SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms

Markus Püschel[*] Bryan Singer Jianxin Xiong José M. F. Moura
Jeremy Johnson David Padua Manuela Veloso Robert W. Johnson

Abstract

SPIRAL is a generator of libraries of fast software implementations of linear signal processing transforms. These libraries are adapted to the computing platform and can be re-optimized as the hardware is upgraded or replaced. This paper describes the main components of SPIRAL: the mathematical framework that concisely describes signal transforms and their fast algorithms; the formula generator that captures at the algorithmic level the degrees of freedom in expressing a particular signal processing transform; the formula translator that encapsulates the compilation degrees of freedom when translating a specific algorithm into an actual code implementation; and, finally, an intelligent search engine that finds within the large space of alternative formulas and implementations the "best" match to the given computing platform. We present empirical data that demonstrates the high performance of SPIRAL generated code.

1 Introduction

The short life cycles of modern computer platforms are a major problem for developers of high performance software for numerical computations. The different platforms are usually source code compatible (i.e., a suitably written C program can be recompiled) or even binary compatible (e.g., if based on Intel's x86 architecture), but the fastest implementation is platform-specific due to differences in, for example, microarchitectures or cache sizes and structures. Thus, producing optimal code requires skilled programmers with intimate knowledge in both the algorithms and the intricacies of the target platform. When the computing platform is replaced, hand-tuned code becomes obsolete.

The need to overcome this problem has led in recent years to a number of research activities that are collectively referred to as "automatic performance tuning." These efforts target areas with high performance requirements such as very large data sets or real time processing.

One focus of research has been the area of linear algebra leading to highly efficient automatically tuned software for various algorithms. Examples include ATLAS [36], PHiPAC [2], and SPARSITY [17].

Another area with high performance demands is digital signal processing (DSP), which is at the heart of modern telecommunications and is an integral component of different multi-media technologies, such as image/audio/video compression and water marking, or in medical imaging like computed image tomography and magnetic resonance imaging, just to cite a few examples. The computationally most intensive tasks in these technologies are performed by discrete signal transforms. Examples include the discrete Fourier transform (DFT), the discrete cosine transforms (DCTs), the Walsh-Hadamard transform (WHT) and the discrete wavelet transform (DWT).

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

The research on adaptable software for these transforms has to date been comparatively scarce, except for the efficient DFT package FFTW [11, 10]. FFTW includes code modules, called “codelets,” for small transform sizes, and a flexible breakdown strategy, called a “plan,” for larger transform sizes. The codelets are distributed as a part of the package. They are automatically generated and optimized to perform well on *every platform*, i.e., they are not platform-specific. Platform-adaptation arises from the choice of plan, i.e., how a DFT of large size is recursively reduced to smaller sizes. FFTW has been used by other groups to test different optimization techniques, such as loop interleaving [13], and the use of short vector instructions [7]; UHFFT [21] uses an approach similar to FFTW and includes search on the codelet level and additional recursion methods.

SPIRAL is a generator for platform-adapted libraries of DSP transforms, i.e., it includes no code for the computation of transforms prior to installation time. The users trigger the code generation process after installation by specifying the transforms to implement. In this paper we describe the main components of SPIRAL: the mathematical framework to capture transforms and their algorithms, the formula generator, the formula translator, and the search engine.

SPIRAL’s design is based on the following realization:

- DSP transforms have a *very large* number of different *fast* algorithms (the term “fast” refers to the operations count).
- Fast algorithms for DSP transforms can be represented as *formulas* in a concise mathematical notation using a small number of mathematical constructs and primitives.
- In this representation, the different DSP transform algorithms can be automatically generated.
- The automatically generated algorithms can be *automatically translated* into a high-level language (like C or Fortran) program.

Based on these facts, SPIRAL translates the task of finding hardware adapted implementations into an intelligent search in the space of possible fast algorithms and their implementations.

The main difference to other approaches, in particular to FFTW, is the concise mathematical representation that makes the high-level structural information of an algorithm accessible within the system. This representation, and its implementation within the SPIRAL system, enables the automatic generation of the algorithm space, the high-level manipulation of algorithms to apply various search methods for optimization, the systematic evaluation of coding alternatives, and the extension of SPIRAL to different transforms and algorithms. The details will be provided in Sections 2–4.

The architecture of SPIRAL is displayed in Figure 1. Users specify the transform they want to implement and its size, e.g., a DFT (discrete Fourier transform) of size 1024. The **Formula Generator** generates one, or several, out of many possible fast algorithms for the transform. These algorithms are represented as programs written in a SPIRAL proprietary language—the signal processing language (SPL). The SPL program is compiled by the **Formula Translator** into a program in a common language such as C or Fortran. Directives supplied to the formula translator control implementation choices such as the degree of unrolling, or complex versus real arithmetic. Based on the runtime of the generated program, the **Search Engine** triggers the generation of additional algorithms and their implementations using possibly different directives. Iteration of this process leads to a C or Fortran implementation that is adapted to the given computing platform. Optionally, the generated code is verified for correctness. SPIRAL is maintained at [22].

Reference [19] first proposed, for the domain of DFT algorithms, to use formula manipulation to study various ways of optimizing their implementation for a specific platform. Other research on adaptable packages for the DFT includes [1, 4, 16, 28], and for the WHT includes [20]. The use of dynamic data layout techniques to improve performance of the DFT and the WHT has been studied in the context of SPIRAL in [24, 25].

This paper is organized as follows. In Section 2 we present the mathematical framework that SPIRAL uses to capture signal transforms and their fast algorithms. This framework constitutes the foundation for

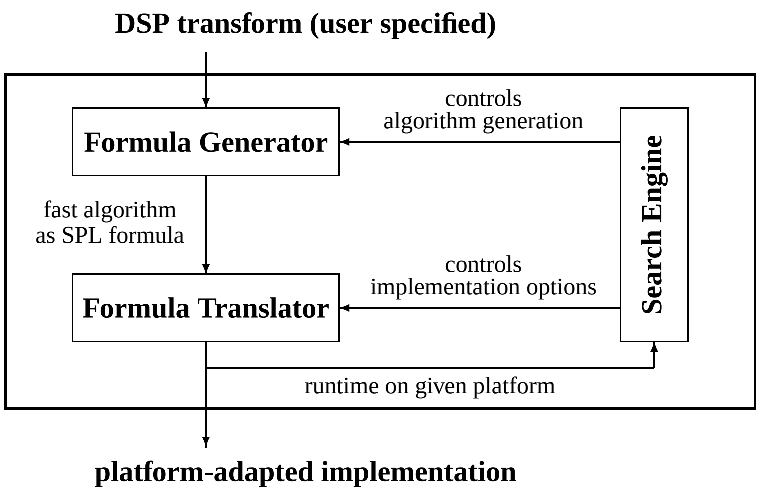


Figure 1: The architecture of SPIRAL.

SPIRAL's architecture. The following three sections explain the three main components of SPIRAL, the formula generator (Section 3), the formula translator (Section 4), and the search engine (Section 5). Section 6 presents empirical runtime results for the code generated by SPIRAL. For most transforms, highly tuned code is not readily available as benchmark. An exception is the DFT for which we compared SPIRAL generated code with FFTW, one of the fastest FFT packages available.

2 SPIRAL's Framework

SPIRAL captures linear discrete signal transforms (also called DSP transforms) and their fast algorithms in a concise mathematical framework. The transforms are expressed as a matrix-vector product

$$y = M \cdot x, \tag{1}$$

where x is a vector of n data points, M is an $n \times n$ matrix representing the transform, and y is the transformed vector.

Fast algorithms for signal transforms arise from factorizations of the transform matrix M into a product of sparse matrices,

$$M = M_1 \cdot M_2 \cdots M_t, \quad M_i \text{ sparse.} \tag{2}$$

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 write these factorizations in a very concise form.

We illustrate SPIRAL's framework with a 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,

$$\mathrm{DFT}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

This factorization represents a fast algorithm for computing the DFT of size four and is an instantiation of the Cooley-Tukey algorithm [3], usually referred to as the fast Fourier transform (FFT). Using the structure of the sparse factors, (3) is rewritten in the concise form

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

where we used the following notation. The tensor (or Kronecker) product of matrices is defined by

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

The symbols $\mathrm{I}_n, \mathrm{L}_r^{rs}, \mathrm{T}_r^{rs}$ represent, respectively, the $n \times n$ identity matrix, the $rs \times rs$ stride permutation matrix that maps the vector element indices j as

$$\mathrm{L}_r^{rs} : \; j \mapsto j \cdot r \bmod rs - 1, \text{ for } j = 0, \ldots, rs - 2; \quad rs - 1 \mapsto rs - 1, \tag{5}$$

and the diagonal matrix of twiddle factors ($n = rs$),

$$\mathrm{T}_r^{rs} = \bigoplus_{j=0}^{s-1} \mathrm{diag}(\omega_n^0, \ldots, \omega_n^{r-1})^j, \quad \omega_n = e^{2\pi i/n}, \quad i = \sqrt{-1}, \tag{6}$$

where

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

denotes the direct sum of A and B. Finally,

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

is the DFT of size 2.

A good introduction to the matrix framework of FFT algorithms is provided in [32, 31]. SPIRAL extends this framework 1) to capture the entire class of linear DSP transforms and their fast algorithms; and 2) to provide the formalism necessary to automatically generate these fast algorithms. We now extend the simple example above and explain SPIRAL's mathematical framework in detail. In Section 2.1 we define the concepts that SPIRAL uses to capture transforms and their fast algorithms. Section 2.2 introduces a number of different transforms considered by SPIRAL. Section 2.3 discusses the space of different algorithms for a given transform. Finally, Section 2.4 explains how SPIRAL's architecture (see Figure 1) is derived from the presented framework.

2.1 Transforms, Rules, and Formulas

In this section we explain how DSP transforms and their fast algorithms are captured by SPIRAL. At the heart of our framework are the concepts of *rules* and *formulas*. In short, rules are used to expand a given transform into formulas, which represent algorithms for this transform. We will now 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, e.g., DFT_n, which stands for the matrix

$$\mathrm{DFT}_n = [e^{2\pi ik\ell/n}]_{k,\ell=0,\ldots,n-1}, \quad i = \sqrt{-1}. \tag{7}$$

Throughout this paper, the only parameter will be the size n of the transform. Sometimes we drop the subscript when referring to the transform. Fixing the parameter 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 a 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 rule is the Cooley-Tukey FFT for a DFT_n, given by

$$\mathrm{DFT}_n = (\mathrm{DFT}_r \otimes \mathrm{I}_s) \cdot \mathrm{T}^n_s \cdot (\mathrm{I}_r \otimes \mathrm{DFT}_s) \cdot \mathrm{L}^n_r, \quad \text{for } n = r \cdot s, \tag{8}$$

where the twiddle matrix T^n_s and the stride permutation L^n_r are defined in (6) and (5). A rule like (8) is called *parameterized,* since it depends on the factorization of the transform 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, an instantiation of the rule is chosen. As an example, applying (8) to DFT_8, using the factorization $8 = 4 \cdot 2$, yields

$$(\mathrm{DFT}_4 \otimes \mathrm{I}_2) \cdot \mathrm{T}^8_2 \cdot (\mathrm{I}_4 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}^8_4 \, . \tag{9}$$

In SPIRAL's framework, a breakdown-rule does not yet determine an algorithm. For example, applying the Cooley-Tukey rule (8) 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 recursively applying rules 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.,

$$\mathrm{DFT}_2 = \mathrm{F}_2, \quad \text{where } \mathrm{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{10}$$

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

Formulas. Applying a rule to a transform of given size yields a *formula*. Examples of formulas are (9) and the right-hand side of (4). A formula is a mathematical expression representing a structural decomposition of a matrix. The expression is composed from the following:

- *mathematical operators* like the matrix product $\cdot$, the tensor product $\otimes$, the direct sum $\oplus$;
- *transforms of a fixed size* such as DFT_4, $\mathrm{DCT}^{(\mathrm{II})}_8$;
- *symbolically represented matrices* like I_n, L^{rs}_r, T^{rs}_r, F_2, or R_α for a 2×2 rotation matrix of angle α:

$$\mathrm{R}_\alpha = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix};$$

- *basic primitives* such as arbitrary matrices, diagonal matrices, or permutation matrices.

On the latter we note that we represent an $n \times n$ permutation matrix in the form $[\pi, n]$, where σ is the defining permutation in cycle notation. For example, $\sigma = (2, 4, 3)$ signifies the mapping of indices $2 \to 4 \to 3 \to 2$, and

$$[(2,4,3),4] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

An example of a formula for a DCT of size 4 (introduced in Section 2.2) is

$$[(2,3),4] \cdot (\mathrm{diag}(1, \sqrt{1/2}) \cdot \mathrm{F}_2 \oplus \mathrm{R}_{13\pi/8}) \cdot [(2,3),4] \cdot (\mathrm{I}_2 \otimes \mathrm{F}_2) \cdot [(2,4,3),4]. \tag{11}$$

Algorithms. 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 (8) (with $8 = 4 \cdot 2$) once yields Formula (9). 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 (8) (with $4 = 2 \cdot 2$) yields

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

Finally, by applying the (base case) Rule (10) to expand all occurring DFT_2's we obtain the formula

$$\Big(\big((\mathrm{F}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}^4_2 \cdot (\mathrm{I}_2 \otimes \mathrm{F}_2) \cdot \mathrm{L}^4_2\big) \otimes \mathrm{I}_2\Big) \cdot \mathrm{T}^8_2 \cdot (\mathrm{I}_4 \otimes \mathrm{F}_2) \cdot \mathrm{L}^8_4, \qquad (12)$$

which does not contain any transforms. In our framework, we call such a formula *fully expanded*. A fully expanded formula uniquely determines an algorithm for the represented transform:

$$\text{fully expanded formula} \leftrightarrow \text{algorithm}.$$

In other words, the transforms in a formula serve as place-holder that need to be expanded by a rule to specify the way they are computed.

