SPL: A Language and Compiler for DSP Algorithms

Jianxin Xiong[1] Jeremy Johnson[2] Robert Johnson[3] David Padua[1]

[1] Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

{jxiong,padua}@cs.uiuc.edu

[2] Department of Mathematics and Computer Science
Drexel University
Philadelphia, PA 19104

jjohnson@mcs.drexel.edu

[3] MathStar Inc.
Minneapolis, MN 55402

rwj@mathstar.edu

ABSTRACT

We discuss the design and implementation of a compiler that translates formulas representing signal processing transforms into efficient C or Fortran programs. The formulas are represented in a language that we call SPL, an acronym from Signal Processing Language. The compiler is a component of the SPIRAL system which makes use of formula transformations and intelligent search strategies to automatically generate optimized digital signal processing (DSP) libraries. After a discussion of the translation and optimization techniques implemented in the compiler, we use SPL formulations of the fast Fourier transform (FFT) to evaluate the compiler. Our results show that SPIRAL, which can be used to implement many classes of algorithms, produces programs that perform as well as "hard-wired" systems like FFTW.

1. INTRODUCTION

Since the advent of digital signal processing, there has been an enormous effort to obtain high-performance implementations of signal processing algorithms such as the fast Fourier transform (FFT). This effort has produced thousands of variants of fundamental algorithms and an equally large number of implementation techniques. There have been more than 4000 papers written on the FFT alone [7] and undoubtedly an even greater number of implementations of this algorithm, many of which have been carefully hand-optimized. The cost of these hand optimizations and their long implementation times are a strong motivation to automate the implementation and optimization of signal processing algorithms [9]. To this end, we have designed SPL [8], a domain-specific language for describing and implementing fast signal transforms and related computations, and implemented a compiler that translates formulas written in this language into C or Fortran programs. SPL stands for Signal Processing Language.

The SPL compiler is a component of the SPIRAL system [13], that systematically searches through algorithm and implementation choices to find an optimal implementation for a given computing platform. Figure 1 shows the structure of the SPIRAL system. The algorithmic choices are expressed by mathematical formulas expressed in SPL, and the implementation choices are explored using the SPL compiler. The mathematical nature of SPL programs aids in the automatic generation of potential algorithms using a process called formula generation. The resulting implementations are compared using the performance evaluation component, which returns run times and other performance metrics obtained by executing the code in the target machine or estimated using models. The search engine looks for the fastest implementation out of the set of choices produced by the formula generator and SPL compiler. Due to the exponential size of the search space, intelligent search strategies are required to make this process feasible.

SPL is a descendent of the TPL (Tensor Product Language) [1] that was developed for the automatic generation of FFT algorithms. SPL programs are essentially mathematical formulas describing matrix factorizations. As such, they are built using operators from linear algebra and families of parameterized matrices. Such formulas naturally arise when describing fast signal transforms, where the signal transform corresponds to a matrix-vector product and fast algorithms can be represented by a factorization of the matrix into a product of sparse structured matrices [14, 15, 16].

The SPL compiler translates an SPL expression into a program to compute the matrix-vector product of the ma-

PLDI '01 Snowbird, Utah USA

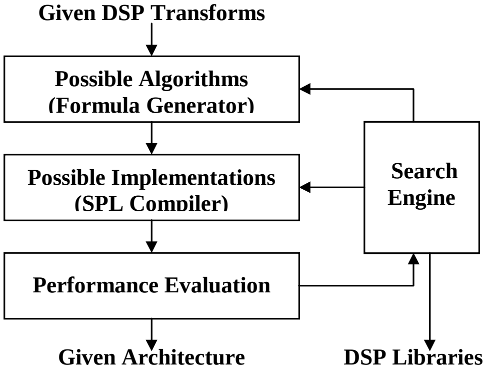


Figure 1: The SPIRAL framework

trix given by the SPL expression following a method first outlined in [10]. The semantics of the components of a SPL expression are defined using a template mechanism. This template mechanism allows the user to include additional operators and matrices in the SPL language. It also provides a mechanism to control the optimization and code generation strategies used by the compiler.

In this paper, we describe the translation process and optimization techniques used by the SPL compiler and present performance data for the code it generates. We use the FFT as a benchmark due to the availability of very high-performance implementations that can be used to measure the quality of the code produced by the SPL compiler. In particular, we show that the code produced is competitive with the best available software packages (FFTW [6] was used in our comparison) for performing similar computations.

Our system is more general than earlier systems, discussed in Section 5, that were hard-wired to generate only one class of signal processing programs. The use of SPL enables our system to generate any class of algorithm that can be represented as matrix expressions. Knowledge of particular matrix expressions is not included in the compiler, instead this knowledge is used by the formula generator when SPL programs are created and manipulated. As long as the algorithms can be described in SPL, all the steps needed to get a good implementation are automatic.

The rest of this paper is organized as follows. First, some background information about matrix factorizations and SPL are presented in Section 2. The organization and algorithms used by the compiler are presented in Section 3. Section 4 presents performance results. Finally, conclusions and future work are discussed in Section 5.

2. SPL, MATRIX FACTORIZATIONS, AND FAST SIGNAL TRANSFORMS

In this section we review the relationship between fast signal transforms and matrix factorizations, and we summarize how SPL can be used to represent matrix factorizations.

