GENERATION AND MANIPULATION OF DSP TRANSFORM ALGORITHMS

Markus Püschel and José M. F. Moura

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, USA

ABSTRACT

SPIRAL generates automatically high quality implementations for a variety of DSP signal transforms. The generated code is adapted to the machine configuration and architecture. SPIRAL achieves this through an intelligent search in the Cartesian product of the space of algorithms and the space of coding variants. This paper describes the mathematical framework that SPIRAL uses to represent, generate, and manipulate transform algorithms.

1. INTRODUCTION

Many different linear discrete signal transforms are used in digital signal processing. Important examples include the discrete Fourier transform (DFT), the discrete cosine transforms (DCTs), and the discrete wavelet transform (DWTs). Fast algorithms for these transforms typically are highly structured and recursive, i.e., break down a transform of one size into, possibly different, transforms of smaller sizes. Combining these recursions in all possible ways leads to a very large (usually exponentially) number of different fast algorithms for one given transform. These algorithms are equivalent in arithmetic cost (number of additions and multiplications), but differ in data flow, which, on modern architectures, leads to a large spread in runtime performance. Since the runtime depends to a large extent on architectural features (e.g., number of registers, size and structure of caches), the best algorithm for a DSP transform depends on the given computing platform (e.g., Pentium III, Pentium 4, Athlon XP). Furthermore, given the complexity of DSP algorithms and the complexity of modern architectures, it is very hard find the fastest algorithm for a given platform. The question is: *How to provide portable high performance for DSP algorithms across platforms?*

One approach to this problem is to provide a library that has flexibility in how to compute a transform, like FFTW [1] for the DFT, and to use search to find the best choice.

SPIRAL's [2, 3] approach addresses the entire domain of (linear) signal transforms and provides a *generator* for implementations that are adapted to the given target computing platform. SPIRAL represents fast transform algorithms as *formulas* in a mathematical language and translates the implementation problem into a search in the space of these formulas and the possible implementation choices. The architecture of SPIRAL is displayed in Fig. 1. The user specifies a transform to implement, e.g., a DFT of size 1024. The **Formula Generator** module recursively expands the DFT into a formula representing a fast algorithm. The **Formula Translator** translates the formula a program (currently C, Fortran, or C enhanced with vector instructions). The runtime of this generated program is fed back through the **Search Engine**, which controls the generation of the next formula and possible implementation options (e.g., degree of loop unrolling) using intelligent search techniques such as dynamic programming or evolutionary algorithms. Iteration of this loop yields an implementation of the DFT_{1024} that is optimized to the given platform. SPIRAL's approach optimizes, akin to a human expert programmer, simultaneously in the mathematical domain of algorithm and the implementation domain. The code generated by SPIRAL is highly competitive with hand-written programs [4]. For the DFT, SPIRAL generated code recently outperformed Intel's Math Kernel Library [5, 6].

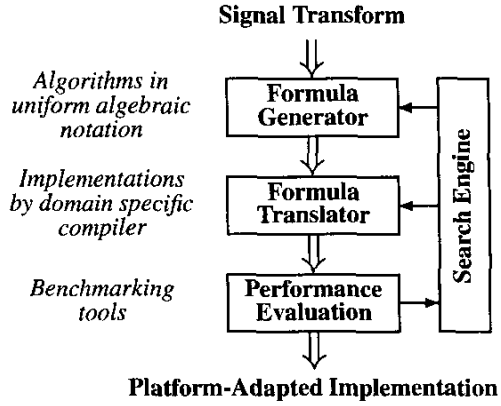

Fig. 1. The architecture of SPIRAL.

In this paper we explain the mathematical framework that SPIRAL uses to represent, generate, and manipulate fast transform algorithms.

In Section 2, we introduce the notation we use to represent fast transform algorithms. Section 3 defines the key

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

0-7803-8116-5/02/$17.00 ©2002 IEEE.

concepts of SPIRAL's framework; examples are given in Section 4. The space of different algorithms is discussed in Section 5. We conclude with a short overview on SPIRAL in Section 6.