Our framework can be restated in terms of formal languages [27]. We can define a grammar by taking transforms as (parameterized) nonterminal symbols, all other constructs in formulas as terminal symbols, and an appropriate set of rules as productions. The language generated by this grammar consists exactly of all fully expanded formulas, i.e., algorithms for transforms.

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.

2.2 Examples of Transforms and their Rules

SPIRAL considers a broad class of DSP transforms and associated rules. Examples include the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), the discrete cosine and sine transforms (DCTs and DSTs), the Haar transform, and the discrete wavelet transform.

We provide a few examples. The DFT_n, the workhorse in DSP, is defined in (7). The Walsh-Hadamard transform WHT_{2^k} is defined as

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

There are sixteen types of trigonometric transforms, namely eight types of DCTs and eight types of DSTs [35]. As examples, we have

$$\begin{aligned} \mathrm{DCT}^{(\mathrm{II})}_n &= \big[\cos\left((\ell + 1/2)k\pi/n\right)\big], \\ \mathrm{DCT}^{(\mathrm{IV})}_n &= \big[\cos\left((k + 1/2)(\ell + 1/2)\pi/n\right)\big], \\ \mathrm{DST}^{(\mathrm{II})}_n &= \big[\sin\left((k + 1)(\ell + 1/2)\pi/n\right)\big], \\ \mathrm{DST}^{(\mathrm{IV})}_n &= \big[\sin\left((k + 1/2)(\ell + 1/2)\pi/n\right)\big], \end{aligned} \qquad (13)$$

where the superscript indicates in romans the type of the transform, and the index range is $k, \ell = 0, \ldots, n-1$ in all cases. Some of the other DCTs and DSTs relate directly to the ones above; for example,

$$\mathrm{DCT}^{(\mathrm{III})}_n = (\mathrm{DCT}^{(\mathrm{II})}_n)^T, \quad \text{and } \mathrm{DST}^{(\mathrm{III})}_n = (\mathrm{DST}^{(\mathrm{II})}_n)^T, \quad \text{where } (\cdot)^T = \text{transpose}.$$

The $\mathrm{DCT}^{(\mathrm{II})}$ and the $\mathrm{DCT}^{(\mathrm{IV})}$ are used in the image and video compression standards JPEG and MPEG, respectively [26].

The (rationalized) Haar transform is recursively defined by

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

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

$$\begin{aligned} \mathrm{CosDFT} &= \mathrm{Re}(\mathrm{DFT}_n), \quad \text{and} \\ \mathrm{SinDFT} &= \mathrm{Im}(\mathrm{DFT}_n). \end{aligned} \tag{14}$$

We list a subset of the rules considered by SPIRAL for the above transforms in Equations (15)–(28). Due to lack of space, we do not give the exact form of every matrix appearing in the rules, but simply indicate their type. In particular, $n \times n$ permutation matrices are denoted by P_n, P'_n, P''_n, diagonal matrices by D_n, other sparse matrices by S_n, S'_n, and 2×2 rotation matrices by $R_k, R_k^{(j)}$. The same symbols may have different meanings in different rules. By $A^P = P^{-1} \cdot A \cdot P$, we denote matrix conjugation; the exponent P is always a permutation matrix. The exact form of the occurring matrices can be found in [34, 33, 6].

$$\mathrm{DFT}_2 = \mathrm{F}_2 \tag{15}$$

$$\mathrm{DFT}_n = (\mathrm{DFT}_r \otimes \mathrm{I}_s) \cdot \mathrm{T}^n_s \cdot (\mathrm{I}_r \otimes \mathrm{DFT}_s) \cdot \mathrm{L}^n_r, \quad n = r \cdot s \tag{16}$$

$$\mathrm{DFT}_n = \mathrm{CosDFT}_n + i \cdot \mathrm{SinDFT}_n \tag{17}$$

$$\mathrm{DFT}_n = \mathrm{L}^n_r \cdot (\mathrm{I}_s \otimes \mathrm{DFT}_r) \cdot \mathrm{L}^n_s \cdot \mathrm{T}^n_s \cdot (\mathrm{I}_r \otimes \mathrm{DFT}_s) \cdot \mathrm{L}^n_r, \quad n = r \cdot s \tag{18}$$

$$\begin{aligned} \mathrm{DFT}_n = {} & (\mathrm{I}_2 \oplus (\mathrm{I}_{n/2-1} \otimes \mathrm{F}_2 \cdot \mathrm{diag}(1, i)))^{P_n} \cdot (\mathrm{DCT}^{(\mathrm{I})}_{n/2+1} \oplus (\mathrm{DST}^{(\mathrm{I})}_{n/2-1})^{P'_{n/2-1}}) \\ & \cdot (\mathrm{I}_2 \oplus (\mathrm{I}_{n/2-1} \otimes \mathrm{F}_2))^{P''_n}, \quad 2 \mid n \end{aligned} \tag{19}$$

$$\mathrm{CosDFT}_n = S_n \cdot (\mathrm{CosDFT}_{n/2} \oplus \mathrm{DCT}^{(\mathrm{II})}_{n/4}) \cdot S'_n \cdot \mathrm{L}^n_2, \quad 4 \mid n \tag{20}$$

$$\mathrm{SinDFT}_n = S_n \cdot (\mathrm{SinDFT}_{n/2} \oplus \mathrm{DCT}^{(\mathrm{II})}_{n/4}) \cdot S'_n \cdot \mathrm{L}^n_2, \quad 4 \mid n \tag{21}$$

$$\mathrm{DCT}^{(\mathrm{II})}_2 = \mathrm{diag}(1, \sqrt{1/2}) \cdot \mathrm{F}_2 \tag{22}$$

$$\mathrm{DCT}^{(\mathrm{II})}_n = P_n \cdot (\mathrm{DCT}^{(\mathrm{II})}_{n/2} \oplus (\mathrm{DCT}^{(\mathrm{IV})}_{n/2})^{P'_n}) \cdot (\mathrm{I}_{n/2} \otimes \mathrm{F}_2)^{P''_n}, \quad 2 \mid n \tag{23}$$

$$\mathrm{DCT}^{(\mathrm{IV})}_n = S_n \cdot \mathrm{DCT}^{(\mathrm{II})}_n \cdot D_n \tag{24}$$

$$\begin{aligned} \mathrm{DCT}^{(\mathrm{IV})}_n = {} & (\mathrm{I}_1 \oplus (\mathrm{I}_{n/2-1} \otimes \mathrm{F}_2) \oplus \mathrm{I}_1) \cdot P_n \cdot (\mathrm{DCT}^{(\mathrm{II})}_{n/2} \oplus (\mathrm{DST}^{(\mathrm{II})}_{n/2})^{P'_{n/2}}) \\ & \cdot (R_1 \oplus \ldots \oplus R_{n/2})^{P''_n}, \quad 2 \mid n \end{aligned} \tag{25}$$

$$\begin{aligned} \mathrm{DCT}^{(\mathrm{IV})}_{2^k} = {} & P_{2^k} \cdot (R_1 \oplus \ldots \oplus R_{2^{k-1}}) \\ & \cdot \left(\prod_{j=k-1}^{1} (\mathrm{I}_{2^{k-j-1}} \otimes \mathrm{F}_2 \otimes \mathrm{I}_{2^j}) \cdot \left(\mathrm{I}_{2^{k-j-1}} \otimes \left(\mathrm{I}_{2^j} \oplus R_1^{(j)} \oplus \ldots \oplus R_{2^j-1}^{(j)} \right) \right) \right) \cdot P'_{2^k} \end{aligned} \tag{26}$$

$$\mathrm{WHT}_{2^k} = \prod_{j=1}^{t} \left(\mathrm{I}_{2^{k_1+\cdots+k_{j-1}}} \otimes \mathrm{WHT}_{2^{k_j}} \otimes \mathrm{I}_{2^{k_{j+1}+\cdots+k_t}} \right), \quad k = k_1 + \cdots + k_t \tag{27}$$

$$\mathrm{RHT}_{2^k} = (\mathrm{RHT}_{2^{k-1}} \oplus \mathrm{I}_{2^{k-1}}) \cdot (\mathrm{F}_2 \otimes \mathrm{I}_{2^{k-1}}) \cdot \mathrm{L}^{2^k}_2, \quad k > 1 \tag{28}$$

The above rule can be informally divided into the following classes.

- *Base case rules* expand a transform of (usually) size 2 (e.g., Rules (15) and (22)).
- *Recursive rules* expand a transform in terms of similar (e.g., Rules (16) and (27)) or different (e.g., Rules (23) and (19)) transforms of smaller size.

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 1: Number of algorithms for DFT and $\mathrm{DCT}^{(\mathrm{IV})}$ of size 2^k, for $k = 1, \dots, 9$.

- *Transformation rules* expand a transform in terms of different transforms of the same size (e.g., Rules (17) and (24)).
- *Iterative rules* completely expand a transform (e.g., Rule (26)).

Further we note the following important facts.

- Some rules are *parameterized*, i.e., they have different instantiations. For example, Rule (16) depends on the factorization $n = r \cdot s$, which in general is not unique.
- For the DFT there are rules that are substantially different from the Cooley-Tukey rule (16) (e.g., Rule (19), which computes a DFT via DCTs and DSTs).
- The Cooley-Tukey variant proposed in [24] is represented by Rule (18), which arises from Rule (16) by replacing $\mathrm{DFT}_r \otimes \mathrm{I}_s$ with $\mathrm{L}^n_r \cdot (\mathrm{I}_s \otimes \mathrm{DFT}_r) \cdot \mathrm{L}^n_s$.

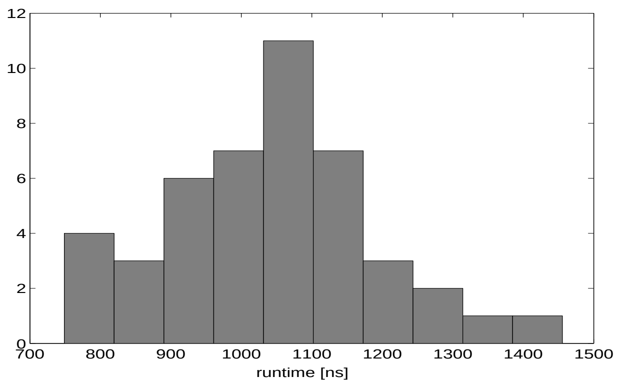
Inspecting rules 15 through 28 we confirm that these rules involve only a few constructs and primitives. In particular, enlarging the transform domain from the DFT and the Cooley-Tukey rule (16) to the trigonometric transforms (Rules (23)–(26)) requires only the addition of the direct sum $\oplus$ and of a few primitives like diagonal and permutation matrices. Other rules that can be represented using only the above constructs include split-radix FFT, Good-Thomas FFT, and Rader FFT, see [31]. A rule framework for FIR filters is presented in [14].

2.3 The 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. As an example, a DFT_{16} can be expanded using Rule (16) or Rule (19). If Rule (16) is chosen, then the actual expansion depends on the factorization of the size 16, namely one of $8 \cdot 2,\ 4 \cdot 4,\ 2 \cdot 8$. After the expansion, a similar degree of freedom applies to the smaller transform obtained. The net result is that, for a given transform, there is a very large number of fully expanded formulas, i.e., algorithms. For example, Table 1 shows the surprisingly large number of 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 1 show that, even for a modest transform 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 data flow during the computation, which leads to a large spread in the runtimes of the corresponding implementations, even for very small transform sizes.



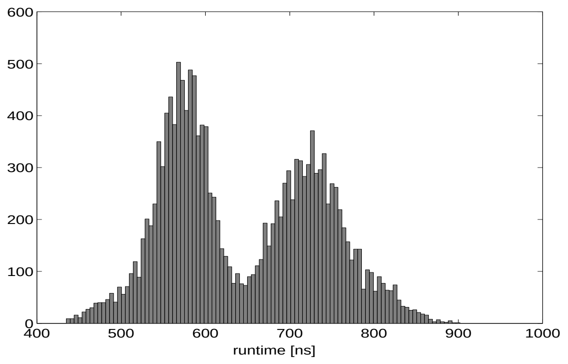


Figure 2: Histogram of the runtimes (in nanoseconds) of all 45 algorithms for a WHT_{2^5}, and all 15,778 algorithms for a $\mathrm{DCT}^{(\mathrm{IV})}_{16}$, implemented in straight-line code on a Pentium 4, 1.8 GHz, running Linux.

