforms. The SPIRAL generated code is optimized and *tuned* to the actual computing architecture, and it is competitive with the best code available developed by human experts. As an example, [3] shows that the DFT (discrete Fourier transform) code generated automatically by SPIRAL is faster than the code provided by Intel's Math Kernel library for DFT sizes up to 2^{13}. Besides the DFT, SPIRAL generates optimized code for the trigonometric transforms like the DCT and DST, the Hartley and the Walsh-Hadamard transform, and many others.

In this paper, we extend SPIRAL to generate optimal platform-adapted code for finite impulse response (FIR) filters. We demonstrate the quality of the generated code on two common computer platforms: the Intel Pentium 4 and the SUN UltraSPARC II.

This work was supported by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.

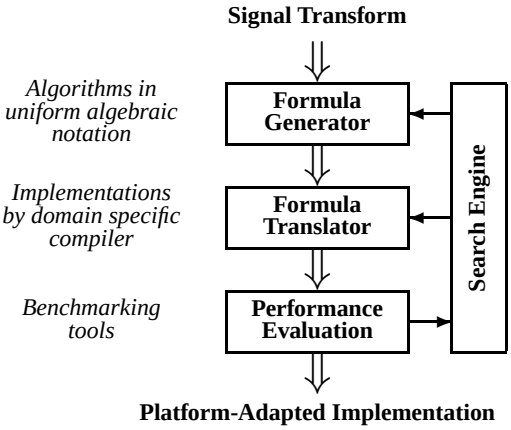

Fig. 1. The architecture of SPIRAL.

2. SPIRAL

The architecture of SPIRAL is shown in Fig. 1, [1, 2]. A given transform is symbolically manipulated by the formula generator in SPIRAL to generate many alternative algorithms for this transform, represented as mathematical formulas. These are obtained by using the properties of the constructs and by selective use of recursive transformations, which are called breakdown rules in SPIRAL. The formula generator outputs a description of each algorithm using a Lisp-type SPIRAL proprietary language called SPL (signal processing language). The SPL program is then automatically translated into a high-level language program (C or Fortran) in the formula translator. The resulting code is timed, and the runtime is used by a search engine to intelligently control the generation of new algorithms. Furthermore, the search engine controls implementation options, such as the degree of loop unrolling. Iteration of this loop yields a platform-adapted implementation for the transform input into SPIRAL.

We now explain in more detail the structure of SPIRAL, which, at the high level, captures numerous different transforms and their algorithms in a compact mathematical framework that uses only a small number of constructs and primitive symbols.

Transforms. A *transform* in SPIRAL is a class of parameterized matrices that are typically highly structured. An example of a transform is the DFT on input size n, given by

$$\mathrm{DFT}_n = \left[e^{-2\pi jk\ell/n}\right]_{k,\ell=0,\ldots,n-1}. \qquad (1)$$

Rules. A transform is structurally decomposed using break-

down *rules*. Rules are represented by using a set of mathematical constructs and primitives. As an example, the tensor product is a construct often found in rules for DSP transforms.

$$A \otimes B = [a_{k,\ell} \cdot B], \quad \text{where } A = [a_{k,\ell}].$$

A well-known example of a rule is the mixed-radix Cooley-Tukey (CT) rule for a DFT of size $n = pq$,

$$\mathrm{DFT}_n = (\mathrm{DFT}_p \otimes \mathrm{I}_q)\, \mathrm{T}^n_q\, (\mathrm{I}_p \otimes \mathrm{DFT}_q)\, \mathrm{L}^n_p,\ n = p \cdot q, \quad (2)$$

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

Algorithms. An *algorithm* is obtained by recursive application of applicable rules until a base case is reached (i.e., no further rules can be applied). For example, if n is a 2-power and only (2) is used, then the base case is DFT_2; one possible algorithm generated this way is the radix-2 decimation-in-time (DIT) algorithm [4], for $n = 8$ given by

$$\begin{aligned} \mathrm{DFT}_8 = \quad & (\mathrm{DFT}_2 \otimes \mathrm{I}_4)\, \mathrm{T}^8_4 \\ & \big((\mathrm{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2\, (\mathrm{I}_2 \otimes \mathrm{DFT}_2)\, \mathrm{L}^4_2\big)\, \mathrm{L}^2_8. \end{aligned} \quad (3)$$

The number of algorithms for each transform is very large and depends on the transform size and the number of available rules. The rule framework makes algorithm generation fast and efficient [2]. SPIRAL uses this set of different algorithms as a search space to find a platform-adapted implementation. Since SPIRAL uses high-level descriptions of the algorithms for DSP transforms, it is easily extended with new transforms and new rules.