2. NOTATION

We use the following operators and primitives to represent structured matrices. The *direct sum* $\oplus$ and the *tensor or Kronecker product* $\otimes$ of matrices A, B are defined as

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}, \quad A \otimes B = [a_{k\ell}B], \quad A = [a_{k\ell}].$$

Further we use $\cdot$ for the matrix product, and $A^P = P^{-1}AP$ for matrix conjugation. We use the following symbols for frequently occurring classes of matrices. The $n \times n$ identity matrix is denoted by I_n, a 2×2 rotation with angle α and a basic butterfly matrix by

$$\mathrm{R}_\alpha = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}, \quad F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

respectively. Further, for $n = rs$, we denote by

$$\mathrm{L}^n_r : \; k \mapsto k \cdot r \bmod (n-1), \text{ for } 0 \le k < n-1$$
$$n-1 \mapsto n-1,$$

the stride permutation matrix, and by

$$\mathrm{T}^n_r = \bigoplus_{k=0}^{s-1} \mathrm{diag}(\omega_n^0, \ldots, \omega_n^{r-1})^k, \quad \omega_n = e^{2\pi j/n}.$$

the twiddle diagonal matrix. We reserve the letter P for permutation matrices, D for diagonal matrices, and S for generic sparse matrices, Specific permutation matrices are written as $[\sigma, n]$, where σ is a permutation, and n the matrix size.

3. TRANSFORMS AND ALGORITHMS

Mathematically, a linear discrete signal transform is given by a multiplication

$$x \mapsto Mx, \tag{1}$$

where x is the sampled signal and M is the transform matrix. Fast algorithms for the transform can be represented as a factorization of M into a product of structured sparse matrices,

$$M = M_1 M_2 \cdots M_k, \quad M_i \text{ sparse}.$$

Typically, these factorizations reduce the arithmetic cost of computing the transform from $O(n^2)$, as required by direct matrix-vector multiplication, to $O(n \log n)$. It is a special property of signal transforms that these factorizations exist *and* that the matrices M_i are highly structured. In SPIRAL, we use this structure to efficiently represent and generate these factorizations.

To illustrate SPIRAL's framework we start with a very simple example, the discrete Fourier transform (DFT) of size four, indicated as DFT_4. The DFT_4 can be factorized into a product of four sparse matrices,

$$\mathrm{DFT}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & j \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This factorization represents a fast algorithm for computing the DFT of size four and is an instantiation of the Cooley-Tukey algorithm [7], usually referred to as the fast Fourier transform (FFT). Using the structure of the sparse factors, the factorization is rewritten in the concise form

$$\mathrm{DFT}_4 \;=\; (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}^4_2 \cdot (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}^4_2 \tag{2}$$

using the notation introduced in Section 2.

We now extend this example to detail SPIRAL's framework, which, apart from transforms, includes three important concepts:

- *breakdown rules* decompose transforms into other transforms;
- *ruletrees* are very efficient representations of breakdown strategies, or fast algorithms, and can be converted into
- *formulas*, which are symbolic representations of fast algorithms and can be translated into code.

We define these concepts and illustrate them using the DFT.

Transforms. A *transform* is a parameterized class of matrices denoted by a mnemonic expression, e.g., DFT, with one or several parameters in the subscript. For example, DFT_n represents the matrix

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}, \quad \omega_n = e^{2\pi j/n}. \tag{3}$$

Fixing the parameter (here n) determines an instantiation of the transform, e.g., DFT_8 by fixing $n = 8$. By abuse of notation, we will refer to an instantiation also as transform. By *computing a transform* M, we mean evaluating the matrix-vector product $y = M \cdot x$ in Equation (1).