As an example, Figure 2 shows a histogram of runtimes for all 45 algorithms for a WHT_{2^5} and for all 15,778 algorithms for a $\mathrm{DCT}^{(\mathrm{IV})}_{16}$ implemented by SPIRAL in straight-line code (i.e., without using loops). The platform is a Pentium 4, 1.8 GHz, running Linux, using gcc 2.95. Each of the WHT algorithms requires precisely 80 additions and 80 subtractions. The runtimes range between about 750 and 1450 nanoseconds, which is around a factor of 2. For the $\mathrm{DCT}^{(\mathrm{IV})}$ algorithms the number of additions ranges from 96 to 104, and the number of multiplications ranges from 48 to 56. The runtimes are between approximately 430 and 900 nanoseconds, more than a factor of 2. In both cases, the fastest algorithms are rare. For example, for the $DCTIV$, only about 1.5% of the algorithms are within a 10% runtime range of the fastest algorithm.

2.4 Framework Summary

The mathematical framework presented in this section 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; 2) the automatic generation of formulas; and 3) the automatic translation of fully expanded formulas into programs. Further, to generate a *very fast* implementation, requires 4) a search module that controls the formula generation and possible implementation choices, such as the degree of unrolling.

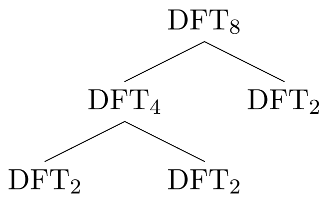
Taken together we obtain the architecture of SPIRAL displayed in Figure 1. The following three sections are devoted to the three key modules of SPIRAL: the formula generator (Section 3), the formula translator (Section 4), and the search module (Section 5).

3 Formula Generator

The task of the formula generator module within SPIRAL (see Figure 1) is to generate algorithms, given as formulas, for a user specified transform. The formula generator is interfaced with SPIRAL's search module, which controls the formula generation. In this section we overview the design and the main components of SPIRAL's formula generator.

The most important features of the formula generator are:

- *Extensibility.* The formula generator, and hence SPIRAL, can be easily expanded by including new transforms and new rules.



$$\leftrightarrow \quad \begin{array}{c}\Big(\big((\mathrm{F}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}_2^4 \cdot (\mathrm{I}_2 \otimes \mathrm{F}_2) \cdot \mathrm{L}_2^4\big) \otimes \mathrm{I}_2\Big) \\ \cdot\, \mathrm{T}_2^8 \cdot (\mathrm{I}_4 \otimes \mathrm{F}_2) \cdot \mathrm{L}_4^8\end{array}$$

Figure 3: A fully expanded ruletree for the DFT_8 and the corresponding fully expanded formula; the rules at the nodes are omitted.

- *Efficiency.* Formula generation is fast, i.e, does not constitute a bottleneck in SPIRAL's code generation process, and it is storage efficient, an important feature for some search methods (e.g., STEER, see Section 5), which concurrently work with a large number of algorithms.

In the following we overview how we realized these features by introducing appropriate data structures to represent transforms, rules, and algorithms. We conclude with a sketch of the internal architecture of the formula generator and some notes on its implementation.

3.1 Extensibility: Databases of Transforms and Rules

SPIRAL's framework (Section 2) shows that algorithms for a given transform arise from the successive application of a small number of rules. This fact leads naturally to an *extensible* design of the formula generator, in which transforms and rules are collected in respective databases. An entry for a transform in the transform database collects the information about the transform that is necessary for the operation of the formula generator. For example, definition, parameters, and dimension of the transform have to be known. Similarly, an entry for a rule in the rule database collects the necessary information about the rule, such as the associated transform, applicability conditions, and the actual structure of the rule. Thus, extension of the formula generator, and hence SPIRAL, with new transforms or rules require the user to create a new entry in the respective database.

3.2 Efficiency: Ruletrees

The formula generator represents formulas by a recursive data structure corresponding to their abstract syntax trees. For algorithms of large transform sizes this representation becomes storage intensive. Furthermore, several search algorithms (Section 5) require the local manipulation of algorithms, which is unduly difficult if they are represented as formulas.

To overcome this problem, the formula generator uses a different representation for algorithms, namely *ruletrees*. Every algorithm for a transform is determined by the sequence of rules applied in the expansion process. Thus we can represent an algorithm by a *ruletree* in which each node contains the transform at this stage and the rule applied to it. A ruletree is called *fully expanded*, if all rules in the leaves are base case rules. Fully expanded ruletrees correspond to fully expanded formulas and thus to algorithms.

As a simple example, Figure 3 shows a ruletree for the DFT_8 corresponding to Formula (12), which was derived by two applications of Rule (16) and 3 applications of the (base case) Rule (15); we omitted the rule names in the nodes.

Ruletrees are storage efficient; each node only contains pointers to the appropriate transform and rule in the database. Furthermore, ruletrees can be easily manipulated, e.g., by expanding a subtree in a different way. The efficient representation also leads to the very fast generation of ruletrees. On current computing platforms, thousands of trees can be generated in a few seconds.

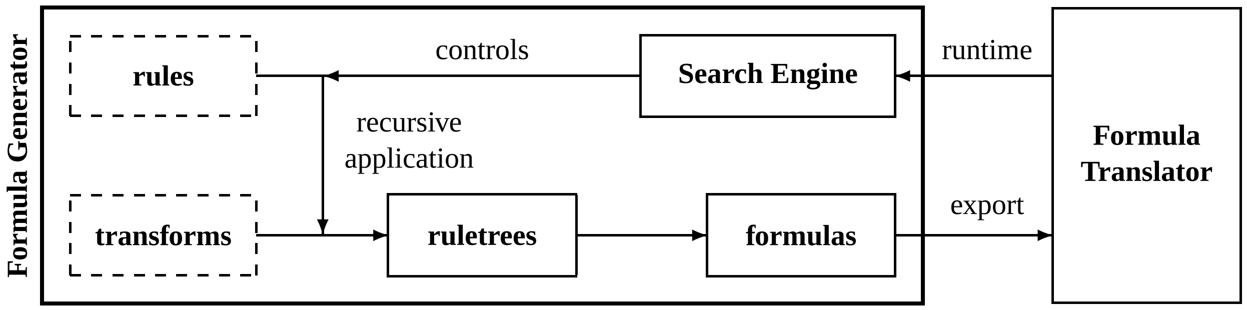


Figure 4: Internal architecture of the formula generator including the search module. The main components are recursive data types for representing ruletrees and formulas, and extensible databases (dashed boxes) for rules and transforms.

3.3 Infrastructure and Implementation

The internal architecture of the formula generator, including the search engine, is displayed in Figure 4. 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 (explained in Section 4). The runtime of the generated programs is returned to the search engine, which controls the generation of the next set of ruletrees (see Section 5).

Formula generation and formula manipulation fall into the realm of symbolic computation, which led us to choose the language and computer algebra system GAP [12], including AREP [5], as an implementation platform. GAP provides the infrastructure for symbolic computation with a variety of algebraic objects. The GAP share package AREP is focused on structured matrices and their symbolic manipulation. A high level language like GAP facilitates the implementation of the formula generator. As an additional advantage, GAP provides *exact* arithmetic for square roots, roots of unity, and trigonometric expressions that make up the entries of most DSP transforms and formulas.

4 Formula Translator

The task of the formula translator module within SPIRAL is to translate fully expanded formulas generated by the formula generator into programs. Currently, the translated programs are either C or Fortran procedures, though other languages including assembly or machine code could be produced. Once a formula has been translated into a program, it can be executed and timed by the performance evaluation component and the resulting time returned to the search engine. This allows SPIRAL to search for fast implementations of DSP transforms using the mathematical framework presented in Section 2.

The formulas input to the formula translator are represented in the SPL language. SPL is a domain-specific language for representing structured matrix factorizations. SPL borrows concepts from TPL (Tensor Product Language) [1]. SPL programs consist of formulas that are symbolic representations of structured matrix factorizations of matrices with fixed row and column dimensions. Each formula corresponds to a fixed transform matrix, which can be obtained by evaluating the formula. Alternatively, the formula can be interpreted as an algorithm for applying the transform represented by the formula to an arbitrary input vector. Using the structure in a formula, the SPL compiler translates the formula into a procedure for applying the corresponding transform to the input vector.

A key feature of the SPL compiler is that the code generation process can be controlled by the user without modifying the compiler. This is done through the use of compiler directives and a template mechanism called meta-SPL. Meta-SPL allows SPIRAL, in addition to searching through the space of algorithms, to search through the space of possible implementations for a given formula. A key research question is to determine which optimizations should be performed by default by the SPL compiler and which optimizations should be controlled by the search engine in SPIRAL. Some optimizations, such as common subexpression elimination, are always applied; however, other potential optimizations, such as loop unrolling, for which it is not clear when and to what level to apply, are implementation parameters for the search engine to explore. It is worth noting that it is necessary for the SPL compiler to apply standard compiler optimizations such as common subexpression elimination, rather leaving them to the backend C or Fortran compiler, since these compilers typically do not fully utilize these optimizations on the type of code produced by the SPL compiler [38].

The SPL representation of algorithms around which SPIRAL is built is very different from FFTW's codelet generator, which represents algorithms as a collection of arithmetic expression trees for each output, which then is translated into a dataflow graph using various optimizations [10]. In both cases, the representation is restricted to a limited class of programs corresponding to linear computations of a fixed size, which have simplifying properties such as no side-effects and no conditionals. This opens the possibility for domain-specific optimizations. Advantages of the SPL representation of algorithms as generated by the formula generator include the following: 1) Formula generation is separated from formula translation and both the formula generator and formula translator can be developed, maintained, and used independently. In particular, the formula generator can be used by DSP experts to include new transforms and algorithms. SPL provides a natural language to transfer information between the two components. 2) The representation is concise. The arithmetic expression tree representation used in FFTW expresses each output as linear function of all n inputs and thus grows roughly as $O(n^2 \log(n))$, which restricts its application to small transform sizes (which, of course, is the intended scope and sufficient in FFTW). 3) High-level mathematical knowledge is maintained, and this knowledge can be used to obtain optimizations and program transformations not available using standard compiler techniques. This is crucial, for example, for short-vector code generation (Section 4.4). 4) The mathematical nature of SPL allows other programs, in our case the formula generator, to easily manipulate and derive alternate programs. Moreover, algorithms are expressed naturally using the underlying mathematics. 5) SPL provides hooks that allow alternative code generation schemes and optimizations to be controlled and searched externally without modifying the compiler.

In the following three subsections we describe SPL, meta-SPL, and the SPL compiler, respectively. An overview of the language and compiler will be given, and several examples will be provided, illustrating the syntax of the language and the translation process used by the compiler. Additional information may be found in [38] and [37]. We conclude with a brief overview of an extension to the SPL compiler that generates short vector code for last generation platforms that feature SIMD (single-instruction multiple-data) instruction set extensions.

4.1 SPL

In this section we describe the constructs and syntax of the SPL language. Syntax is described informally guided by the natural mathematical interpretation. A more formal treatment along with a BNF grammar is available in [37]. In the next section we describe meta-SPL, a meta-language that is used to define the semantics of SPL and allows the language to be extended.

SPL programs consist of the following: 1) SPL formulas, representing fully expanded formulas in the sense of Section 2.1; 2) constant expressions for entries appearing in formulas; 3) define statements for assigning names to formulas or constant expressions; and 4) compiler directives and type declarations. Each formula corresponds to a factorization of a real or complex matrix of a fixed size. The size is determined from the formula using meta-SPL, and the type is specified as real or complex. Meta-SPL is also used to define the symbols occurring in formulas. Rather than constructing matrices, meta-SPL is used by the compiler to generate a procedure for applying the transform (as represented by a formula) to an input vector of the given size and type.

Constant expressions. The elements of a matrix can be real or complex numbers. Complex numbers $a+b\sqrt{-1}$ are represented by the pair of real numbers (a, b). In SPL, these numbers can be specified as scalar constant expressions, which may contain function invocations and symbolic constants like `pi`. For example, `12`, `1.23`, `5*pi`, `sqrt(5)`, and `(cos(2*pi/3.0),sin(2*pi/3))` are valid scalar SPL expressions. All constant scalar expressions are evaluated at compile-time.

SPL formulas. SPL formulas are built from general matrix constructions, parameterized symbols denoting families of special matrices, and matrix operations such as matrix composition, direct sum, and the tensor product. Each construction has a list of arguments that uniquely determine the corresponding matrix. The distinction between the different constructions is mainly conceptual; however it also corresponds to different argument types.

SPL uses a prefix notation similar to Lisp to represent formulas. The following lists example constructions that are provided from each category. However, it is possible to define new general matrix constructions, parameterized symbols, and matrix operations using meta-SPL.

General matrix constructions. Let a_{ij} denote an SPL constant and i_k, j_k, and σ_k denote positive integers. Examples include the following.