2.1 Matrix Factorizations and Fast Signal Transforms

A signal transform can be represented by the matrix-vector product, $y = Mx$, where the vector x denotes the input signal, the matrix M the transform, and y the transformed signal. Fast algorithms for computing $y = Mx$ can be obtained by factoring the matrix M into a product of sparse structured matrices. For example, a one-dimensional discrete Fourier transform (DFT) is defined as $y = F_n x$, where the (p,q) element of F_n is ω_n^{pq} with $\omega_n = e^{-\frac{2\pi i}{n}}$. The 4-point DFT

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

can be factored as

$$F_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -i \end{bmatrix}$$
$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Given the definitions

$$I_m \equiv \textit{the } m \times m \textit{ identity matrix},$$

$$W_m(\omega_n) \equiv \begin{bmatrix} 1 & & & \\ & \omega_n & & \\ & & \ddots & \\ & & & \omega_n^{m-1} \end{bmatrix}$$

(*can be written as* W_m *whenever it is clear in the context*),

$$L_s^{rs} \equiv \textit{the } rs \times rs \textit{ stride permutation matrix with stride } s,$$

the previous factorization can be written as

$$F_4 = \begin{bmatrix} I_2 & I_2 \\ I_2 & -I_2 \end{bmatrix} \begin{bmatrix} I_2 & \\ & W_2 \end{bmatrix} \begin{bmatrix} F_2 & \\ & F_2 \end{bmatrix} L_2^4.$$

In general, we have

$$F_n = \begin{bmatrix} I_{\frac{n}{2}} & I_{\frac{n}{2}} \\ I_{\frac{n}{2}} & -I_{\frac{n}{2}} \end{bmatrix} \begin{bmatrix} I_{\frac{n}{2}} & \\ & W_{\frac{n}{2}} \end{bmatrix} \begin{bmatrix} F_{\frac{n}{2}} & \\ & F_{\frac{n}{2}} \end{bmatrix} L_2^n. \quad (1)$$

and $y = F_n x$ can be computed in 4 steps:

$$\begin{aligned} \textit{step } (1): \quad t_1 &= L_2^n x \\ \textit{step } (2): \quad t_2 &= \begin{bmatrix} F_{\frac{n}{2}} & \\ & F_{\frac{n}{2}} \end{bmatrix} t_1 \\ \textit{step } (3): \quad t_3 &= \begin{bmatrix} I_{\frac{n}{2}} & \\ & W_{\frac{n}{2}} \end{bmatrix} t_2 \\ \textit{step } (4): \quad y &= \begin{bmatrix} I_{\frac{n}{2}} & I_{\frac{n}{2}} \\ I_{\frac{n}{2}} & -I_{\frac{n}{2}} \end{bmatrix} t_3 \end{aligned}$$

This four-step computation, applied recursively, is the well-known Cooley-Tukey FFT algorithm. Step (1) is a stride permutation that can be performed in n operations. Step (2) computes two FFTs of size $\frac{n}{2}$. Step (3) is the product of a diagonal matrix with a vector and therefore requires n operations. Step (4) requires 2 operations per row. The total number of operations, $f(n)$, is then $n + 2f(\frac{n}{2}) + n + 2n = f(\frac{n}{2}) + O(n)$, and, therefore, $f(n) = O(nlogn)$.

The previous factorizations can be represented more concisely using the tensor product. Let A be an $m \times m$ matrix and B be an $n \times n$ matrix. The *tensor product* of A and B, denoted as $A \otimes B$, is the following $mn \times mn$ matrix:

$$A \otimes B \;\equiv \left[\begin{array}{ccc} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mm}B \end{array} \right]. \qquad (2)$$

The tensor product has interesting interpretations when one of the operands is the identity matrix. Thus,

$$I_m \otimes A = \left[\begin{array}{ccc} A & & \\ & \ddots & \\ & & A \end{array} \right]$$

represents an identical transformation applied to successive sub-vectors of the input vector, and

$$A \otimes I_m = \left[\begin{array}{ccc} a_{11}I_m & \cdots & a_{1n}I_m \\ \vdots & \ddots & \vdots \\ a_{n1}I_m & \cdots & a_{nn}I_m \end{array} \right]$$

represents the same transformation applied to strided sub-vectors of the input vector with stride m.

The FFT factorization expressed in Equation 1 can be represented using the tensor product notation as follows:

$$F_n = (F_2 \otimes I_{\frac{n}{2}})T^n_{\frac{n}{2}}(I_2 \otimes F_{\frac{n}{2}})L^n_2, \qquad (3)$$

where $T^n_{\frac{n}{2}} = \left[\begin{array}{cc} I_{\frac{n}{2}} & \\ & W_{\frac{n}{2}} \end{array} \right]$, is known as the *twiddle matrix.* The general definition of twiddle matrix is:

$$T^{mn}_m = \left[\begin{array}{cccc} I_m & & & \\ & W_m & & \\ & & \ddots & \\ & & & W^{n-1}_m \end{array} \right]. \qquad (4)$$