Rules. A *break-down rule*, or simply *rule*, is an equation that structurally decomposes a transform. The applicability of the rule may depend on the parameters, i.e., the size of the transform. An example for a rule is the Cooley-Tukey FFT for a DFT_n, given by (for $n = rs$)

$$\mathrm{DFT}_n = (\mathrm{DFT}_r \otimes \mathrm{I}_s) \cdot \mathrm{T}^n_s \cdot (\mathrm{I}_r \otimes \mathrm{DFT}_s) \cdot \mathrm{L}^n_r, \tag{4}$$

where the twiddle matrix T^n_s and the stride permutation L^n_r are defined in Section 2. A rule like (4) is called *parameterized*, since it depends on the factorization of the transform

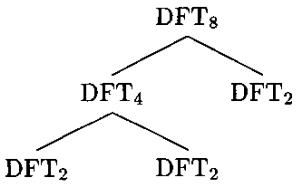

Fig. 2. A fully expanded ruletree for the DFT_8; the rules at the nodes are omitted.

size n. Different factorizations of n give different instantiations of the rule. In the context of SPIRAL, a rule determines a sparse structured matrix factorization of a transform, and breaks down the problem of computing the transform into computing possibly different transforms of usually smaller size (here: DFT_r and DFT_s). We *apply a rule* to a transform of a given size n by replacing the transform by the right hand-side of the rule (for this n). If the rule is parameterized, then an instantiation of the rule is chosen. As an example, applying (4) to DFT_8, using the factorization $8 = 4 \cdot 2$, yields

$$(\mathrm{DFT}_4 \otimes \mathrm{I}_2) \cdot \mathrm{T}^8_2 \cdot (\mathrm{I}_4 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}^8_4 . \tag{5}$$

In SPIRAL's framework, a rule does not yet determine an algorithm. For example, applying the Cooley-Tukey rule (4) once reduces the problem of computing a DFT_n to computing the smaller transforms DFT_r and DFT_s. At this stage it is undetermined how these are computed. By applying rules recursively, we eventually obtain base cases like DFT_2. These are fully expanded by trivial break-down rules, the *base case* rules, that replace the transform by its definition, e.g.,

$$\mathrm{DFT}_2 = \mathrm{F}_2 \tag{6}$$

Note that F_2 is *not* a transform, but a symbol for the matrix.

Ruletrees. Recursive application of rules can naturally be represented by a *ruletree*. Each node in the ruletree contains a transform and the rule applied at this stage. We call a ruletree fully expanded, if all leaves contain base case rules. A fully expanded ruletree represents an algorithm for computing the transform at the root of the tree. An example for a fully expanded ruletree is given in Fig. 2. In the following, we always assume that ruletrees are fully expanded.

Ruletrees are very *efficient representations* of fast algorithms; and allow *easy manipulation*, e.g., by expanding a subtree differently, to generate variations of a given algorithm.

Formulas. Applying a rule to a transform of a given size yields a *formula*. Examples of formulas are (5) and the right-hand side of (2). A formula is a mathematical expression representing a structural decomposition of a matrix. The expression is composed from (see Section 2) mathematical operators like the matrix product $\cdot$, the tensor product $\otimes$, the direct sum $\oplus$, and basic primitives such as arbitrary matrices, diagonal matrices (e.g., $\mathrm{diag}(4,1,3)$), permutation matrices (e.g., $[(1,2),3]$ for the permutation $(1,2)$ on 3 points), symbolically represented matrices (e.g., I_n, L^{rs}_r, T^{rs}_r, F_2, R_α for a 2×2 rotation matrix of angle α), and transforms of fixed size (e.g., DFT_4, $\mathrm{DCT}^{(\mathrm{II})}_8$). An example of a formula for a DCT of size 4 (introduced in Section 4) is

$$[(2,3),4] \cdot (\mathrm{diag}(1, \sqrt{1/2}) \cdot \mathrm{F}_2 \oplus \mathrm{R}_{13\pi/8}) \cdot [(2,3),4] \cdot (\mathrm{I}_2 \otimes \mathrm{F}_2) \cdot [(2,4,3),4].$$