- `(matrix (`a_{11} `...` a_{1n}`) ... (`a_{m1} `...` a_{mn}`))` - the $m \times n$ matrix $[a_{ij}]_{1\leq i\leq m,\ 1\leq j\leq n}$.
- `(sparse (`i_1 j_1 $a_{i_1 j_1}$`) ... (`i_t j_t $a_{i_t j_t}$`))` - the $m \times n$ matrix where $m = \max(i_1, \ldots, i_t)$, $n = \max(j_1, \ldots, j_t)$ and the non-zero entries are $a_{i_k j_k}$ for $k = 1, \ldots, t$.
- `(diagonal (`a_1 `...` a_n`))` - the $n \times n$ diagonal matrix $\mathrm{diag}(a_1, \ldots, a_n)$.
- `(permutation (`σ_1 `...` σ_n`))` - the $n \times n$ permutation matrix: $k \mapsto \sigma_k$, for $k = 1, \ldots, n$.

Parameterized Symbols. Parameterized symbols represent families of matrices parameterized by integers. Examples include the following.

- `(I n)` - the $n \times n$ identity matrix I_n.
- `(F n)` - the $n \times n$ DFT matrix F_n.
- `(L n s)` - the $n \times n$ stride permutation matrix L^n_s, where $s|n$.
- `(T n s)` - the $n \times n$ twiddle matrix T^n_s, where $s|n$.

Matrix operations. Matrix operations take a list of SPL formulas, i.e. matrices, and construct another matrix. In the following examples, A and A_i are arbitrary SPL formulas and P is an SPL formula corresponding to a permutation matrix.

- `(compose` A_1 `...` A_t`)` - the matrix product $A_1 \cdots A_t$.
- `(direct-sum` A_1 `...` A_t`)` - the direct sum $A_1 \oplus \cdots \oplus A_t$.
- `(tensor` A_1 `...` A_t`)` - the tensor product $A_1 \otimes \cdots \otimes A_t$.
- `(conjugate A P)` - the matrix conjugation $A^P = P^{-1} \cdot A \cdot P$, where P is a permutation.

Define Statements are provided for assigning names to formulas or constant expressions. They provide a short-hand for entering subformulas or constants in formulas.

- `(define name formula)`
- `(define name constant-expression)`

Compiler Directives. There are two types of compiler directives. The first type is used to specify the matrix type, and the second type is used to influence the code produced by the compiler.

- `#datatype REAL | COMPLEX` - set the type of the input and output vectors.
- `#subname name` - name of the procedure produced by the compiler for the code that follows.
- `#codetype REAL | COMPLEX` - if the datatype is complex, indicate whether complex are real variables will be used to implement complex arithmetic in the generated code.
- `#unroll ON | OFF` - if `ON` generate straight-line code and if `OFF` generate loop code.

```
#datatype COMPLEX
#codetype REAL
#unroll ON
(define F4
  (compose
    (tensor (F 2) (I 2))
    (T 4 2)
    (tensor (I 2) (F 2)
    (L 4 2)))
#subname F_8
#unroll OFF
(compose
   (tensor F4 (I 2))
   (T 8 2)
   (tensor (I 4) (F 2))
   (L 8 4))
```

```
#datatype REAL
#unroll ON
#subname DCT2_4
(compose
  (permutation (1 3 2 4))
  (direct_sum
    (compose (diagonal (1 sqrt(1/2))) (F 2))
    (matrix
      ( cos(13*pi/8) sin(13*pi/8))
      (-sin(13*pi/8) cos(13*pi/8))
    )
  )
  (permutation (1 3 2 4))
  (tensor (I 2) (F 2))
  (permutation (1 4 2 3))
)
```

Figure 5: SPL expressions for DFT_8 and $\mathrm{DCT}_4^{(\mathrm{II})}$.

Figure 5 shows SPL expressions for the fully expanded formulas for the transforms DFT_8 and $\mathrm{DCT}_4^{(\mathrm{II})}$, corresponding to (12) and (11), respectively. These examples use all of the components of SPL including parameterized symbols, general matrix constructions, matrix operations, constant expressions, define statements, and compiler directives. These SPL formulas will be translated into C or Fortran procedures with the names `F_8` and `DCT2_4` respectively. The procedure `F_8` has a complex input and output of size 8 and uses a mixture of loop and straight-line code. Complex arithmetic is explicitly computed with real arithmetic expressions. The procedure `DCT2_4` has a real input and output of size 4 and is implemented in straight-line code.

4.2 Meta-SPL

The semantics of SPL programs are defined in meta-SPL using a template mechanism. Templates tell the compiler how to generate code for the various symbols that occur in SPL formulas. In this way, templates are used to define general matrix constructions, parameterized symbols and matrix operations, including those built-in and those newly created by the user. They also provide a mechanism to compute the input and output dimensions of the matrices corresponding to a formula. In addition to templates, meta-SPL can define functions (scalar, vector, and matrix) that can be used in template definitions. Finally, statements are provided to inform the parser of the symbols that are defined by templates.

Symbol definition, intrinsic functions, and templates. Meta-SPL provides the following directives to introduce parameterized matrices, general matrix constructions, matrix operators, intrinsic functions and templates.

- `(primitive name shape)` - introduce new parameterized symbol.
- `(direct name size-rule)` - introduce a new general matrix construction.
- `(operation name size-rule)` - introduce new matrix operation.
- `(function name <arguments> <dimension> expression)` - define an intrinsic function.
- `(template formula [condition] (i-code-list))` - define a template.

The parameters `shape` and `size-rule` specify how the row and column dimensions of the represented matrix are computed.

```
(template (T n s)  ;; ---- n and s are integer parameters
  [n >= 1 && s >= 1 && n%s == 0]
  ( coldim = n
    rowdim = n
    for i=0,...,n-1
      y(i) = w(n,i*s) * x(i)
    end ) )
```

Figure 6: Template for `(T n s)`.

A template contains a pattern followed by an optional guard condition and a code sequence using an intermediate code representation called i-code. When the SPL compiler encounters an expression that matches the pattern and satisfies the guard condition, it inserts the corresponding i-code into the translated program, where the parameters are replaced by the values in the matched expression.

Templates use the convention that the input and output vectors are always referred to by the names x and y and have sizes given by `coldim` and `rowdim`. Intermediate code can refer to x, y, the input parameters, and temporary variables. The code consists of a sequence of two operand assignments and conditionals are not allowed. Loops are allowed; however, the number of iterations, once the template is instantiated will always be constant.

The following examples illustrate how templates are used to define parameterized symbols and matrix operations. A detailed description of templates is provided in [37]. Note that the syntax used here is slightly simplified to allow for a more accessible presentation. Also row and column dimensions are computed explicitly rather than relying on size and shape rules.

Figure 6 shows the template definition for the parameterized symbol `(T n s)`. The pattern `(T n s)` will match any SPL expression containing the symbol `T` in the first position followed by two integer parameters. The guard condition specifies that the two integer parameters are positive and the second parameter divides the first. The resulting code multiplies the input vector `x` by constants produced from intrinsic function calls: `w(n, k)` $= \omega_n^k$.

Figure 7 provides template definitions for the matrix operations `compose` and `tensor`. These examples show how to apply matrix operations to code sequences and is the foundation for the translation process. Given i-code for the sub-expressions representing the operands of the matrix operation, the code construction creates a code sequence for the matrix obtained by applying the operation to the operand matrices.

The code for the matrix composition $y = (AB)x$ is obtained by applying the code for B to the input x and assigning the output to the temporary vector t, introduced by `deftemp`, and then applying the code for A to t to get the output y . The i-code for the matched parameters A and B is called using the call statement. The call is inlined with appropriate parameter substitution and index adjustment. For vector indexing we use the notation start:stride:end, e.g., $1 : 2 : 7 = 1, 3, 5, 7$.

The code for the tensor product of an $m \times n$ matrix A and a $p \times q$ matrix B is obtained from the factorization $A \otimes B = (A \otimes \mathrm{I}_p)(\mathrm{I}_n \otimes B)$. The two factors are combined using composition. The first factor simply loops over n calls to the code for B, and the second factor loops over p calls to the code for A; however, in the latter case, the data is accessed at stride p.

4.3 SPL Compiler

This section describes the organization of the SPL compiler and illustrates the process used to translate SPL formulas into programs. In addition, mechanisms are described that allow SPIRAL to search over different implementation strategies.

```
(template (compose A B)  ; y = (A B) x = A(B(x)).
  ( deftemp t(B.rowdim)
    coldim = A.rowdim
    rowdim = B.coldim
    t(0:1:B.rowdim-1) = call B(x(0:1:B.coldim-1))
    y(0:1:A.rowdim-1) = call A(t(0:1:B.rowdim-1)) ) )

(template (tensor A B)  ; y = (A tensor B) x
  ( rowdim = A.rowdim * B.rowdim
    coldim = A.coldim * B.coldim
    deftemp t(A.coldim*B.rowdim)

    for i=0:A.coldim-1
        t(i*B.rowdim:1:(i+1)*B.rowdim-1) =
          call B(x(i*B.coldim:1:(i+1)*B.coldim-1));
    end
    for j=0:B.rowdim-1
        y(j*A.rowdim:B.rowdim:(j+1)*A.rowdim-1) =
           call A(t(j*A.coldim:B.rowdim:(j+1)*A.coldim -1 )
    end ) )
```

Figure 7: Templates for `compose` and `tensor`.

The SPL compiler translates an SPL formula (matrix factorization) into an efficient program (currently in C or Fortran) to compute the matrix-vector product of the matrix given by the SPL formula. Translation proceeds by applying various transformations, corresponding to the algebraic operators in SPL, to code segments starting with code for the matrices occurring in the SPL formula. The code segments and transformations are defined by the template mechanism in meta-SPL discussed in the previous section. Meta-SPL also provides a mechanism to control the optimization and code generation strategies used by the compiler.

The input to the SPL compiler consists of an SPL program, a meta-SPL template, and a set of interspersed compiler directives. The SPL program and meta-SPL can be intermixed in the input so long as the definition of any new symbol appears before its use. The output of the SPL compiler is a set of Fortran or C procedures which compute the matrix-vector products corresponding to all of the top level SPL formulas in the SPL program. The compiler proceeds in five steps: 1) parsing; 2) intermediate code generation; 3) intermediate code restructuring; 4) optimization; and 5) target code generation; as illustrated in Figure 8.

Parsing. The parser creates three data structures from the input SPL and meta-SPL program: a set of abstract syntax trees (AST), a table containing templates, and a symbol table. Each SPL formula is translated into an abstract syntax tree (AST). The leaf nodes of an AST contain primitive matrices and the internal nodes correspond to matrix operators. The AST is a binary tree; n-ary formulas, such as `(compose A1 ... An)` are associated right-to-left. Template definitions are stored in the template table. Each entry in the template table contains an AST which represents the pattern, an arithmetic expression tree which represents the condition, and a linked-list that holds the i-code. Each name defined by `define`, `primitive`, `operation`, `direct`, or `function` is stored in the symbol table.

Intermediate Code Generation. I-code is generated for each AST created from the SPL program using the necessary symbol values and template definitions obtained from the symbol and template tables. A recursive top-down pattern matching algorithm is used to match the symbols occurring in the AST with templates in the template table. After matching a template, parameters are evaluated and the template i-code

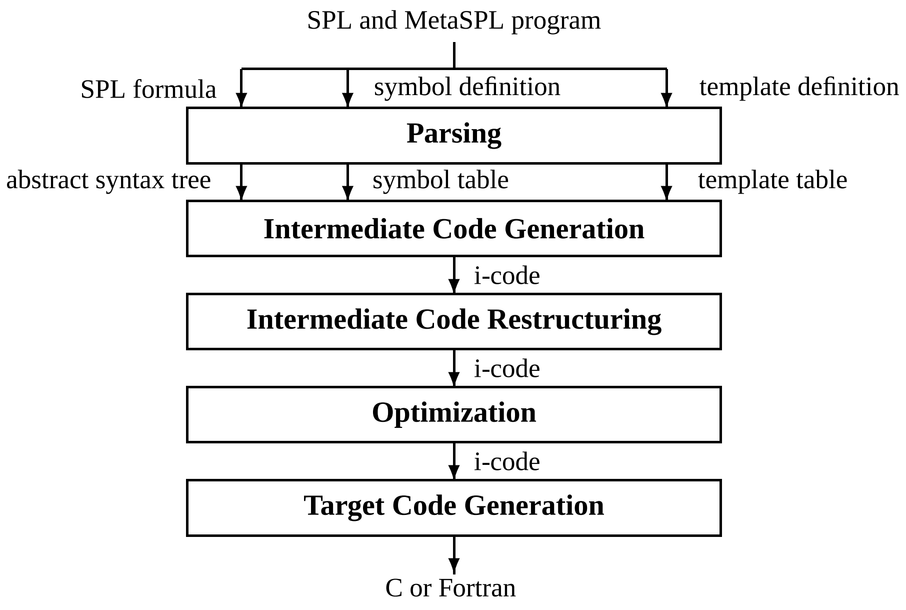


Figure 8: The SPL compiler.

is inserted into the i-code for the AST from the bottom up. The i-code for a node in the AST is constructed according to the matching template definition from the i-code of its children. As the i-code is combined it may be necessary to rename variables, in-line call statements, and unify address expressions.