A more general form of the previous factorization, that does not require n to be even, is:

$$F_{rs} = (F_r \otimes I_s)T^{rs}_s(I_r \otimes F_s)L^{rs}_r. \qquad (5)$$

This is also called *decimation in time FFT.*

Other factorizations are possible by using the identity:

$$A_{m\times m} \otimes B_{n\times n} = L^{mn}_m(B_{n\times n} \otimes A_{m\times m})L^{mn}_n. \qquad (6)$$

Examples of such factorizations include, *decimation in frequency FFT*:

$$F_{rs} = L^{rs}_r(I_r \otimes F_s)T^{rs}_s(F_r \otimes I_s), \qquad (7)$$

a parallel form of the FFT ($I_m \otimes A$ can be trivially implemented as a parallel loop):

$$F_{rs} = L^{rs}_r(I_r \otimes F_s)L^{rs}_s T^{rs}_s(I_r \otimes F_s)L^{rs}_r, \qquad (8)$$

and a vector form:

$$F_{rs} = (F_r \otimes I_s)T^{rs}_s L^{rs}_r(F_s \otimes I_r). \qquad (9)$$

Finally, the following identity further generalizes the previous factorizations of F_n:

$$\begin{aligned} F_n = \;\; & \textstyle\prod_{i=1}^{t} \left[(I_{n_{i-}} \otimes F_{n_i} \otimes I_{n_{i+}})(I_{n_{i-}} \otimes T^{n_i n_{i+}}_{n_{i+}})\right] \cdot \\ & \textstyle\prod_{i=t}^{1}(I_{n_{i-}} \otimes L^{n_i n_{i+}}_{n_i}), \end{aligned} \qquad (10)$$

where $n = n_1 \cdots n_t$, $n_{i-} = n_1 \cdots n_{i-1}$, and $n_{i+} = n_{i+1} \cdots n_t$. Proof of the above factorization equations can be found in [10].

Applying this last factorization recursively to each F_{n_i} until the base case is equal to F_2 leads to an algorithm for computing F_n. The special case when $t = 2$, $n_1 = 2$, and $n_2 = \frac{n}{2}$ leads to the standard recursive FFT. Other recursive breakdown strategies are obtained by choosing different values for n_1 and n_2 or by choosing other factorizations such as those shown above. The special case when $n_1 = \cdots\ n_t = 2$ leads to the standard iterative, radix-two, FFT. Equation 10 allows us to explore various amounts of recursion and iteration along with different breakdown strategies.

Other transforms can be factored in a similar manner. These include the *Walsh-Hadamard transform (WHT)*

$$\begin{array}{lcl} WHT_2 & = & F_2, \\ WHT_{2^n} & = & \prod_{i=1}^{t}(I_{2^{n_1}\cdots 2^{n_{i-1}}} \otimes WHT_{2^{n_i}} \otimes I_{2^{n_{i+1}}\cdots 2^{n_t}}), \end{array}$$

and the *discrete cosine transform (DCT)*

$$\begin{array}{lcl} DCTII_2 & = & diag(1, 1/\sqrt{2}) \cdot F_2, \\ DCTII_n & = & P \cdot (DCTII_{\frac{n}{2}} \oplus DCTIV_{\frac{n}{2}}) \cdot (I_{\frac{n}{2}} \otimes F_2)^Q, \\ DCTIV_n & = & S \cdot DCTII_n \cdot D. \end{array}$$

In this last set of equations, P,Q are permutation matrices, D, S are diagonal matrices, $\oplus$ is the direct sum, and A^Q means the conjugation $Q^{-1}AQ$.

2.2 The SPL Language

SPL is a convenient language for describing matrix factorizations, and hence fast algorithms for computing matrix-vector products. In particular, it can be used for describing fast signal transforms such as the FFT. SPL expressions are matrix expressions involving general matrices and parameterized matrices such as F_8 and L^{32}_8. An SPL expression may involve a variety of operations including composition, direct sum, and tensor product. SPL represents matrix operations in Cambridge Polish notation. For example, `(compose A B)` represents the matrix product $A \cdot B$. A matrix can be specified by giving the value of all its elements, or it can be represented by a parameterized matrix. For example, a 2×2 identity matrix can be represented as `(matrix (1 0) (0 1))`, `(diagonal (1 1))` or `(I 2)`.

An SPL program contains one or more SPL expressions (or formulas). Each expression can be optionally preceded by definitions and compiler directives. An SPL formula can be interpreted as a matrix or, more importantly for our purposes, as a subroutine that accepts a vector as the input and output the result of the corresponding matrix-vector product. In this section, we briefly review the syntax used by SPL. Further details are available in [8].

Typical components of SPL expressions are:

```
(1) General matrices, for example:
    (matrix ((a11 ... a1n) ... (am1 ... amn))
    (diagonal (a11 ... ann))
    (permutation (k1 ... kn))

(2) Parameterized matrices, for example:
    (I n)     ; identity matrix
    (F n)     ; Fourier transform by definition
    (L mn n)  ; stride permutation matrix
    (T mn n)  ; twiddle matrix
```

```
(3) Matrix operations, for example:
    (compose A1 ... An)    ; matrix product
    (tensor A1 ... An)     ; tensor product
    (direct-sum A1 ... An) ; direct sum
```

The elements of a matrix can be real or complex numbers. In SPL, these numbers can be specified as constant scalar expressions, which may contain function invocations and symbolic constants like `pi`. Thus, expressions like `12`, `1.23`, `sqrt(2)`, and `(cos(2*pi/3.0),sin(2*pi/3.0))` are valid scalar SPL expressions. All constant scalar expressions are evaluated at compile-time.