The motivation for considering rules and formulas is to provide a flexible and extensible framework that derives and represents algorithms for transforms. Our notion of algorithms is best explained by expanding the previous example DFT_8. Applying Rule (4) (with $8 = 4 \cdot 2$) once yields the formula in (5). This formula does not determine an algorithm for the DFT_8, since it is not specified how to compute DFT_4 and DFT_2. Expanding DFT_4 using again Rule (4) (with $4 = 2 \cdot 2$) yields

$$\left(\left((\mathrm{DFT}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}^4_2 \cdot (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}^4_2\right) \otimes \mathrm{I}_2\right) \cdot \mathrm{T}^8_2 \cdot (\mathrm{I}_4 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}^8_4 .$$

Finally, by applying the (base case) Rule (6) to expand all occurring DFT_2's we obtain the fully expanded formula

$$\left(\left((\mathrm{F}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}^4_2 \cdot (\mathrm{I}_2 \otimes \mathrm{F}_2) \cdot \mathrm{L}^4_2\right) \otimes \mathrm{I}_2\right) \cdot \mathrm{T}^8_2 \cdot (\mathrm{I}_4 \otimes \mathrm{F}_2) \cdot \mathrm{L}^8_4 . \tag{20}$$

This formula represents a fast algorithm for computing the DFT_8. Thus, in our framework, a formula is called *fully expanded* if it does not contain any transforms. A fully expanded formula corresponds to a fully expanded ruletree and uniquely determines an algorithm for the represented matrix. In other words, the transforms in a formula serve as place-holder that need to be expanded by a rule to fix the way they are computed. As an example, the ruletree in Fig. 2 corresponds to the formula in (20).

In the following section we demonstrate that the presented framework is not restricted to the DFT, but is applicable to a large class of DSP transforms.

4. EXAMPLES OF TRANSFORMS AND RULES

We provide a few examples of transforms and rules that are included in SPIRAL. The DFT_n is defined in (3). Further, there are sixteen types of trigonometric transforms, namely eight types of DCTs and eight types of DSTs. As examples, we have

$$\begin{aligned} \mathrm{DCT}^{(\mathrm{II})}_n &= \left[\cos\left((\ell + 1/2)k\pi/n\right)\right]_{k,\ell}, \\ \mathrm{DCT}^{(\mathrm{IV})}_n &= \left[\cos\left((k + 1/2)(\ell + 1/2)\pi/n\right)\right]_{k,\ell}, \\ \mathrm{DST}^{(\mathrm{II})}_n &= \left[\sin\left((k + 1)(\ell + 1/2)\pi/n\right)\right]_{k,\ell}, \\ \mathrm{DST}^{(\mathrm{IV})}_n &= \left[\sin\left((k + 1/2)(\ell + 1/2)\pi/n\right)\right]_{k,\ell}, \end{aligned} \tag{21}$$

$$\mathrm{DFT}_n = \mathrm{CosDFT}_n + j \cdot \mathrm{SinDFT}_n \tag{7}$$

$$\mathrm{DFT}_n = (\mathrm{I}_2 \oplus (\mathrm{I}_{n/2-1} \otimes \mathrm{F}_2 \cdot D_2))^{P_n} \cdot (\mathrm{DCT}^{(\mathrm{I})}_{n/2+1} \oplus (\mathrm{DST}^{(\mathrm{I})}_{n/2-1})^{P'_{n/2-1}}) \cdot (\mathrm{I}_2 \oplus (\mathrm{I}_{n/2-1} \otimes \mathrm{F}_2))^{P''_n}, \quad 2 \mid n \tag{8}$$