Intermediate Code Restructuring. Several code transformations, depending on the compiler options and directives, are performed. These transformations include loop unrolling, complex to real arithmetic conversion and intrinsic function evaluation.

Optimization. After the code restructuring the compiler performs some basic optimization on the i-code sequence. These optimizations, which include constant folding, copy propagation, common subexpression elimination, and dead code elimination, have proven to be necessary to achieve good performance when the target C/Fortran compiler is not aggressive enough with its optimizations. Recently, code reordering (or scheduling) for locality was included as optional optimization [23] but is not discussed in this paper.

Target Code Generation. The optimized i-code is translated to the target language, currently C or Fortran.

An Example. We illustrate the entire compilation process for the SPL formula

```
(compose (tensor (F 2) (I 2)) (T 4 2) (tensor (I 2) (F 2)) (L 4 2))
```

corresponding to the formula for a DFT_4 given in Equation (4). Figure 9 shows the AST for this formula. Since the DFT_4 is a complex transform, the formula is compiled with the flag `#datatype COMPLEX` (see Section 4.1). Accordingly, the i-code generated in the compilation steps operates on complex numbers.

The SPL compiler processes the AST in Figure 9 bottom-up starting with the leaves. Using the template definitions for `F`, `L`, and `T`, the following code is produced. The constant `(0.0,1.0)` denotes $i = \sqrt{-1}$.

```
F2 :=                  T42 :=                 L42 :=
y(2) = x(1) - x(2)     y(1) = x(1)            y(1) = x(1)
y(1) = x(1) + x(2)     y(2) = x(2)            y(2) = x(3)
                       y(3) = x(3)            y(3) = x(2)
                       y(4) = (0.0,1.0)*x(4)  y(4) = x(4)
```

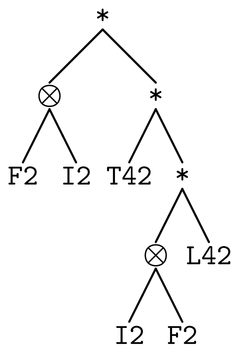


Figure 9: The abstract syntax tree (AST) for the DFT_4 formula in (4).

Next, templates for the special cases, $\mathrm{I} \otimes A$ and $B \otimes \mathrm{I}$, of the tensor product are used to construct code for `(tensor (I 2) (F 2))` and `(tensor (F 2) (I 2))`, respectively. The code for `(compose (tensor (I 2) (F 2)) (L 4 2))` is obtained by concatenating the code for `(tensor (I 2) (F 2))` with the code for `(L 4 2))`. An optimization removes unnecessary temporaries, i.e., the permutation is translated into a readdressing of the input variables for `(tensor (I 2) (F 2))` according to `(L 4 2)`. We display the i-code generated for all internal nodes in the AST in Figure 9.

```
F2 ⊗ I2 :=                (I2 ⊗ F2)L42 :=           T42(I2 ⊗ F2)L42 :=
y(1) = x(1) + x(3)        y(1) = x(1) + x(3)        y(1) = x(1) + x(3)
y(3) = x(1) - x(3)        y(2) = x(1) - x(3)        y(2) = x(1) - x(3)
y(2) = x(2) + x(4)        y(3) = x(2) + x(4)        y(3) = x(2) + x(4)
y(4) = x(2) - x(4)        y(4) = x(2) - x(4)        f = x(2) - x(4)
                                                    y(4) = (0.0,1.0)*f
```

Finally the code for the root node is generated. At this stage the abovementioned optimizations are applied including common subexpression elimination or (the optional) conversion from complex to the interleaved format (alternately real and imaginary part) invoked by the tag `#codetype REAL`. In our example the following code is generated for `#codetype COMPLEX` and `#codetype REAL`, respectively:

```
F4 :=                            F4 :=
f0 = x(1) - x(3)                 f0 = x(1) - x(5)
f1 = x(1) + x(3)                 f1 = x(2) - x(6)
f2 = x(2) - x(4)                 f2 = x(1) + x(5)
f3 = x(2) + x(4)                 f3 = x(2) + x(6)
y(3) = f1 - f3                   f4 = x(3) - x(7)
y(1) = f1 + f3                   f5 = x(4) - x(8)
f6 = (0.0,1.0) * f2              f6 = x(3) + x(7)
y(4) = f0 - f6                   f7 = x(4) + x(8)
y(2) = f0 + f6                   y(5) = f2 - f6
                                 y(6) = f3 - f7
                                 y(1) = f2 + f6
                                 y(2) = f3 + f7
                                 y(7) = f0 + f5
                                 y(8) = f1 - f4
                                 y(3) = f0 - f5
                                 y(4) = f1 + f4
```

Loop Code. The code generated in the previous example consists entirely of straight-line code. The SPL compiler can be instructed to generate loop code by using templates with loops such as those for the twiddle symbol and the tensor product. Using these templates, the SPL compiler translates the expression `(compose (F 2) (I 4)) (T 8 4))` into the following Fortran code.

```
do i0 = 0, 7
  t0(i0+1) = T8_4(i0+1) * x(i0+1)
end do
do i0 = 0, 3
  y(i0+5) = t0(i0+1) - t0(i0+5)
  y(i0+1) = t0(i0+1) + t0(i0+5)
end do
```

Since loops are used, constants must be placed in an array so that they can be indexed by the loop variable. Initialization code is produced by the compiler to initialize arrays containing constants. In this example, the constants from T^8_4 are placed in the array `T8_4`.

Observe that the data must be accessed twice: once to multiply by the twiddle matrix and once to compute $F_2 \otimes I_4$. By introducing an additional template that matches the expression that combines $F_2 \otimes I_4$ and the preceding twiddle matrix, the computation can be performed in one pass.

```
do i0 = 0, 3
  r1 = 4 + i0
  f0 = T8_4(r1+1) * x(i0+5)
  y(i0+5) = x(i0+1) - f0
  y(i0+1) = x(i0+1) + f0
end do
```

This example shows how the user can control the compilation process through the use of templates.

In order to generate efficient code it is often necessary to combine loop code and straight-line code. This can be accomplished through the use of loop unrolling. After matching a template pattern with loops the instantiated code has constant loop bounds. The compiler may be directed to unroll the loops, fully or partially to reduce loop overhead and increase the number of choices in instruction scheduling. When the loops are fully unrolled, not only is the loop overhead eliminated but it also becomes possible to substitute scalar expressions for array elements. The use of scalar variables tends to improve the quality of the code generated by Fortran and C compilers which are usually unable to analyze codes containing array subscripts. The down side of unrolling is the increase in code size.

In SPL, the degree of unrolling can be specified for the whole program or for a single formula. For example, the compiler option `-B32` instructs the compiler to fully unroll all loops in those sub-formulas whose input vector is smaller than or equal to 32. Individual formulas can be unrolled through the use of the `#unroll` directive. For example

```
#unroll on
(define I2F2 (tensor (I 2) (F 2)))
#unroll off
(tensor (I 32) I2F2)
```

will be translated into the following code.

```
do i0 = 0, 31
  y(4*i0+2) = x(4*i0+1) - x(4*i0+2)
```

```
    y(4*i0+1) = x(4*i0+1) + x(4*i0+2)
    y(4*i0+4) = x(4*i0+3) - x(4*i0+4)
    y(4*i0+3) = x(4*i0+3) + x(4*i0+4)
  end do
```

The ability to control code generation through the use of compiler options, compiler directives and templates allows the SPIRAL system to search over implementation strategies by setting compiler flags, inserting directives, and inserting template definitions.

4.4 Short Vector Extension of the SPL Compiler

Most recent micro-architectures feature special instruction sets that have the potential to considerably speed-up computation. Examples include fused multiply-add instructions and short vector SIMD (single instruction, multiple data) instructions. Examples for the latter include SSE on Pentium III and 4 and SSE2 on Pentium 4. For example, using SSE, four single-precision floating point additions or multiplications can be performed in a single instruction and, on Pentium III/4, in one cycle.

Compiler vectorization to date is limited to very simply structured code and looping patterns and fails on more complex algorithms such as fast DSP transforms. Thus, to obtain optimal performance for these algorithms, careful hand-tuning, often in assembly code, is necessary. In [8, 9] we have extended SPIRAL to automatically generate short-vector code for various architectures. The generated code is very competitive with the best available code including the short-vector DFT library provided by Intel. The key to obtaining this high-performance is automatic formula manipulation that transforms a given formula, using mathematical rules, into a form suitable for mapping into short-vector code. This manipulation is enabled through the mathematical representation of formulas and is not feasible using, for example, a C code representation of an algorithm. We do not explain the approach to greater detail in this paper, but refer the reader to [8, 9].

5 Search Engine

To find a platform-adapted implementation for a given transform, SPIRAL considers the space of algorithms and their possible implementations. On the algorithmic level, the degrees of freedom are given by the many possible fully expanded formulas (or ruletrees) for the transform. For a given formula, there are degrees of freedom in generating code, one important example being the choice of the unrolling strategy. The space of alternative implementations is too large to be tested exhaustively (e.g., Table 1) and exhibits a wide variation in runtimes (e.g., Figure 2).

The formula generator (Section 3) and the formula translator (Section 4) can generate any of these different possible implementations. The task of the search engine is to *intelligently* search the space of implementations for the optimal one for the given platform. The search is performed in a feedback loop; runtimes of previously generated implementations are used to control formula generation and code generation for further implementations.

We have implemented a number of different search methods within SPIRAL, including exhaustive search, dynamic programming, random search, hill climbing search, and STEER, a stochastic evolutionary search algorithm. Further, we have developed a "meta-search" algorithm that searches for the fastest implementation for a specified length of time using a combination of the search algorithms indicated above. Each of the search algorithms operates with the ruletree representation of algorithms (see Section 3.2) and optionally searches over implementation degrees of freedom. These implementation choices are either varied globally, i.e., for entire ruletrees, or applied to specific nodes in the ruletree by setting appropriate flags. For example, a flag "unrolling" in a node of a ruletree ensures that the code generated for that node contains no loops.

The different search algorithms are described in more detail below.

5.1 Exhaustive Search

Exhaustive search is straightforward. It generates all the possible implementations for a given transform and times each one to determine the fastest one. This search method is only feasible for very small transform sizes, as the number of possible algorithms for most transforms grows exponentially with the transform size. For example, Table 1 shows that exhaustive search becomes impractical for DFTs of size 2^6 and for $\mathrm{DCT}^{(\mathrm{IV})}$'s of size 2^5.

5.2 Dynamic Programming

Dynamic programming (DP) is a common approach to search in this type of domain [18, 11, 16, 28]. DP recursively builds a table of the best ruletrees found for each transform. A given transform is expanded once using all applicable rules. The ruletrees, i.e., expansions, of the obtained children are looked up in the table. If a ruletree is not present, then DP is called recursively on this child. Finally, the table is updated with the fastest ruletree of the given transform. DP usually times fewer ruletrees than the other search methods.

Dynamic programming makes the following assumption:

> **Dynamic Programming Assumption:** The best ruletree for a transform is also the best way to split that transform in a larger ruletree.

This assumption does not hold in general; the performance of a ruletree varies greatly depending on its position in a larger ruletree due to the pattern of data flow and the internal state of the given machine. In practice, though, DP usually finds reasonably fast formulas. A variant of DP considered by SPIRAL keeps track of the n best ruletrees found at each level, thus relaxing the DP assumption.

5.3 Random Search

Random search generates a number of random ruletrees and chooses the fastest. Note that it is a nontrivial problem to uniformly draw ruletrees from the space of all possibilities, i.e., algorithms. Thus, in the current implementation, a random ruletree is generated by choosing (uniformly) a random rule in each step of the expansion of the given transform.

Random search has the advantage that it times as few or as many formulas as the user desires, but leads to poor results if the fast ruletrees are scarce.

5.4 STEER

STEER (Split Tree Evolution for Efficient Runtimes) uses a stochastic, evolutionary search approach [29, 30]. STEER is similar to genetic algorithms [15], except that, instead of using a bit vector representation, it uses ruletrees as its representation. Unlike random search, STEER uses evolutionary operators to stochastically guide its search toward more promising portions of the space of formulas.

Given a transform and size of interest, STEER proceeds as follows:

1. Randomly generate a population P of legal ruletrees of the given transform and size.
2. For each ruletree in P, obtain its running time.
3. Let P_{fastest} be the set of the b fastest trees in P.
4. Randomly select from P, favoring faster trees, to generate a new population P_{new}.
5. Cross-over c random pairs of trees in P_{new}.
6. Mutate m random trees in P_{new}.

7. Let $P \leftarrow P_{\text{fastest}} \cup P_{\text{new}}$.
8. Repeat step 2 and following.

All selections are performed with replacement so that P_{new} may contain multiple copies of the same tree. Since timing ruletrees is expensive, runtimes are cached and only new ruletrees in P at step 2 are actually timed.