This paper extends SPIRAL to automatically generate fast implementations for FIR filters. Filtering is interpreted as matrix-vector multiplication, so that FIR filtering becomes a matrix transform. Then, we represent fast algorithms as breakdown rules and construct the algorithm space. We start by introducing the set of necessary mathematical constructs and transforms.

3. TRANSFORMS AND CONSTRUCTS

FIR filter. A k-tap *FIR filter* is usually represented as the *linear convolution*

$$y_m = \sum_{i=0}^{k-1} h_i x_{m-i}, \quad (4)$$

where $\mathbf{h} = [h_0\, h_1 \ldots h_{k-1}]$ are the filter coefficients.

For input size n, (4) can be represented as a matrix-vector product $\mathbf{y} = F_n(\mathbf{h})\mathbf{x}$, where $F_n(\mathbf{h})$ is a rectangular $(n+k-1) \times n$ matrix, which we call *FIR filter transform*,

$$F_n(\mathbf{h}) = \underbrace{\begin{bmatrix} h_0 & & & & \\ h_1 & h_0 & & & \\ h_2 & h_1 & h_0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ h_{k-1} & & & & h_0 \\ & h_{k-1} & \vdots & & \vdots \\ & & h_{k-1} & & \\ & & & \ddots & \vdots \\ & & & & h_{k-1} \end{bmatrix}}_{n \text{ columns}}. \quad (5)$$

We call k the filter length, and n the filter size. Our goal is to generate fast implementations of $F_n(\mathbf{h})$.

Circulant. The cyclic convolution on n points is given by

$$y_m = \sum_{i=0}^{n-1} a_i x_{(m-i) \bmod n}. \quad (6)$$

In matrix representation, the cyclic convolution is given by the *circulant transform*

$$C(\mathbf{a}) = \left[a_{(i-j) \bmod n}\right]_{i,j=0,\ldots,n-1}, \quad (7)$$

where $\mathbf{a} = [a_0\, a_1 \ldots a_{n-1}]$ are the convolution coefficients.

Toeplitz. The *Toeplitz transform* has the form

$$T(\mathbf{b}) = [b_{i-j}]_{i,j=0,\ldots,n-1}. \quad (8)$$

where the vector $\mathbf{b} = \left[b_{-(n-1)} \ldots b_0 \ldots b_{n-1}\right]$ contains the defining entries from the first column $[b_0, \ldots, b_{-(n-1)}]^T$, and the first row $[b_0, \ldots, b_{n-1}]$.

Direct Sum. The direct sum of two matrices A and B is defined as

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}. \quad (9)$$

Overlapped Direct Sum. We define the *row overlapped direct sum* and the *column overlapped direct sum* of matrices A and B as, respectively,

$$A \oplus_k B = \begin{bmatrix} \boxed{\text{A}} & \\ & \boxed{\text{B}} \end{bmatrix}, \quad A \oplus^k B = \begin{bmatrix} \boxed{\text{A}} & \\ & \boxed{\text{B}} \end{bmatrix},$$

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

Overlapped Tensor Product. We define the *column overlapped tensor product* through the column overlapped direct sum:

$$\mathrm{I}_s \otimes^k A = \underbrace{A \oplus^k A \oplus^k \cdots \oplus^k A}_{s\text{-fold}}, \quad \text{or} \quad (10)$$

$$\mathrm{I}_s \otimes^k A = \begin{bmatrix} \boxed{\text{A}} & & & \\ & \boxed{\text{A}} & & \\ & & \boxed{\text{A}} & \\ & & & \ddots \\ & & & \boxed{\text{A}} \end{bmatrix}, \quad (11)$$

The *row overlapped* tensor product $\mathrm{I}_s \otimes_k A$ is defined analogously.