$$\mathrm{CosDFT}_n = S_n \cdot (\mathrm{CosDFT}_{n/2} \oplus \mathrm{DCT}^{(\mathrm{II})}_{n/4}) \cdot S'_n \cdot \mathrm{L}^n_2, \quad 4 \mid n \tag{9}$$

$$\mathrm{SinDFT}_n = S_n \cdot (\mathrm{SinDFT}_{n/2} \oplus \mathrm{DCT}^{(\mathrm{II})}_{n/4}) \cdot S'_n \cdot \mathrm{L}^n_2, \quad 4 \mid n \tag{10}$$

$$\mathrm{DCT}^{(\mathrm{II})}_2 = \mathrm{diag}(1, 1/\sqrt{2}) \cdot \mathrm{F}_2 \tag{11}$$

$$\mathrm{DCT}^{(\mathrm{II})}_n = P_n \cdot (\mathrm{DCT}^{(\mathrm{II})}_{n/2} \oplus (\mathrm{DCT}^{(\mathrm{IV})}_{n/2})^{P'_n}) \cdot (\mathrm{I}_{n/2} \otimes \mathrm{F}_2)^{P''_n}, \quad 2 \mid n \tag{12}$$

$$\mathrm{DCT}^{(\mathrm{IV})}_2 = [(1,2),2] \cdot \mathrm{R}_{13\pi/8} \tag{13}$$

$$\mathrm{DCT}^{(\mathrm{IV})}_n = S_n \cdot \mathrm{DCT}^{(\mathrm{II})}_n \cdot D_n \tag{14}$$

$$\mathrm{DCT}^{(\mathrm{IV})}_n = (\mathrm{I}_1 \oplus (\mathrm{I}_{n/2-1} \otimes \mathrm{F}_2) \oplus \mathrm{I}_1) \cdot P_n \cdot (\mathrm{DCT}^{(\mathrm{II})}_{n/2} \oplus (\mathrm{DST}^{(\mathrm{II})}_{n/2})^{P'_{n/2}}) \cdot (R_1 \oplus \ldots \oplus R_{n/2})^{P''_n}, \quad 2 \mid n \tag{15}$$

$$\mathrm{DCT}^{(\mathrm{IV})}_{2^k} = P_{2^k} \cdot (R_1 \oplus \ldots \oplus R_{2^{k-1}}) \cdot \left(\prod_{j=k-1}^{1} \left(\mathrm{I}_{2^{k-j-1}} \otimes (\mathrm{F}_2 \otimes \mathrm{I}_{2^j}) \cdot \left(\mathrm{I}_{2^j} \oplus R^{(j)}_1 \oplus \ldots \oplus R^{(j)}_{2^j-1} \right) \right) \right) \cdot P'_{2^k} \tag{16}$$

$$\mathrm{MDCT}_n = \mathrm{DCT}^{(\mathrm{IV})}_n \cdot S_{n \times 2n} \tag{17}$$

$$\mathrm{WHT}_{2^k} = \prod_{j=1}^{t} \left(\mathrm{I}_{2^{k_1+\cdots+k_{j-1}}} \otimes \mathrm{WHT}_{2^{k_j}} \otimes \mathrm{I}_{2^{k_{j+1}+\cdots+k_t}} \right), \quad k = k_1 + \cdots + k_t \tag{18}$$

$$\mathrm{RHT}_{2^k} = (\mathrm{RHT}_{2^{k-1}} \oplus \mathrm{I}_{2^{k-1}}) \cdot (\mathrm{F}_2 \otimes \mathrm{I}_{2^{k-1}}) \cdot \mathrm{L}^{2^k}_2, \quad k > 1 \tag{19}$$

Table 1. A subset of the rules included in SPIRAL.