During this process, the ruletrees may not be optimized as a whole, but still include subtrees that are very efficient. Crossover [15] provides a method for exchanging subtrees between two different ruletrees, potentially allowing one ruletree to take advantage of a better subtree in another ruletree. Crossover on a pair of ruletrees t_1 and t_2 proceeds as follows:

1. Let n_1 and n_2 be random nodes in t_1 and t_2 respectively such that n_1 and n_2 represent the same transform and size.
2. If no n_1 and n_2 exists, then the trees can not be crossed-over.
3. Otherwise, swap the subtrees rooted at n_1 and n_2.

Mutations [15] make changes to ruletrees to introduce new diversity to the population. If a given ruletree performs well, then a small modification to the ruletree may perform even better. Mutations provide a method for searching the space of similar ruletrees. STEER uses three different mutations:

- *Regrow:* Remove the subtree under a node and grow a new random subtree.
- *Copy:* Find two nodes within the ruletree that represent the same transform and size. Copy the subtree underneath one node to the subtree of the other.
- *Swap:* Find two nodes within the ruletree that represent the same transform and size. Swap the subtrees underneath the two nodes.

These mutations are illustrated in Figure 10.

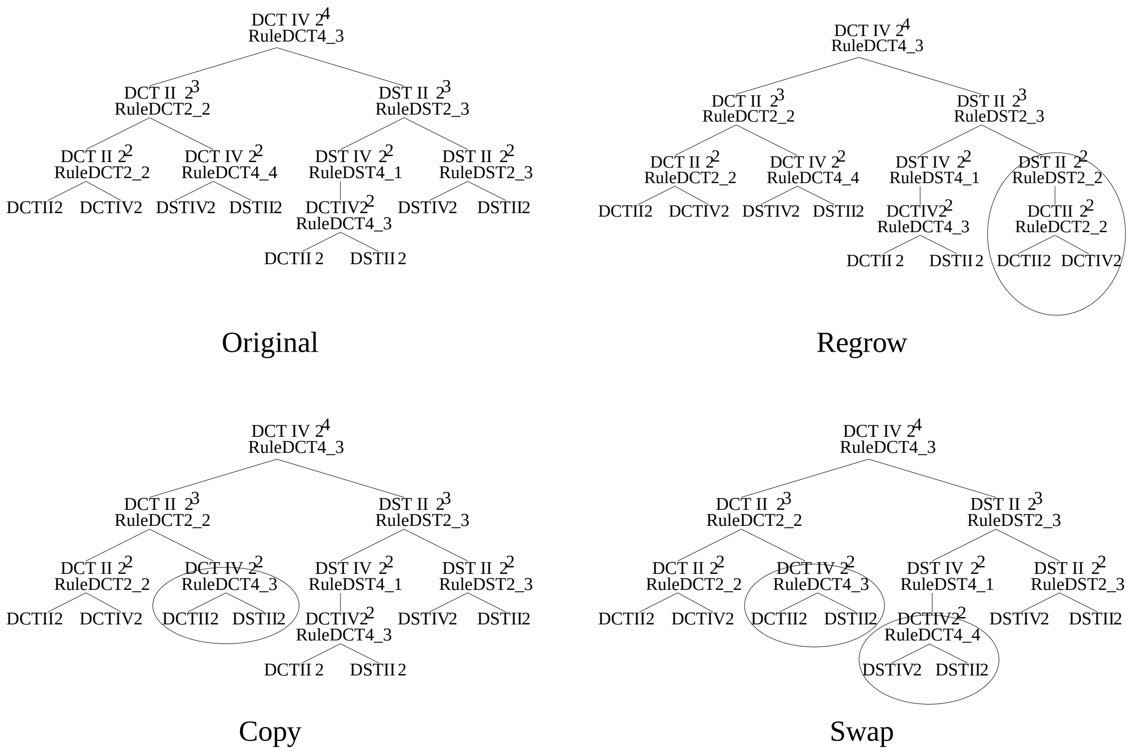


Figure 10: Examples of mutations performed on the tree labeled “Original.” Areas of interest are circled.

As with the other search methods, STEER optionally searches over implementation options such as the unrolling strategy. For example, enabling a “local unrolling” option determines, for each node in the ruletree, whether it should be translated into loop code or unrolled code. The unrolling decisions within a ruletree are subject to mutations within reasonable limits.

STEER explores a larger portion of the space of ruletrees than dynamic programming, while still remaining tractable. Thus, STEER often finds faster formulas than dynamic programming, at the small cost of a more expensive search.

5.5 Hill Climbing Search

Hill climbing is a refinement of random search, but is not as sophisticated as STEER. First, hill climbing generates a random ruletree and times it. Then, it randomly mutates this ruletree with the mutations defined for STEER, and times the resulting ruletree. If the new ruletree is faster, hill climbing continues by applying a mutation to the new ruletree and the process is repeated. Otherwise, if the original ruletree was faster, hill climbing applies another random mutation to the original ruletree and the process is repeated. After a certain number of mutations, the process is restarted with a completely new random ruletree.

5.6 Timed Search

When there is a limited time for search, SPIRAL’s search engine has an integrated approach that combines the above search methods to find a fast implementation in the given time. Timed search takes advantage of the strengths of the different search methods, while limiting the search. It is completely configurable, so that a user can specify which search methods are used and how long they are allowed to run. SPIRAL provides reasonable defaults. These defaults first run a random search over a small number of ruletrees to find a reasonable implementation. Then it calls dynamic programming since this search method often finds good ruletrees in a relatively short time. Finally, if time remains, it calls STEER to search an even larger portion of the search space.

5.7 Integration and Implementation

All search methods above have been implemented in SPIRAL. As with the formula generator (Section 3), the search engine is implemented entirely in GAP. The search engine uses the ruletree representation of algorithms as its only interface to the formula generator. Thus, all search methods are immediately available when new transforms or new rules are added to SPIRAL.

6 Experimental Results

In this section, we present a number of experiments we have conducted using the SPIRAL system version 3.1 [22]. Table 2 shows the computing platforms we used for the experiments. In the remainder of this section we refer to the machines by the name given in the first column.

All timings for the transform implementations generated by SPIRAL are obtained by taking the mean value of a sufficiently large number of repeated evaluations. As C compiler options, we used “-O6 -fomit-frame-pointer -malign-double -fstrict-aliasing -mcpu=pentiumpro” for gcc, “-fast -xO5 -dalign” for cc, and “-G6 -O3” and “-G7 -O3” for the Intel compiler on the Pentium III and the Pentium 4, respectively.

The experimental data presented in this section, as well as in Section 2.3, illustrate the following key points: 1) The fully-expanded formulas in the algorithm space for DSP transforms have a wide range of performance and formulas with the best performance are rare (see Figure 2). 2) The best formula is platform dependent. The best formula on one machine is usually not the best formula on another machine. Search can be used to adapt the given platform by finding formulas well suited to that platform. This is shown

name	CPU and speed	RAM	OS	C compiler
athlon	Athlon, 1100 MHz	512 MB	Linux 2.2.16	gcc 2.95.2
p4linux	Pentium 4, 1400 MHz	1024 MB	Linux 2.4.2	gcc 2.95.2
p4win	Pentium 4, 1400 MHz	1024 MB	Win 2000	Intel 5.0
sun	UltraSparc II, 450 MHz	1536 MB	SunOS 5.7	cc 6 update 2 C 5.3
p3linux	Pentium III, 900 MHz	256 MB	Linux 2.2.17	gcc 2.91.66
p3win	Pentium III, 650 MHz	256 MB	Win 2000	Intel 5.0

Table 2: Computing platforms for experiments.

in Section 6.1. 3) Intelligent search can be used to find good formulas while only considering a small portion of the entire search space. This is shown in Section 6.2 by presenting the best runtimes found and the number of formulas considered for the DFT and $\mathrm{DCT}^{(\mathrm{II})}$ using different search methods. This is crucial due to the large number of possible formulas (see Table 1). 4) In addition to considering the space of formulas, the SPIRAL search engine can also search through alternative implementations. SPIRAL does this by searching over alternative SPL compiler directives and templates. An example is presented in Section 6.3 where SPIRAL searches also explores the degree of unrolling used by the SPL compiler. 5) The resulting performance produced by SPIRAL is very good. This is shown in Section 6.4 by comparing the implementations produced by SPIRAL for the DFT to FFTW [11], one of the best available FFT packages. In Section 6.5, the performance of a wide range of transforms is compared to that of the DFT. The results show that the performance that SPIRAL obtains is generally available to all DSP transforms and does not arise from special cases available only to the FFT.

6.1 Architecture Dependence

The best implementation of a given transform varies considerably between computing platforms. To illustrate this, we generated an adapted implementation of a $\mathrm{DFT}_{2^{20}}$ on four different platforms, using a dynamic programming search (Section 5.2). Then we timed these implementations on each of the other platforms. The results are displayed in Table 3. Each row corresponds to a timing platform; each column corresponds to a generated implementation. For example, the runtime of the implementation generated for p4linux, timed on athlon is in row 3 and column 2. As expected, the fastest runtime in each row is on the main diagonal. Furthermore, the other implementations in a given row perform significantly slower.

		fast implementation for			
		p3linux	p4linux	athlon	sun
timed on	p3linux	0.83	1.08	0.99	1.10
	p4linux	0.97	0.63	0.73	1.23
	athlon	1.23	1.23	1.07	1.22
	sun	0.95	1.67	1.42	0.82

Table 3: Comparing fast implementations of $\mathrm{DFT}_{2^{20}}$ generated for different machines. The runtimes are given in seconds.

6.2 Comparison of Search Methods

Figures 11 and 12 compare several search methods presented in Section 5. The considered transforms are DFT_{2^k}, for $k = 1, \ldots, 17$, and $\mathrm{DCT}^{(\mathrm{II})}_{2^k}$, for $k = 1, \ldots, 7$. The experiment is run on p3linux. The target language is C, and the default global unrolling of all nodes of size 2^5 and smaller is used, except for the timed search. Further, for the DFT of large size ($2^k > 32$) we consider only the Cooley-Tukey Rule (16) and its variant (18), excluding Rules (17), (19), and others, which perform poorly for large sizes. This considerably cuts down the DFT algorithm space.

There are two different types of plots shown in these figures. The upper plots, respectively, show the runtimes of the fastest implementation found by different search methods, divided by the runtimes for the best implementation found by 1-best dynamic programming (that is, the default dynamic programming that only keeps the single best formula for each transform and size). Thus, lower points correspond to faster formulas found. The bottom plots, respectively, show the number of implementations timed by the different search methods. Most of the search time is spent on timing the implementations; thus, the number of implementations timed is a good measure of the total time spent by the search algorithm. The y-axis is in logarithmic scale. The plots include an additional line indicating the total number of possible algorithms using the rules in SPIRAL. The timed search may time the same formula multiple times, since it calls various search methods, and thus, at small sizes, may time more implementations than possible algorithms. For dynamic programming, the number of implementations of smaller size timed is also included. We omitted random search since it always times 100 implementations.

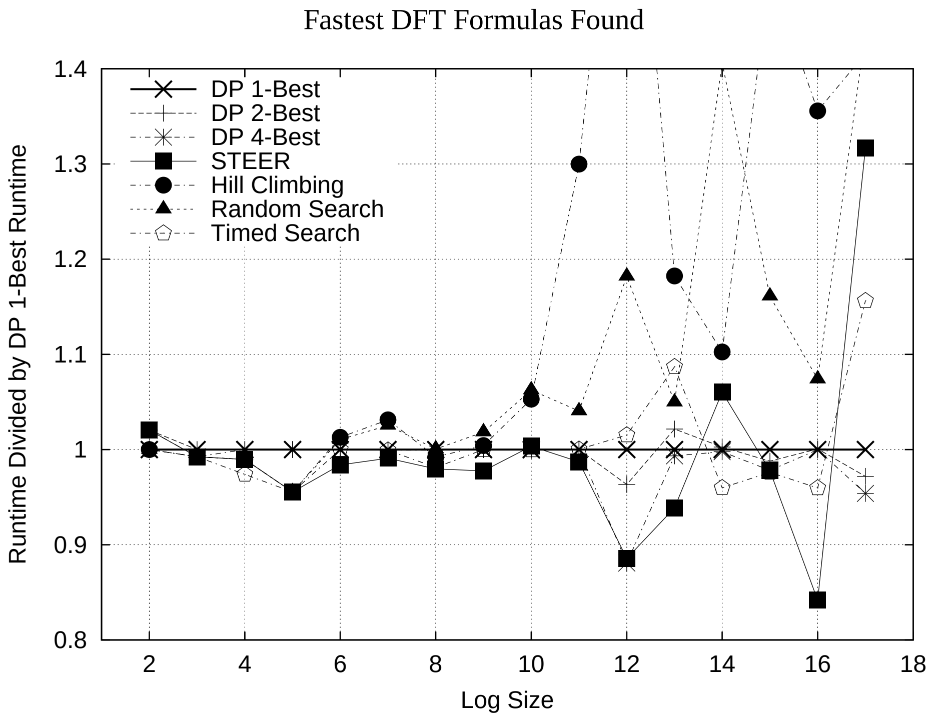
For the DFT, both random search and hill climbing search find rather slow formulas in comparison to the other search methods at many of the larger sizes. In several of these cases however, the algorithms found by random search are still within a reasonable 30–40% of the best. This is due to two reasons. First, as said above, we turned off some rules for large sizes that are known to perform poorly. Second, we do not draw uniformly from the algorithm space (see Section 5.3), which, in this case, also works in favor for the random search. (The majority of DFT ruletrees based on Rule (16) have a balanced top-level split, which leads to bad performance.)