4. BREAKDOWN RULES AND ALGORITHMS

Using the constructs defined in Section 3, we represent the filter (5) in the concise form

$$F_n(\mathbf{h}) = \mathrm{I}_n \otimes^{k-1} \mathbf{h}^{\mathrm{T}}, \quad (12)$$

where k is the filter length. Interpreted as an algorithm, (12) computes $F_n(\mathbf{h})$ column by column, using $k-1$ additional additions (the column overlap) in each step. This algorithm is a special case of the overlap-add (OA) rule [5], which in our notation becomes

$$F_n(\mathbf{h}) = \mathrm{I}_{n/b} \otimes^{k-1} F_b(\mathbf{h}), \quad b|n. \quad (13)$$

The OA rule divides a filter transform of size n into smaller filter transforms of size b that operate on independent segments of the input data (input locality). The dual version of the OA rule is the overlap-save (OS) rule [5], which computes the filter transform as

$$F_n(\mathbf{h}) = T(\mathbf{h}_L) \oplus_{k-1} (\mathrm{I}_{m/b} \otimes_{k-1} F_b^T(\mathbf{h})) \oplus_{k-1} T(\mathbf{h}_R), \quad (14)$$

where $m = n - k + 1$. The Toeplitz matrices $T(\mathbf{h}_L)$ and $T(\mathbf{h}_R)$ represent, respectively, the upper left and lower right $k \times k$ triangle in (5). The remaining middle part is a transposed filter that is again computed in blocks that produce independent segments of the output vector (output locality).

As a compromise between the input and the output locality of the previous two rules, we introduce the *blocking rule*, used in [6] for arbitrary sparse matrices. It divides the filter matrix (5) into square blocks with Toeplitz structure:

$$F_n^T(\mathbf{h}) = \mathrm{I}_{n/b} \otimes_v \bigoplus_{i=1}^{\lceil k/b \rceil}{}^{b} T(\mathbf{h}_i) \quad (15)$$

where b is the square block size, $v = (\lceil k/b \rceil - 1)b$ is the overlap, and $\mathbf{h}_i$ are segments of $\mathbf{h}$ overlapped on $b-1$ points. A similar rule can be applied to the occurring Toeplitz matrices, thus enabling multiple levels of blocking.

All of the above rules decompose the filter transform in the time domain and lead to algorithms with the same arithmetic cost as a computation by definition (4). The difference is in the order of computation, which has a significant impact on runtimes as we show later.

It is well known that filtering can be performed in the frequency domain using the DFT, based on the fact that the DFT diagonalizes the circulant matrix (7). To apply this algorithm to a filter, we first extend (5) to a circulant matrix (7) by appending $k - 1$ columns. Formally,

$$F_n(\mathbf{h}) = C\left(E_{k,n-1} \cdot \mathbf{h}\right) \cdot E_{n,k-1}, \quad (16)$$

where $E_{n,k-1}$ is a suitable padding matrix of size $n+k-1 \times n+k-1$. Then, the circulant matrix is decomposed using the DFT. For real valued input data, the complexity of the standard fast DFT algorithms can be reduced using the symmetry properties of the DFT. There are several approaches in this direction. We use the discrete Hartley transform (DHT) to decompose a circulant matrix as

$$C(\mathbf{a}) = \mathrm{DHT}_n^{-1} \cdot \mathrm{X}(\mathrm{DHT}_n \cdot \mathbf{a}) \cdot \mathrm{DHT}_n\,. \quad (17)$$

Here, $\mathrm{X}(\mathrm{DHT}_n \cdot \mathbf{a})$ represents an X-shaped matrix with the real and imaginary parts of the vector $\mathrm{DHT}_n \cdot \mathbf{a}$ arranged on the diagonal and the opposite lower diagonal. For the DHT, we use a radix-2 rule [7].

5. EXPERIMENTAL RESULTS

We included the breakdown rules (13)–(17) into SPIRAL to generate and evaluate a large number of different filter transform algorithms. In addition to this algorithmic degree of freedom, we also included the degree of loop unrolling in the search space.

We ran experiments on two common computer architectures: an Intel Pentium 4 (2.53 GHz, running Linux) using gcc 2.95, and a SUN UltraSPARC II (450MHz) using the SUN Workshop 6 compiler. All algorithms were automatically generated, implemented in C code, and benchmarked by SPIRAL.