An SPL program may contain templates, a mechanism to add new parameterized matrices and matrix operations:

```
(4) Template definition
    (template pattern condition i-code-list)
```

We'll discuss templates in Section 3.

In SPL, a formula can be assigned a name so that it can be reused elsewhere.

```
(5) Name assignment
    (define name formula)
```

Lines starting with "`#`" are compiler directives, and the contents between a "`;`" and the new line are comments.

Compiler directives control some of the compiler actions. For example, an SPL formula can be preceded by a `#subname` directive to specify the name of the subroutine to be generated. Other directives include: `#unroll`, which controls the loop unrolling; `#datatype`, which specifies the type (real or complex) of the data to be manipulated; `#language`, that specifies the target language; and `#codetype`, that specifies whether the intrinsic type of the input vector and intermediate operands will be real or complex when the target language is Fortran. When the `#datatype` is complex and the `#codetype` is real, the complex values will be represented as pairs of real numbers and, for each complex operation, the compiler will generate the corresponding real operations.

To illustrate SPL, consider the formula:

$$F_{16} = (F_4 \otimes I_4) T_4^{16} (I_4 \otimes F_4) L_4^{16}$$

where

$$F_4 = (F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4$$

The SPL program corresponding to this formula is

```
(define F4
    (compose
        (tensor (F 2) (I 2)) (T 4 2)
        (tensor (I 2)(F 2)) (L 4 2)))
#subname fft16
(compose
        (tensor F4 (I 4)) (T 16 4)
        (tensor (I 4) F4) (L 16 4))
```

3. THE COMPILER

The SPL compiler generates a Fortran or C subroutine for each SPL formula. The compiler proceeds in five phases: parsing, intermediate code generation, intermediate code restructuring, optimization, and target code generation. Each of these phases is described below in separate subsections.

3.1 Parsing

The parser translates each SPL formula into an abstract syntax tree (AST) containing matrix operations and matrices. The AST is binary. N-ary formulas such as `(compose A1 ... An)` are associated right-to-left. Nested expressions must be used in order to specify a different association order.

3.2 Intermediate Code Generation

All SPL operations have to be defined using templates. Templates for pre-defined operations, such as those corresponding to the parameterize matrices and matrix operations mentioned in Section 2.2, are placed in a start-up file that is read by the compiler before reading the source SPL program. Only when the formula matches a template does the compiler know the meaning of the formula and can generate code for it. This approach provides great flexibility and extensibility. New parameterized matrices and new matrix operations can be easily added using templates.

A template consists of: a *pattern*, a *condition*, and an intermediate code (or *i-code*) sequence. If an SPL formula matches the pattern and satisfies the condition, then the SPL formula can be translated into the intermediate code specified by the template. Matching is attempted in the reverse order of definition so that new templates override earlier ones and in particular override pre-defined templates that are processed by the SPL compiler as if they were defined at the beginning of the program.

The pattern is an SPL formula that can contain *pattern variables*. All pattern variables end with an underscore (_). Pattern variables that start with a lower-case letter can match any integer constant and those that start with an upper-case letter can only match SPL formulas. For example, the pattern `(I n_)` matches `(I 1)` or `(I 2)`; and the pattern `(compose X_ Y_)` matches `(compose (F 2)(I 3))` or `(compose (compose A B) (tensor (I 2) C))`, where `A`, `B` and `C` are defined symbols representing formulas. Pattern variables can not match undefined symbols. For example, although the pattern `(I n)` can match `(I 2)`, `(I 4)`, and so on, it can not match a generic `(I m)`. This limitation is consistent with the fact that SPL is designed to describe fixed-size computations.

The condition of a template is a C-style boolean expression, enclosed by brackets. For instance, a pattern `(L m_ n_)` with condition `[ m_==2*n_ ]` will match the formula `(L 4 2)`, but not `(L 4 1)`. Here `m_` and `n_` matches 4 and 2, respectively. The condition is optional.

I-code instructions are Fortran-style "`do`" loop headers, "`end do`" statements, or four-tuples containing an operator and up to three operands. Operators are mainly arithmetic operations, such as `+`, `-`, `*`, `/` and assignment. Operands can be constants, pattern variables, scalar variables, vector variables, or *intrinsic functions*. If a pattern variable matches an SPL formula, properties of the formula represented as "components" of the pattern variable can be used as scalar values. For example, if `A_` is a pattern variable, then `A_.in_size` is the size of the input vector of the formula represented by `A_`. Similarly, `A_.out_size` is the size of the output vector. Scalar variables can be loop indices (named `$i0`, `$i1`, ...), integer variables (named `$r0`, `$r1`, ...), and floating point or complex variables (named `$f0`, `$f1`, ...). Vector variables can be the input vector `$in`, output vector `$out`, or temporary vectors (named `$t0`, `$t1`, ...). The subscripts of vector variables are always linear combinations of loop indices with integer coefficients. Intrinsic functions are parameterized scalar function. For example, W(n,k) is an intrinsic function which returns the value ω_n^k. Here n and k can be scalar constants or variables.