It is surprising that hill climbing search performs so poorly, sometimes even considerably worse than random search. Hill climbing search initially generates fewer random ruletrees than random search; if these are poorly chosen, the search may not find mutations that produce good ruletrees. Thus, STEER has an important advantage over hill climbing in that it generates and uses a larger population; also, STEER has an advantage over random search in that it uses evolutionary operators to intelligently search for fast implementations.

For the DFT, there is no one search method that consistently finds faster formulas than the other methods. In fact, for the considered transforms and rules, plain 1-best dynamic programming does not perform much worse than the other methods; it is sometimes 10% slower than some of the other search methods. Generally, either STEER, timed search, or 4-best dynamic programming finds the fastest formula for a given DFT size in the experiments reported here.

For the $\mathrm{DCT}^{(\mathrm{II})}$, STEER finds faster formulas or equally fast formulas than all of the other search methods. For size 2^7, STEER is able to find a formula that is 20% faster than that found by 1-best dynamic programming.

These plots show that the number of possible formulas grows very quickly for very small sized transforms, forcing the search algorithms to only consider a very small portion of the space of formulas. For dynamic programming, it is clear that increasing the number of best formulas kept for each transform and size increases the number of formulas that must be timed. For small sizes, timed search usually times the most formulas as it calls several search algorithms. Since, in the presented experiments, timed search is only allowed 30 minutes, it begins to time slightly fewer formulas at larger sizes as it requires more time to run larger sized formulas. For larger sizes of the DFT, 4-best dynamic programming times the most formulas of all the search algorithms. For $\mathrm{DCT}^{(\mathrm{II})}$ sizes 2^5 to 2^7, STEER times the most formulas.



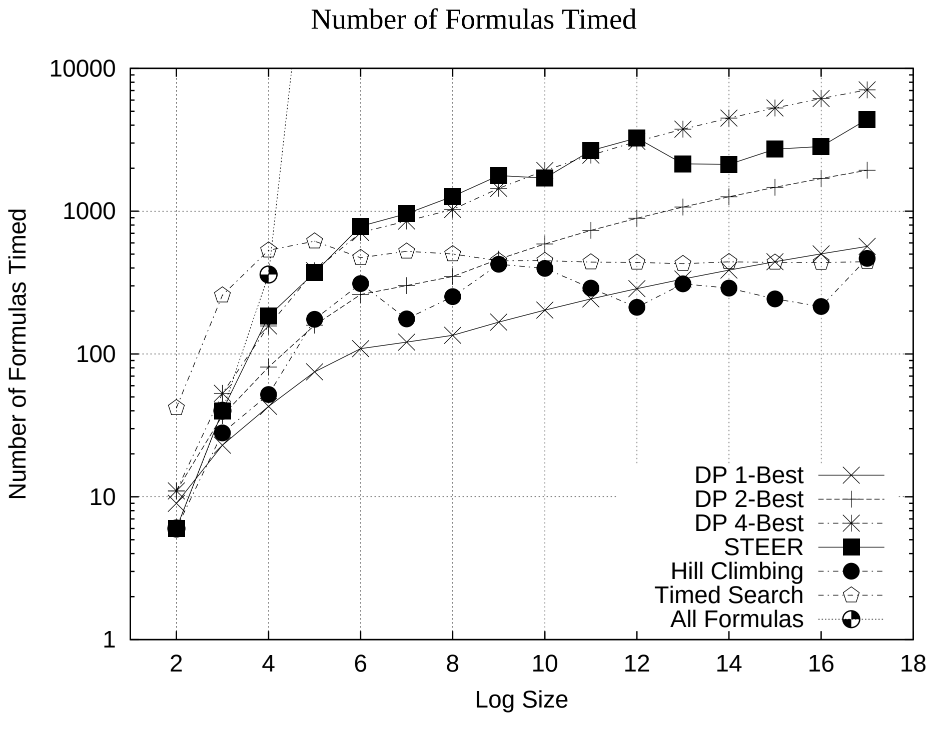
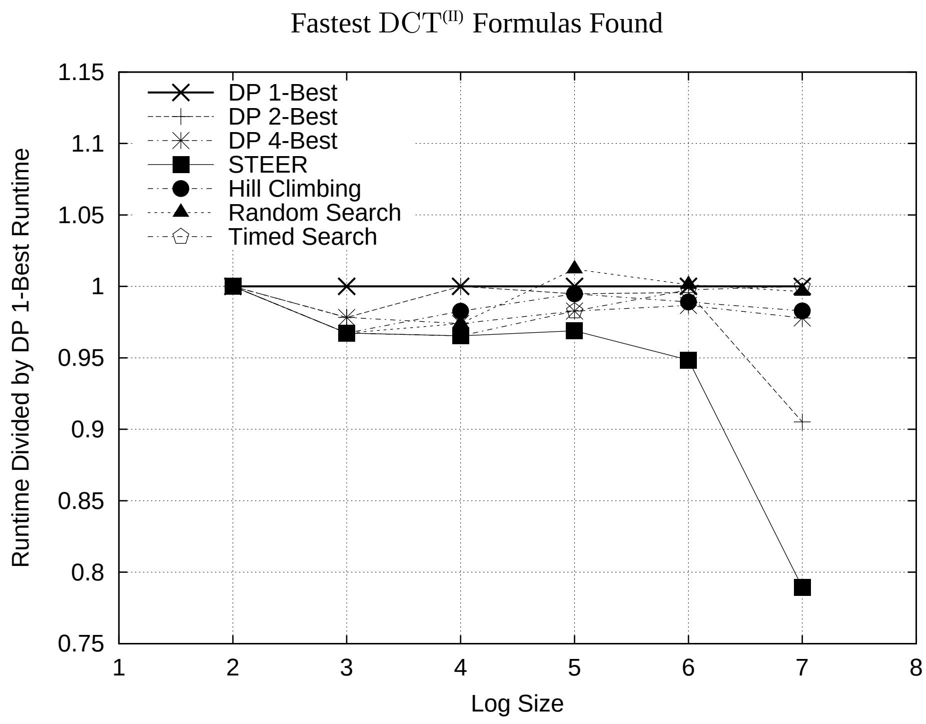


Figure 11: Comparison of search methods for the DFT on p3linux.



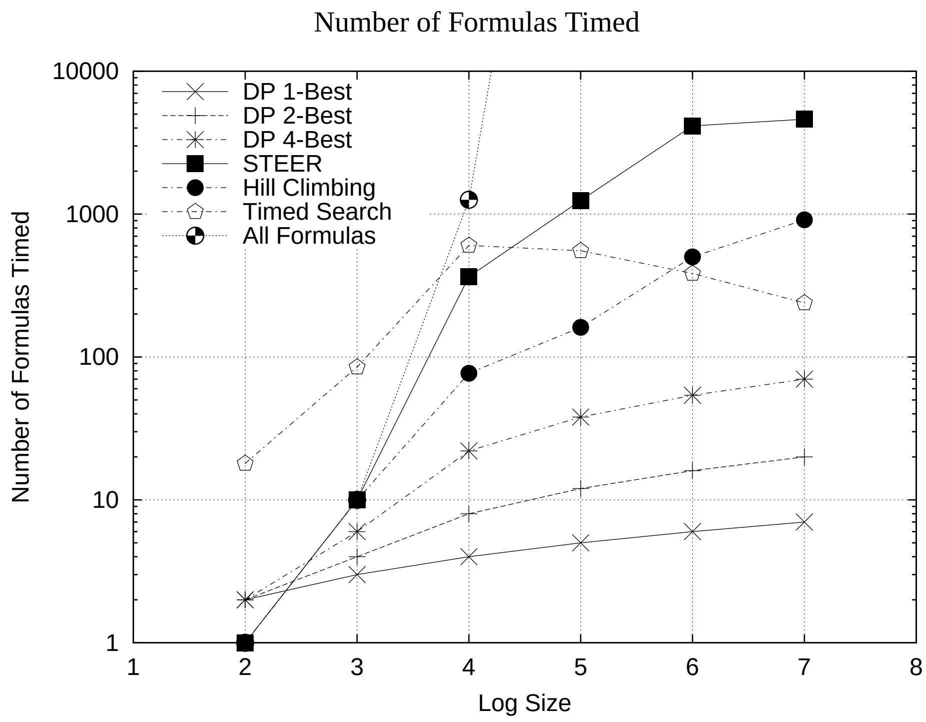


Figure 12: Comparison of search methods for DCT$^{(II)}$ on p3linux.

6.3 Searching over Local Unrolling Settings

Figure 13 compares several of the search methods when also searching over local unrolling settings for the DFT. STEER and 1-best and 4-best dynamic programming were run on the same machine as the experiments in Section 6.2; this time, the search methods also explored the possible local unrolling settings. Sections 4 and 5 describe how this is accomplished. The plots in Figure 13 compare these new runs against the previous runs that used a fixed global unrolling setting.

For many of the smaller sizes of the DFT, the searches that explored the possible local unrolling settings found faster implementations than those that used a fix global unrolling setting. The price paid is the number of timed formulas, which is much larger compared to searching with fixed global unrolling setting. Thus, if search time is not an issue, allowing a search over local unrolling settings can produce faster implementations.

6.4 Comparison with FFTW

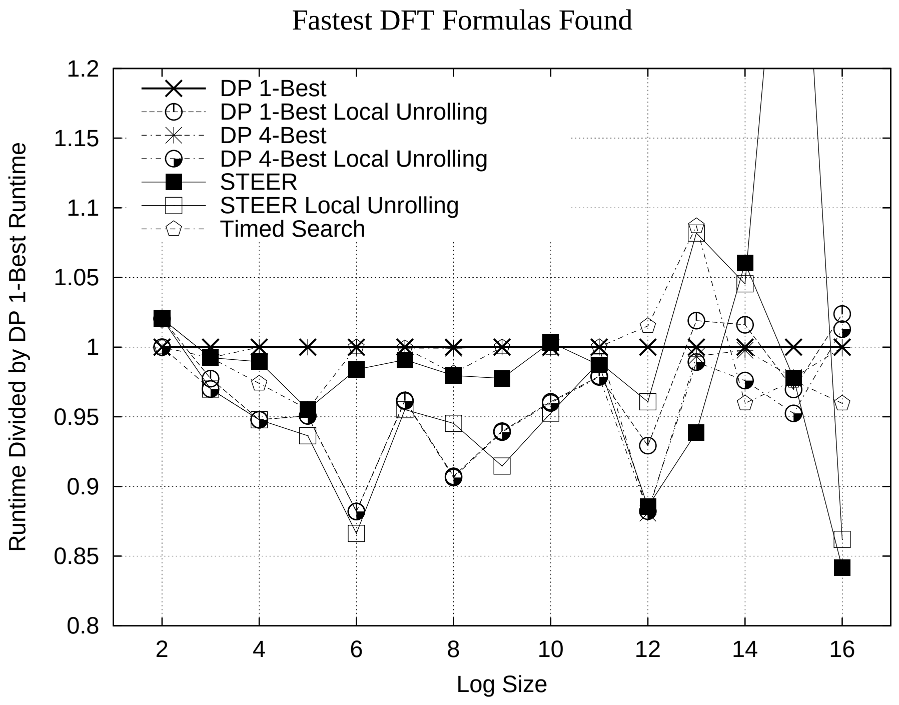
Figure 14 compares the performance of the best SPIRAL implementation of the DFT to FFTW 2.1.3 on all of the computing platforms considered for sizes 2^1 to 2^{22}. For other transforms highly tuned code is not readily available. We observe that for various machines and various sizes SPIRAL is faster than FFTW and vice versa. For small DFT sizes, FFTW code is slower due to its more complicated infrastructure. The same reason seems to favor SPIRAL code on p4win where the Intel vendor compiler can perform additional optimizations on SPIRAL's simple loop code. The main point in this comparison is not that one is faster than the other (without additional analysis and careful consideration of error in timing experiments a definitive conclusion can not be drawn presently), but rather that the runtimes produced by the general SPIRAL system are comparable across a range of platforms and transform sizes to FFTW.

6.5 Performance of Different Transforms

Figure 15 shows the performance of the best implementations of different transforms found by SPIRAL on the Sun machine. The transforms include different types of trigonometric transforms, the Walsh-Hadamard transform and the (rationalized) Haar transform. The performance of these transforms for sizes 2 to 64 is compared to the best time for the DFT of the same size. Since the performance of these transforms are comparable to that of the DFT and it was shown in Section 6.4 that the SPIRAL DFT times are comparable to the best available FFT packages, we can conclude that good performance is obtained across transforms. Figure 15 shows that the other transforms are faster than the DFT beyond size 4, with the WHT and RHT being the fastest. This is to be expected since these transforms operate on real data whereas the DFT operates on complex data. That the DFT is faster for sizes 2 and 4 may be due to special optimizations that have been applied to the DFT.