where the superscript indicates in romans the type of the transform, and the index range is $0 \le k, \ell < n$ in all cases. Some of the other DCTs and DSTs relate directly to the ones above; for example,

$$\mathrm{DCT}^{(\mathrm{III})}_n = \left(\mathrm{DCT}^{(\mathrm{II})}_n\right)^T, \quad \text{and } \mathrm{DST}^{(\mathrm{III})}_n = \left(\mathrm{DST}^{(\mathrm{II})}_n\right)^T,$$

where $(\cdot)^T$ denotes matrix transposition. The $\mathrm{DCT}^{(\mathrm{II})}$ and the $\mathrm{DCT}^{(\mathrm{IV})}$ are used in the image and video compression standards JPEG and MPEG, respectively. In MPEG audio compression the MDCT_n (an $n \times 2n$ matrix) is used, defined by

$$\mathrm{MDCT}_n = \left[\cos((2\ell + 1 + n)(2k + 1)/(4n)\right],$$

where $0 \le k < n$, $0 \le \ell < 2n$. The Walsh-Hadamard transform WHT_{2^k} is defined as

$$\mathrm{WHT}_{2^k} = \underbrace{\mathrm{F}_2 \otimes \ldots \otimes \mathrm{F}_2}_{k \text{ fold}}.$$

The (rationalized) Haar transform is recursively defined by

$$\mathrm{RHT}_2 = \mathrm{F}_2, \quad \mathrm{RHT}_{2^{k+1}} = \begin{bmatrix} \mathrm{RHT}_{2^k} \otimes [1 \quad 1] \\ \mathrm{I}_{2^k} \otimes [1 \; -1] \end{bmatrix}, \; k > 1.$$

We also consider the real and the imaginary part of the DFT,

$$\begin{array}{rcl} \mathrm{CosDFT} & = & \mathrm{Re}(\mathrm{DFT}_n), \quad \text{and} \\ \mathrm{SinDFT} & = & \mathrm{Im}(\mathrm{DFT}_n). \end{array} \tag{22}$$

Finally, if M is an arbitrary transform, then its corresponding k-dimensional version, $k \le 2$, is given by a k-fold tensor product $M \otimes \ldots \otimes M$. Thus, multi-dimensional transforms are automatically included in our framework.

We list in Table 1 a subset of the rules considered by SPIRAL for the transforms introduced above. Due to lack of space, we do not give the exact form of every matrix appearing in the rules, but simply indicate their type using the notation introduced in Section 2. The symbols P, D, R, S refer to permutation, diagonal, rotation, and other sparse matrices, respectively. The same symbols can have different meanings in different rule The exact form of the occurring matrices can be found in [8, 9, 10].

We note the following important facts.

- *Base case rules* expand transforms of trivial size (e.g., Rules (11) and (13)).
- *Recursive rules* expand a transform in terms of similar (e.g., Rules (4) and (18)) or different (e.g., Rules (12) and (8)) transforms of smaller size.
- *Transformation rules* expand a transform in terms of different transforms of the same size (e.g., Rules (7) and (14)).
- *Iterative rules* completely expand a transform in one step (e.g., Rule (16)).
- *Parameterized rules* have different instantiations (e.g., Rule (4) depends on the factorization $n = r \cdot s$, which in

k	DFT, size 2^k	$\mathrm{DCT}^{(\mathrm{IV})}$, size 2^k
1	1	1
2	7	8
3	48	86
4	434	15,778
5	171016	$\sim 5.0 \times 10^8$
6	$\sim 3.4 \times 10^{12}$	$\sim 5.3 \times 10^{17}$
7	$\sim 3.7 \times 10^{28}$	$\sim 5.6 \times 10^{35}$
8	$\sim 2.1 \times 10^{62}$	$\sim 6.2 \times 10^{71}$
9	$\sim 6.8 \times 10^{131}$	$\sim 6.8 \times 10^{143}$

Table 2. Number of formulas for DFT and $\mathrm{DCT}^{(\mathrm{IV})}$ of size 2^k, $0 \leq k \leq 9$.

general is not unique).