An example template is the following definition of the semantics of (F n_):

```
(template (F n_) [n_>0]
 (do $i0 = 0,n_-1
    $out($i0) = 0
    do $i1 = 0,n_-1
      $r0 = $i0 * $i1
      $f0 = W(n_ $r0) * $in($i1)
      $out($i0) = $out($i0) + $f0
    end
  end))
```

Each template has six implicit parameters: $in, $out, $in_stride, $out_stride, $in_offset and $out_offset, which represent the input vector, the output vector, and the strides and offsets to access each of these vectors (these values were assumed to be 1 in this example to make the code easier to understand). Furthermore, the size of the input and output vectors, $in_size and $out_size respectively, is inferred by the SPL compiler from the template.

Templates can be applied recursively. For example, the following template defines the compose operation.

```
(template (compose A_ B_))
   [ A_.in_size == B_.out_size ]
   ( B_( $in, $t0, 0, 0, 1, 1 )
     A_( $t0, $out, 0, 0, 1, 1 )))
```

Here, pattern variables A_ and B_ are followed by a list of parameters to be used during expansion of the formula represented by the pattern variable. The formula (compose (I 2)(F 2)) will match this template with A_=(I 2) and B_=(F 2). The i-code sequence contains pattern variables, so the compiler tries to find templates which match the sub-formulas (I 2) and (F 2), replacing the two pattern variables and their parameters with the code generated from the sub-formulas. The first parameter of the pattern variables, $in in the first instruction and $t0 in the second, will be used as the value of $in in the matching template. The second parameter will be used as the value of $out. The other four parameters correspond to the offsets and strides for the input and output vectors. In this case, the offsets are always zero, and the strides are always one.

The pattern can be as elaborate as any SPL formula. Thus, code generation strategies can be defined for all formula patterns regardless of their complexity. By using the appropriate template, it is possible to reproduce the effect of some compiler optimizations.

For example, the SPL formula (compose (tensor (I 8) A) (tensor (I 8) B)) could be translated into two consecutive loops using a tensor template (twice) and a compose template. To merge these two loops into one, we can define a template which recognizes the complete formula and generates a single loop. The effect is the same as loop fusion.

Templates can be generated by a search engine. Thus, in a system such as SPIRAL, the search can include different formulas, different code generation strategies, and different optimization parameters in a uniform way.

3.3 Intermediate Code Restructuring

We discuss next three transformations that can be applied to the i-code after it is generated by the template mechanism: loop unrolling, intrinsic function evaluation, and type transformation.

3.3.1 Loop unrolling

The presence of pattern variables in the templates may force the use of loops in the i-code sequence. Thus, although a loop-free i-code sequence is possible when the pattern represent a constant-size parameterized matrix such as (F 2), loops are needed for patterns containing a variable-size term such as (F n_). A template for (F n_) was presented in Section 3.2.

However, after matching the pattern with a specific SPL formula, these loop bounds become constant values. 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 loop control overhead eliminated but it also becomes possible to substitute scalar variables 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 even if the subscripts are constants. The downside 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, with the command-line option "-B 32", all the loops in those sub-formulas whose input vector is smaller than or equal to 32 are fully unrolled. To see how to control loop unrolling for individual formulas, consider the following SPL program:

```
#datatype real
#unroll on
(define I2F2 (tensor (I 2)(F 2)))
#unroll off
#subname I64F2
(tensor (I 32) I2F2)
```

It generates the following Fortran sequence where the unrolled version of I2F2 appears as the body of the loop:

```
subroutine I64F2(y,x)
implicit real*8(f)
implicit integer(r)
real*8 y(128),x(128)
do i0 = 0, 31
    y(4*i0+1) =  x(4*i0+1) + x(4*i0+2)
    y(4*i0+2) =  x(4*i0+1) - x(4*i0+2)
    y(4*i0+3) =  x(4*i0+3) + x(4*i0+4)
    y(4*i0+4) =  x(4*i0+3) - x(4*i0+4)
end do
```

In the generated code, the input and output vector is named as x and y, respectively.

3.3.2 Intrinsic function evaluation

All intrinsic functions are evaluated at compile-time. If all the parameters of an intrinsic function are constant, the intrinsic function invocation is replaced by its value. If one or more of the parameters are loop indices and the others are constant, then the compiler evaluates the intrinsic function for all possible values of the loop indices, places these values in a table, and replaces the intrinsic function invocation with a reference to the table accessed through the loop indices.