As the baseline experiment, we use the algorithm built with only one application of the rule (12): the OA rule with block size $b = 1$, implemented as one loop with the loop body corresponding to the columns in (5). We choose this algorithm as the baseline since it leads to a straightforward implementation, most likely to be chosen by a human programmer. This algorithm has maximal input locality; each input sample is loaded only once.

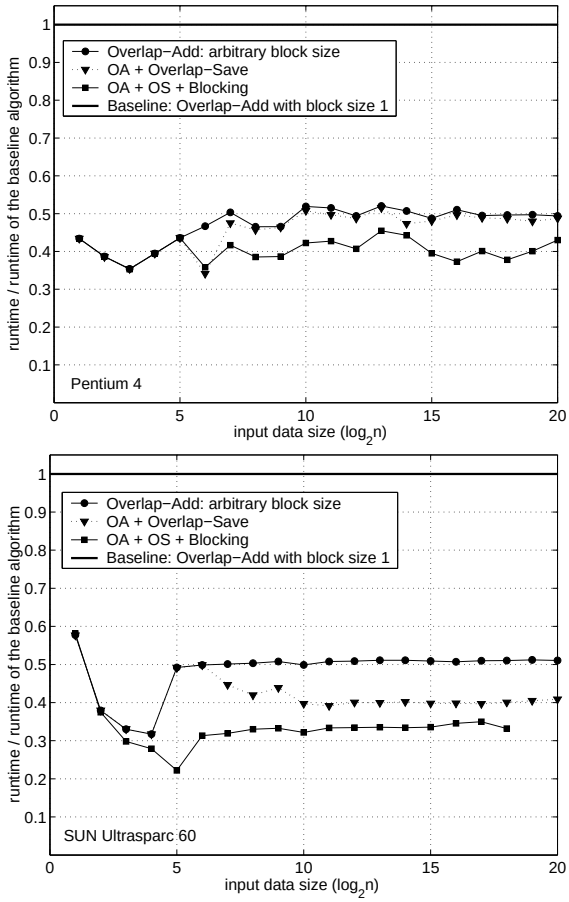

Fig. 2. Runtime comparison (lower = better) of different time-domain algorithms on Pentium 4 (above) and SUN (below).

We note that the runtime of a filter transform does not depend on the actual values of the filter taps as long as they are nonzero. In each experiment, we conduct a dynamic programming search (provided by SPIRAL) using the set of rules considered.

Time-domain methods. In the first experiment, we compare the different time-domain algorithms relative to the base method for a filter length of $k = 33$ (see Fig. 2). We start with the general overlap-add rule (13) with arbitrary block size b. The smaller filter in (13) is computed row-wise using unrolled code. SPIRAL always finds the maximal possible block size (limited by the code size the C compiler could process), improving the runtime by about 50% relative to the base method (Fig. 2, circles). This result indicates that output locality leads to faster code than input locality; indeed, by including the OS rule (14) in the search, the OA rule is not used (i.e., found) anymore, and we obtain further runtime improvement (Fig. 2, triangles). Finally, by adding the

blocking rule (15), we obtain the best time-domain implementations on both platforms, up to 75% better than the base method (Fig. 2, squares). The block size typically found is 2 or 4, allowing for in-register computation. We remind the reader that each of the time-domain algorithms has precisely the same arithmetic cost (total number of additions and multiplications), namely $n \cdot k$ multiplications and $n \cdot (k-1)$ additions. Also worth noting is the quantitative difference in the results on Pentium and SUN.

Frequency-domain methods. In the second experiment, we include the frequency-domain rules (16) and (17) to expand the search space. Fig. 3 shows the results on Pentium for filter input sizes $n = 2^1, \ldots, 2^{18}$ and different filter lengths 64, 128, and 256. The base line this time is the best time-domain implementation found before. For filter length 64 (and smaller, not shown), the frequency-domain rules is not found in the search. For sizes 128 and 256 it improves runtimes by about 15% and 30%, starting with input size 2^7 and 2^8, respectively. These results are surprising; a comparison of time-domain and frequency-domain algorithms based on arithmetic cost suggests that the latter are superior also for smaller filter lengths.