5. ALGORITHM SPACE

For a given transform there is freedom in how to expand, i.e., which rule to apply. This freedom may arise from the applicability of different rules or from the applicability of one rule that has different instantiations. Recursively applied, this degree of freedom leads to a combinatorial explosion of the number of formulas, or algorithms, for a given transform. As examples, Table 2 shows the surprisingly large number of different algorithms arising from the rules considered by SPIRAL, for the DFT and the $\mathrm{DCT}^{(\mathrm{IV})}$ of small 2-power sizes.

The set of all algorithms for a given transform constitutes the *algorithm space* that SPIRAL searches when generating an efficient implementation on a given platform. The numbers in Table 2 show that, even for modest size, an exhaustive search in this space is not feasible.

It is important to note that the numerous fully expanded formulas, i.e., algorithms, generated for a given transform from a set of rules, have (almost) the same arithmetic cost (i.e., the number of additions and multiplications required by the algorithm). They differ in the computational data flow, which leads to a large spread in the runtimes of corresponding implementations, even for very small transform sizes. As an example, Fig. 3 shows a histogram of runtimes for all 15,778 algorithms for a $\mathrm{DCT}_{16}^{(\mathrm{IV})}$ implemented by SPIRAL in straight-line code (i.e., without using loops). The range of runtimes is more than a factor of 3, between 211 and 741 nanoseconds. Furthermore, the fastest algorithms are rare.

6. SPIRAL

The mathematical framework presented in Section 3 provides a clear roadmap on how to implement SPIRAL, a system that automatically searches the algorithm space of a given transform for a fastest implementation on a given platform. At the core of SPIRAL is the representation of an *algorithm* as a (fully expanded) *formula*. This representation connects the mathematical realm of DSP transform algorithms with the realm of their actual C or Fortran implementations. Automation of the implementation process thus requires 1) a computer representation of formulas, which in SPIRAL is achieved by the language SPL [4]; 2) the automatic generation of formulas [11]; and 3) the automatic translation of fully expanded formulas into programs [4, 5]. To generate, in addition, a *very fast* implementation requires 4) a search module that controls the formula generation and possible implementation choices to find the best algorithm for the given computing platform [12].

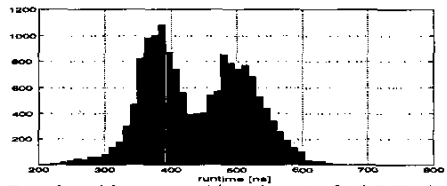

Fig. 3. Runtime histogram (time in ns) of 15,778 algorithms for a $\mathrm{DCT}_{16}^{(\mathrm{IV})}$ on a Pentium 4, 1.4 GHz, running Linux.

Taken together we obtain the architecture of SPIRAL displayed in Fig. 1. In the remainder of this section we very briefly survey the three key modules of SPIRAL: the formula generator, the formula translator, and the search module. For more information we refer to the references given above.

Formula Generator. The task of the formula generator module within SPIRAL (see Fig. 1) is to generate algorithms, given as formulas, for a user specified transform. The internal architecture of the formula generator is displayed in Fig. 4 including the search engine. The dashed boxes indicate databases. A user specified instantiation of a transform is expanded into one or several ruletrees using known rules. The choice of rules is controlled by the search engine. The ruletrees are translated into formulas and exported to the formula translator, which compiles them into C or Fortran programs. The runtime of the generated programs is returned to the search engine, which uses it to control the generation of the next set of ruletrees. The formula generator, and hence SPIRAL, can easily be extended to include new transforms or rules.

Formula Translator. The task of the formula translator is to translate a formula into an implementation (currently C, Fortran, or short vector SIMD code). The translation uses the fact that every formula construct has a natural interpretation as a program. We give a few examples. Diagonals are translated into multiplications that scale the input vector. Permutations are interpreted as readdressing. Products $A \cdot B$ are a sequential execution of first B, then A. Constructs of the form $\mathrm{I}_n \otimes A$ or $A \otimes \mathrm{I}_n$ naturally lead to loop code. The construct $A \otimes \mathrm{I}_n$ naturally leads to vector code.