Model	Ultra5	Origin 200	PC
CPU	333MHz UltraSPARC IIi	180MHz MIPS R10000	400MHz Pentium II
L1 cache	16KB/16KB	32KB/32KB	16KB/16KB
L2 cache	2MB	1MB	512KB
Memory	128MB	384MB	256MB
OS	Solaris 7	IRIX64 6.5	Linux kernel 2.2.18
Compiler	Sun Workshop 5.0	MIPSpro 7.3.1.1m	egcs 1.1.2

Table 1: Experiment platforms

3.3.3 *Type transformation*

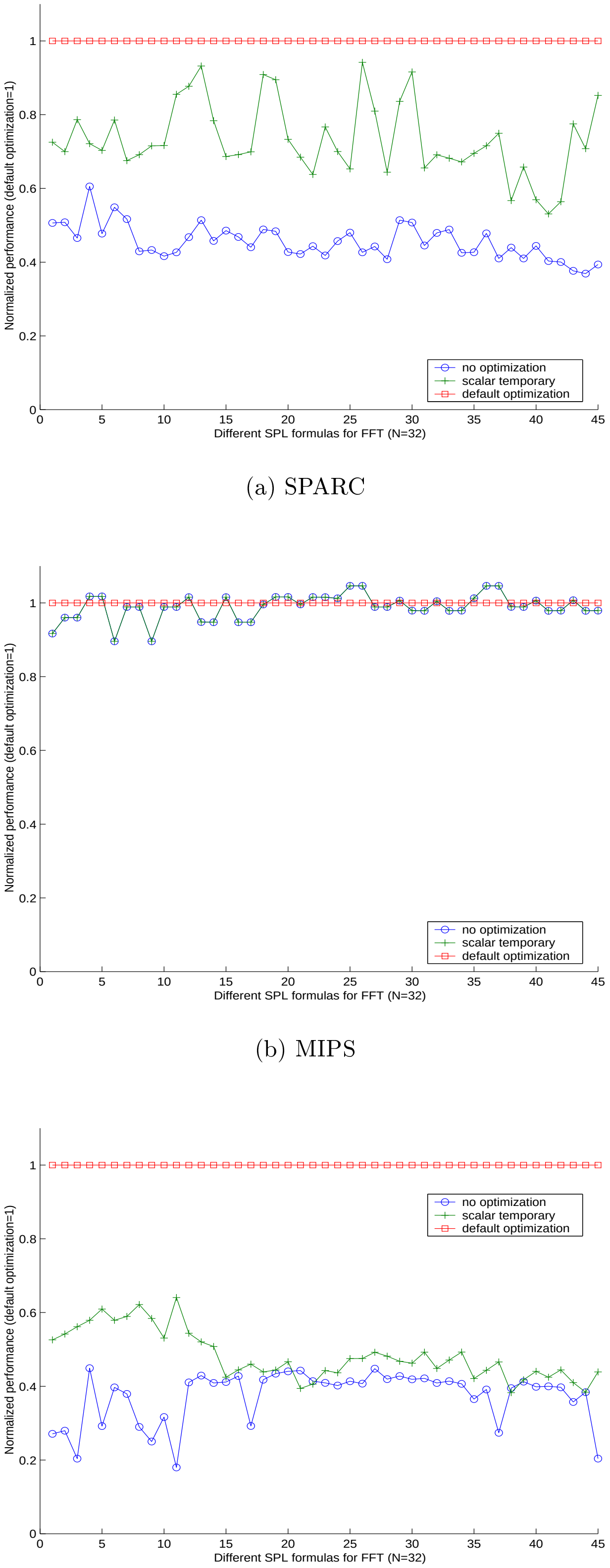
The input and output vectors of the subroutine generated by the SPL compiler could be either real or complex. If the type is complex and the target language is Fortran, the compiler could generate either complex intrinsic types or a pair of real numbers for each occurrence of a complex scalar. The advantage of using complex type is that the resulting code is shorter and clearer. However, of the popular imperative languages only Fortran supports complex data type. Furthermore, in our experiments we decided to use only Fortran codes based on real numbers because this enabled further optimizations, such as the replacement of multiplication by i (the square root of -1) with a swap instruction followed by a negation instruction.

3.4 Compiler Optimizations

The SPL compiler applies constant folding, copy propagation, common subexpression elimination, and dead code elimination. These *default optimizations* are applied in a single pass using a value numbering algorithm. Both scalar variables and array elements are handled by the optimizations. Performing conventional optimizations in the SPL compiler was necessary to improve the quality of the code generated by the native Fortran and C compilers we used in our experiments.

To demonstrate the effect of the optimizations applied by the SPL compiler, we selected a set of 45 SPL formulas, and generated three versions of Fortran code for each of them. Figure 2 shows the normalized performance measured on the three platforms listed in Table 1. The three versions of Fortran code are: (1) no optimization; (2) replacing temporary vectors with scalar variables; (3) default optimization. All these versions are compiled by the corresponding back-end compiler with the maximum optimization turned on. The performance data is obtained by taking the inverse of the execution time, and then normalizing with respect to the performance of the version with default optimization.

The effect of default optimizations depend on the platform and the back-end compiler. In the SPARC machine (Figure 2), replacing array elements with scalar variables improved the performs by 60 percent, the default optimizations introduced similar amount of improvement. On the Pentium II machine, changing array elements to scalar variables did not help much, while the default optimizations improved the performance by a factor of two. On the MIPS machine, however, the effect of these optimizations was insignificant. It means the MIPSpro compiler did a good job in standard optimizations.



(a) SPARC

(b) MIPS

(c) Pentium II

Figure 2: Effect of basic optimization

An additional effect of these optimizations is that they improve the readability of the target code by avoiding much of the redundant code that would be generated otherwise.