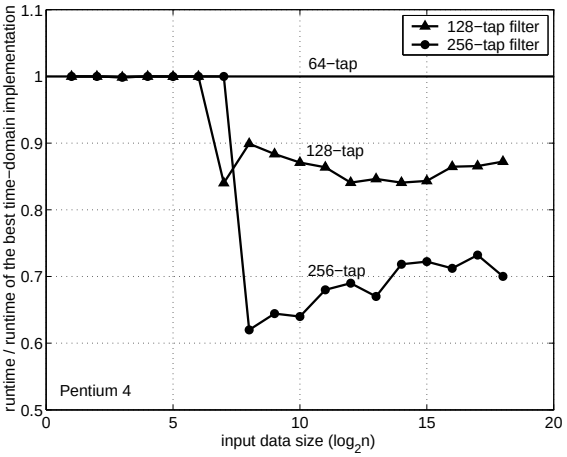

Fig. 3. Runtime improvement using frequency-domain algorithms.

To investigate this discrepancy, we conduct a final experiment with circulant matrices, comparing time-domain and frequency-domain computation, shown in Fig. 4. First, we consider dense circulant matrices of sizes $n = 2^2, \ldots, 2^7$ (Fig. 4, triangles). We display the quotient between the arithmetic cost for the DHT-based method (using radix-2 given by $5n\log_2(n) - 13n/2 + 8$) and for direct computation (given by $2n^2 - n$), shown as a dotted line. The result suggests that the DHT-based method is always preferable. Comparing the actual runtimes, shown as solid line, however, shows that the time-domain computation is faster up to size 32. Since the circulant matrices constructed from rule (16) have maximally half of their entries nonzero, we conduct the same experiment for these "half-dense" circulant matrices (Fig. 4, circles). The cross-over point between time- and frequency-domain algorithms is, as expected, shifted to the right; around 16 for the arithmetic cost, and beyond 64 for the actual runtimes, consistent with Fig. 3.

Conclusion. We have shown that finding the best software implementation of an FIR filter is not a straightforward task. Using a

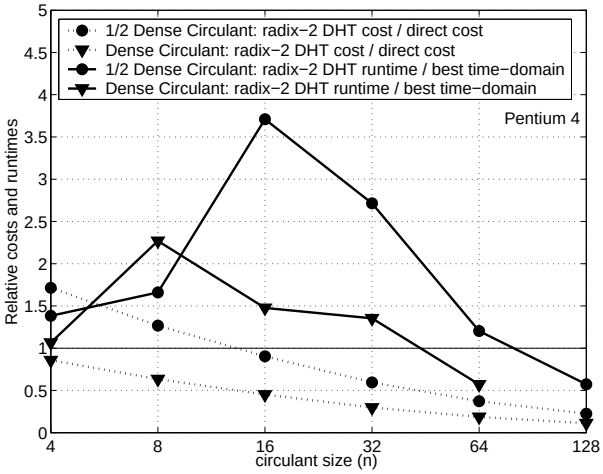

Fig. 4. Comparison of time and frequency based algorithms for a dense and a half-dense circulant matrix.

combination of flexible blocking techniques and search, provided by SPIRAL, a straightforward single-loop implementation can be improved by up to 70% without reducing the arithmetic cost. Further, frequency-domain methods are superior to time-domain algorithms for larger filter lengths, but the cross-over point is not exclusively determined by the arithmetic cost.

6. 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.ece.cmu.edu/∼spiral.

[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," *Journal of High Performance Computing and Applications*, 2003, to appear.

[3] F. Franchetti, M. Püschel, J. M. F. Moura, and C. W. Ueberhuber, "Short Vector SIMD Code Generation for DSP Algorithms," in *Proc. High Performance Embedded Computing (HPEC)*, MIT Lincoln Labs, 2002.

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

[5] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 2nd edition, 1999.

[6] E.-J. Im and K. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *Proc. ICCS*, 2001, pp. 127–136, http://www.cs.berkeley.edu/ yelick/sparsity/.

[7] P. Duhamel and M. Vetterli, "Improved Fourier and Hartley Transform Algorithms: Application to Cyclic Convolution of Real Data," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, no. 6, pp. 818–824, June 1987.