7 Conclusions

We presented the main components of SPIRAL, a system that automatically generates platform-adapted implementations of linear DSP transforms. At the core of SPIRAL is the representation of *algorithms* for transforms as mathematical *formulas* using a *few* constructs and primitives. This insight motivates the design of the language SPL, which is a domain-specific language for (linear) DSP algorithms. The SPL compiler translates algorithms, written in SPL, into C or Fortran procedures, and thus connects the mathematical realm of algorithms with the realm of concrete implementations. The space of algorithms for a given transform is very large, and yet generated from a *small* set of *rules*, which makes possible the automatic generation of these algorithms, and, furthermore, makes it easy to include new transforms. Since



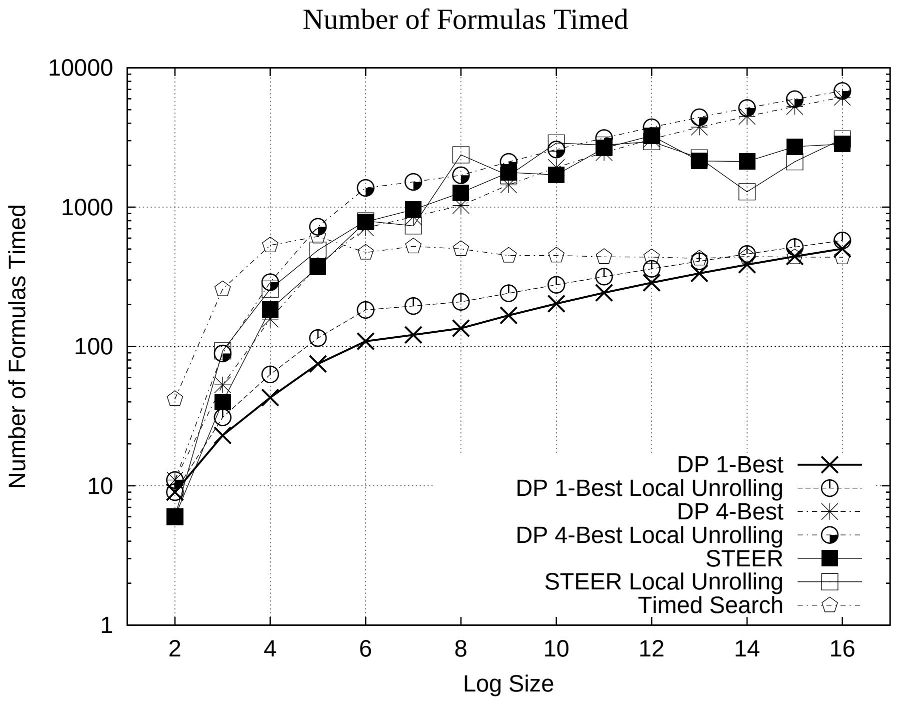


Figure 13: Comparison of search methods searching over local unrolling settings for the DFT on p3linux.

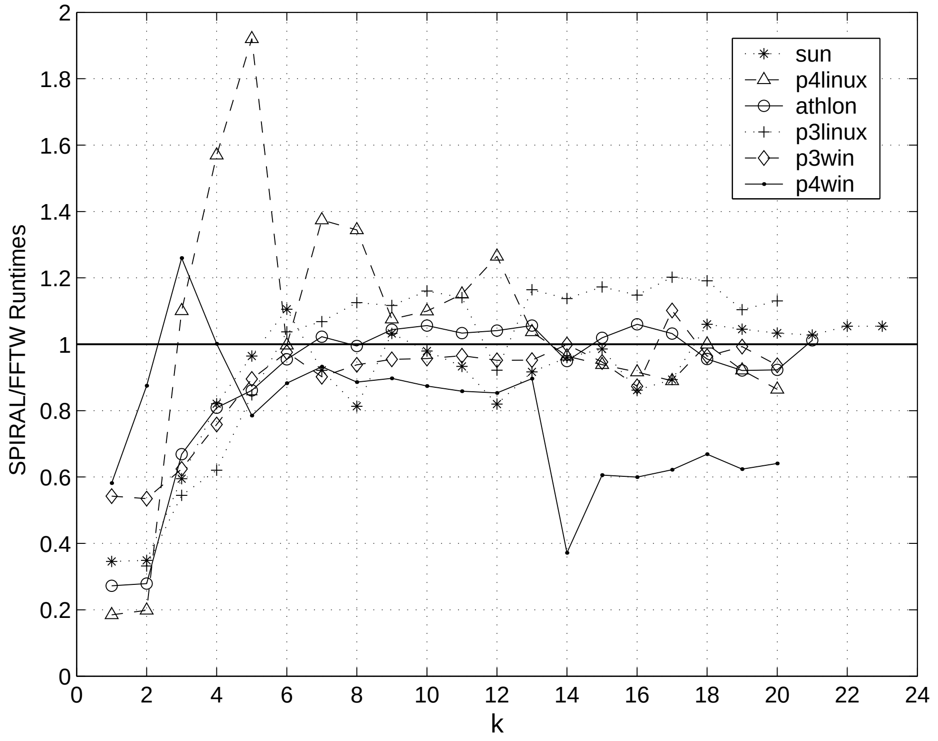


Figure 14: Comparison of the performance of SPIRAL and FFTW on different platforms.

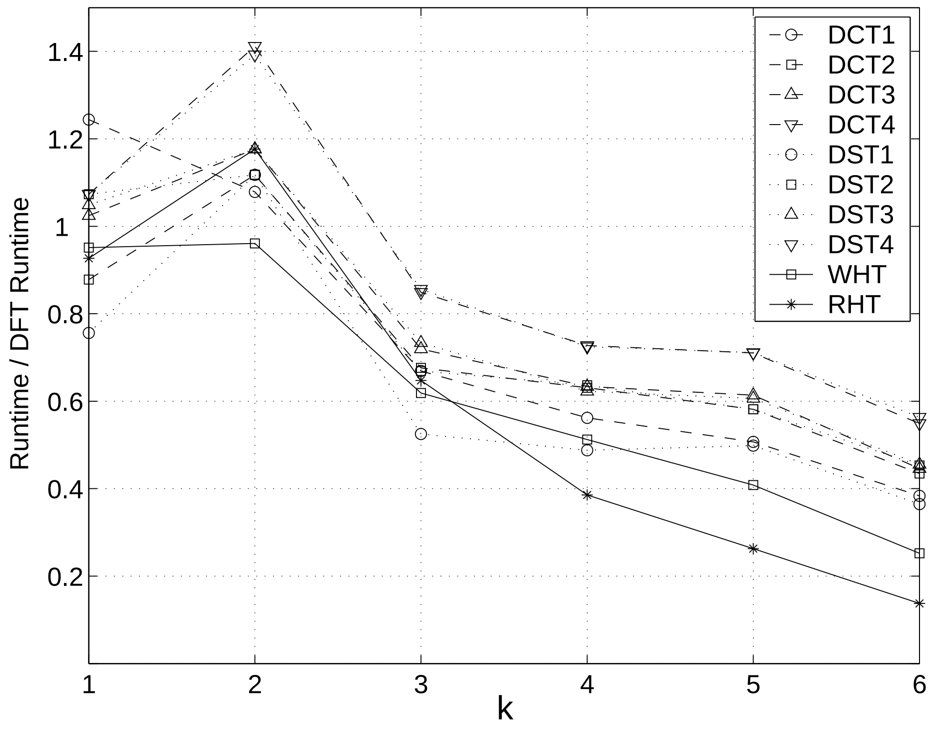


Figure 15: Runtime of various transforms of size 2^k, $k = 1, \ldots, 6$, divided by the runtime of DFT_{2^k} on sun.

algorithms can be automatically generated and implemented, SPIRAL is able to use *search* to find a "good match" of algorithm and platform. Thus, the code optimization is performed mainly at a "high," algorithmic, level. The distinguishing difference between algorithms generated for the same transform is the data flow (not the arithmetic cost), which, on modern architectures, has a strong impact on performance. SPIRAL's search directly addresses the problem of finding the best dataflow.

Since SPIRAL's methodology is based on a description of the algorithms, it is *not* transform specific, which distinguishes it from other approaches. Every algorithm that can be written using the provided constructs can be included. New constructs can be added if necessary. Different target languages or code types can be included by expanding the SPL compiler.

We believe that our approach has the potential to solve the problem of generating efficient DSP implementations across different DSP algorithms and different computing architectures.

References

[1] L. Auslander, J. Johnson, and R. W. Johnson. Automatic implementation of FFT algorithms. Technical Report DU-MCS-96-01, Dept. of Math. and Computer Science, Drexel University, Philadelphia, PA, 1996. Presented at the DARPA ACMP PI meeting.

[2] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proc. Supercomputing*. ACM SIGARC, 1997. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

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

[4] S. Egner. *Zur algorithmischen Zerlegungstheorie linearer Transformationen mit Symmetrie*. PhD thesis, Institut für Informatik, Univ. Karlsruhe, Germany, 1997.

[5] S. Egner and M. Püschel. *AREP – Constructive Representation Theory and Fast Signal Transforms*. GAP share package, 1998. `http://www.ece.cmu.edu/∼smart/arep/arep.html`.

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

[7] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber. Architecture Independent Short Vector FFTs. In *Proc. ICASSP*, volume 2, pages 1109–1112, 2001.

[8] F. Franchetti and M Püschel. A simd vectorizing compiler for digital signal processing algorithms. In *Proc. IPDPS*, pages 20–26, 2002.

[9] F. Franchetti and M Püschel. Short vector code generation for the discrete fourier transform. In *Proc. IPDPS*, pages 58–67, 2003.

[10] M. Frigo. A Fast Fourier Transform Compiler. In *Proc. PLDI*, pages 169–180, 1999.

[11] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP 98*, volume 3, pages 1381–1384, 1998. `http://www.fftw.org`.

[12] The GAP Team, University of St. Andrews, Scotland. *GAP – Groups, Algorithms, and Programming*, 1997. `http://www-gap.dcs.st-and.ac.uk/∼gap/`.

[13] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proc. Supercomputing*, 1999.

[14] A. Gačić, M. Püschel, and J. M. F. Moura. Fast Automatic Implementations of FIR Filters. In *Proc. ICASSP*, volume 2, pages 541–544, 2003.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[16] G. Haentjens. An investigation of recursive FFT implementations. Master's thesis, ECE Dept., Carnegie Mellon University, 2000.

[17] E.-J. Im and K. Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Proc. ICCS*, pages 127–136, 2001.

[18] H. W. Johnson and C. S. Burrus. The design of optimal DFT algorithms using dynamic programming. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-31:378–387, 1983.

[19] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *IEEE Trans. on Circuits and Systems*, 9, 1990.

[20] J. Johnson and M. Püschel. In Search for the Optimal Walsh-Hadamard Transform. In *Proceedings ICASSP*, volume IV, pages 3347–3350, 2000.

[21] D. Mirković and S. L. Johnsson. Automatic Performance Tuning in the UHFFT Library. In *Proc. ICCS*, LNCS 2073, pages 71–80. Springer, 2001.

[22] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso. SPIRAL: Portable Library of Optimized Signal Processing Algorithms, 1998. `http://www.spiral.net`.

[23] J. M. F. Moura and M. Püschel. SPIRAL: An Overview. In *Proc. Workshop on Optimizations for DSP and Embedded Systems*, 2003. Held in conjunction with CGO.

[24] N. Park, D. Kang, K. Bondalapati, and V. K. Prasanna. Dynamic Data Layouts for Cache-conscious Factorization of DFT. In *Proc. IPDPS*, pages 693–701, 2000.

[25] N. Park and V. K. Prasanna. Cache Conscious Walsh-Hadamard Transform. In *Proc. ICASSP*, volume 2, pages 1205–1208, 2001.

[26] K. R. Rao and J. J. Hwang. *Techniques & standards for image, video and audio coding*. Prentice Hall PTR, 1996.

[27] G. E. Révész. *Introduction to Formal Languages*. McGraw-Hill, 1983.

[28] D. Sepiashvili. Performance models and search methods for optimal FFT implementations. Master's thesis, ECE Dept., Carnegie Mellon University, 2000.

[29] B. Singer and M. Veloso. Stochastic search for signal processing algorithm optimization. In *Proc. Supercomputing*, 2001.

[30] B. Singer and M. M. Veloso. Automating the Modeling and Optimization of the Performance of Signal Transforms. *IEEE Trans. on Signal Processing*, 50(8):2003–2014, 2002.

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

[32] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. Siam, 1992.

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

[34] Z. Wang. Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-32(4):803–816, 1984.

[35] Z. Wang and B.R. Hunt. The Discrete W Transform . *Applied Mathematics and Computation*, 16:19–48, 1985.

[36] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing*, 1998. `http://math-atlas.sourceforge.net/`.

[37] J. Xiong. *Automatic Optimization of DSP Algorithms*. PhD thesis, Computer Science, University of Illinois at Urbana-Champaign, 2001. Also as Tech. Report UIUCDCS-R-2001-224, University of Illinois.

[38] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proc. PLDI*, pages 298–308, 2001.