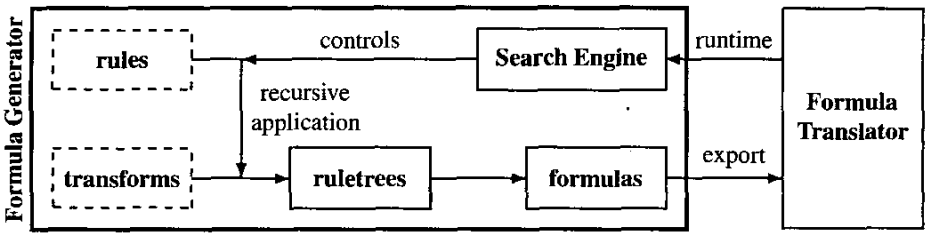

Fig. 4. Internal architecture of the formula generator including the search module. The main components are data types for representing ruletrees and formulas, and extensible databases (dashed boxes) for rules and transforms.

Search Engine. The task of the search engine is to control the formula generation and formula translation to intelligently *search* the space of algorithmic degrees of freedom (different formulas) and implementation options (e.g., different degrees of loop unrolling). Different search strategies are included. We give two examples. A *dynamic programming search* recursively generates a table of best ruletrees for transforms. A given transform is expanded once using all applicable rules; for the smaller transforms, the previously found ruletrees are used. An *evolutionary algorithm* (STEER) first generates a random initial population of ruletrees. Then mutations (e.g., expanding a subtree differently) and cross-breeding (e.g., swapping subtrees) are used to expand this population. The fittest (i.e., fastest) individuals survive to create the next generation. This process is iterated a chosen number of times.

Summary. The concepts of rules, ruletrees, and formulas have been developed to provide SPIRAL with the following properties.

- *Generality:* The capability to combine rules in an arbitrary way allows SPIRAL to explore a very large class of algorithms.
- *Efficiency:* Ruletrees and formulas are a very efficient representation of algorithms: they require little storage, can be generated fast, and can be easily manipulated by the search engine.
- *Extensibility:* Since SPIRAL's formula generation relies only on rules, new transforms can be easily included.

7. REFERENCES

[1] Matteo Frigo and Steven G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. ICASSP*, 1998, vol. 3, pp. 1381–1384, `http://www.fftw.org`.

[2] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, "SPIRAL," `http://www.ece.cmu.edu/~spiral`.

[3] M. Püschel, B. Singer, J. Xiong, J. M. F Moura, J. Johnson, D. Padua, M. M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *J. High Perf. Computing and Appl.*, submitted.

[4] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A Language and Compiler for DSP Algorithms," in *Proc. PLDI*, 2001, pp. 298–308.

[5] F. Franchetti and M. Püschel, "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," in *Proc. IPDPS*, 2002.

[6] F. Franchetti, M. Püschel, J. M. F. Moura, and C. Überhuber, "Short Vector SIMD Code Generation for DSP Algorithms," in *Proc. HPEC*, 2002, to appear.

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

[8] Z. Wang, " Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform ," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-32, no. 4, pp. 803–816, 1984.

[9] M. Vetterli and H.J. Nussbaumer, "Simple FFT and DCT Algorithms with reduced Number of Operations," *Signal Processing*, vol. 6, pp. 267–278, 1984.

[10] D. F. Elliott and K. R. Rao, *Fast Transforms: Algorithms, Analyses, Applications*, Academic Press, 1982.

[11] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura, "Fast Automatic Generation of DSP Algorithms," in *Proc. ICCS 2001*. 2001, pp. 97–106, Springer.

[12] B. Singer and M. Veloso, "Stochastic Search for Signal Processing Algorithm Optimization," in *Proc. Supercomputing*, 2001.