The compiler also applies two, machine-dependent, peephole optimizations. One replaces unary minus operators on double precision operands with subtraction operations or negative constants. For example, the compiler generates "`f2=0-f1`" instead of "`f2=-f1`", and "`f2=(-7)*f1`" instead of "`f2=-7*f1`". The reason is that, on SPARC systems, arithmetic negation is a single precision instruction and it takes at least six cycles for the floating point unit to switch between single precision mode and double precision mode. Another optimization declares all temporary variables as "`automatic`" so they will be allocated on the stack. Many Fortran compiler, by default, regards all variables as static. This transformation led to significant performance improvements on SPARC systems. Both of these transformations are machine-specific and may not have a positive effect on machines other than the SPARC.

3.5 Target Code Generation

In addition to generating code in different target languages, the SPL compiler can add stride and offset information to the input and output vectors, so that the computation can be performed and the result stored in vector elements that are not consecutive and do not start at the beginning of the vector. For example, if the input stride is 2, the output stride is 4 and both offsets are 1, then the code generated for `(I 2)` will copy x(1), x(3) to y(1), y(5) (suppose subscripts start from 0). The compiler also can vectorize the resulting code by adding an outer loop to the the code so the computation changes from A to $A \otimes I_m$, where m is a parameter and A is a formula.

4. EXPERIMENTS

To evaluate the SPL compiler, we present results on code generation of the FFT transforms F_{2^i}, with i between 1 and 20. We used a simple search strategy in these experiments. More elaborate strategies are possible and they may produce codes with better performance than the codes tabulated in this paper. However, the simple strategy we used was sufficient to match the performance of the codes generated by FFTW and demonstrate the viability of this approach.

Our search strategy proceeded by first searching for a good implementation for small-size transforms, 2 to 64, and then searching for a good implementation for larger size transforms that use the small-size results as basic computation modules. For the small sizes, we used dynamic programming over all possible factorizations using Equation 10 (Section 2.1) and, for each size, we selected the factorization with the lowest execution time. For larger sizes, we also used dynamic programming over the factorization obtained by using the equation

$$F_{rs} = (F_r \otimes I_s) T_s^{rs} (I_r \otimes F_s) L_r^{rs}$$

with the restriction that $r \leq 64$. The formula was applied recursively to F_s until $s \leq 64$. In the case of large-size FFTs the dynamic programming algorithm kept the three best results at each stage instead of just one as is usually done. To generate code for large size FFTs, templates for F_r, $2 \leq r \leq 64$, were created using for the intermediate code component the outcome of the search for small-size transforms. The reason we chose this strategy is that it parallels the strategy followed by FFTW.

The experiments were carried out on the same platforms as described in Table 1. In the rest of this section, we first present the performance results for small-size transforms, then the results for larger sizes. In each case, we compare performance with FFTW.

4.1 Small Size FFTs

For small-size problems, after some trial and error, we found that it is usually better to generate straight-line code because loop control overhead is eliminated and all the temporary variables can be scalars. Using scalars enables the back-end compiler to do a better optimization job. Furthermore, for small-size transforms, code size does not affect performance.

For a given FFT size, the number of operations generated by the compiler is fixed, and the performance depends exclusively on factors such as register allocation, memory access pattern, and instruction scheduling. These factors are influenced by the order in which the instructions appear in the source program. We found this order to be an important factor despite the fact that optimizing compilers are supposed to do instruction reordering. In the experiments reported here, the different computation orders were exclusively the result of using different formulas. For example, the following two formulas are different factorizations of F_8:

```
; common definition
(define F4
    (compose (tensor (F 2)(I 2) (T 4 2)
           (tensor (I 2)(F 2)) (L 4 2))))

; formula-1
(compose (tensor (F 2)(I 4)) (T 8 4)
       (tensor (I 2) F4) (L 8 2))

; formula-2
(compose (tensor F4 (I 2)) (T 8 2)
       (tensor (I 4)(F 2)) (L 8 4))
```

The corresponding Fortran code using complex arithmetics perform the same computations but in different order:

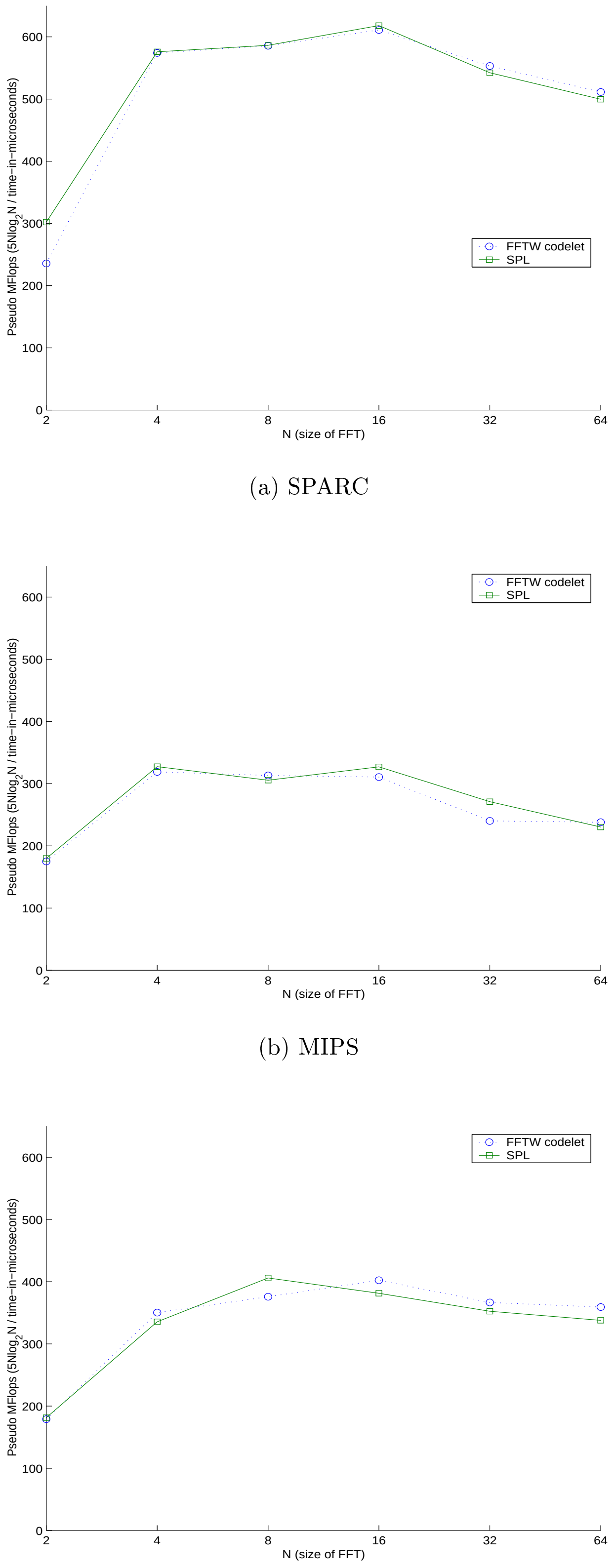
```
; formula-1              ; formula-2
f0 = x(1) + x(5)         f0 = x(1) + x(5)
f1 = x(1) - x(5)         f1 = x(1) - x(5)
f2 = x(3) + x(7)         f2 = x(2) + x(6)
f3 = x(3) - x(7)         f3 = x(2) - x(6)
f4 = (0,-1)*f3           f4 = x(3) + x(7)
f5 = f0 + f2             f5 = x(3) - x(7)
f6 = f0 - f2             f6 = x(4) + x(8)
f7 = f1 + f4             f7 = x(4) - x(8)
f8 = f1 - f4             f8 = (0.7,-0.7)*f3
f9 = x(2) + x(6)         f9 = (0,-1) * f5
f10 = x(2) - x(6)        f10 = (-0.7,-0.7)*f7
f11 = x(4) + x(8)        f11 = f0 + f4
f12 = x(4) - x(8)        f12 = f0 - f4
f13 = (0,-1)*f12         f13 = f2 + f6
f14 = f9 + f11           f14 = f2 - f6
f15 = f9 - f11           f15 = (0,-1)*f14
f16 = f10 + f13          y(1) = f11 + f13
f17 = f10 - f13          y(5) = f11 - f13
f18 = (0.7,-0.7)*f16     y(3) = f12 + f15
f19 = (0,-1) * f15       y(7) = f12 - f15
```

```
f20 = (-0.7,-0.7)*f17     f20 = f1 + f9
y(1) = f5 + f14           f21 = f1 - f9
y(5) = f5 - f14           f22 = f8 + f10
y(2) = f7 + f18           f23 = f8 - f10
y(6) = f7 - f18           f24 = (0,-1)*f23
y(3) = f6 + f19           y(2) = f20 + f22
y(7) = f6 - f19           y(6) = f20 - f22
y(4) = f8 + f20           y(4) = f21 + f24
y(8) = f8 - f20           y(8) = f21 - f24
```

For FFTs of size 2, 4, 8, 16, 32, and 64, dynamic programming was used on the formulas generated by Equation 10. Each formula was passed to the SPL compiler with the unrolling flag turned on to enforce the generation of straight-line (loop free) code.

We compared with the performance of the FFTW codelets, a set of optimized straight-line code for small-size FFTs. These codelets accept two parameters, “istride” and “ostride”, which are used to control the access to the input and output vectors. For each codelet, we measured the performance of the original code and of a modified version involving fewer instructions because it assumes that the stride is always 1. The modified version was expected to be faster because it contains fewer instructions. However, this was not always true. On the SPARC machine, the modified version performed no better than the original version and, in some cases, it was much slower. One explanation is that stride computations are integer operations that can be executed in parallel with floating point operations and removing the stride does not necessarily reduce execution time. Variability caused by scheduling strategies probably account for the slowdown.

Figure 3 compares the performance of the code generated by the SPL compiler for small size FFTs after performing the search described above with the performance of the FFTW codelets. The performance is measured in terms of “pseudo MFlops”, which is a value calculated by using the equation $\frac{5Nlog_2N}{t}$, where N is the size of FFT and t is the execution time in microseconds. The performance of the codes generated by the SPL compiler is very close to the performance of the FFTW codelets.



(a) SPARC

(b) MIPS

(c) Pentium II

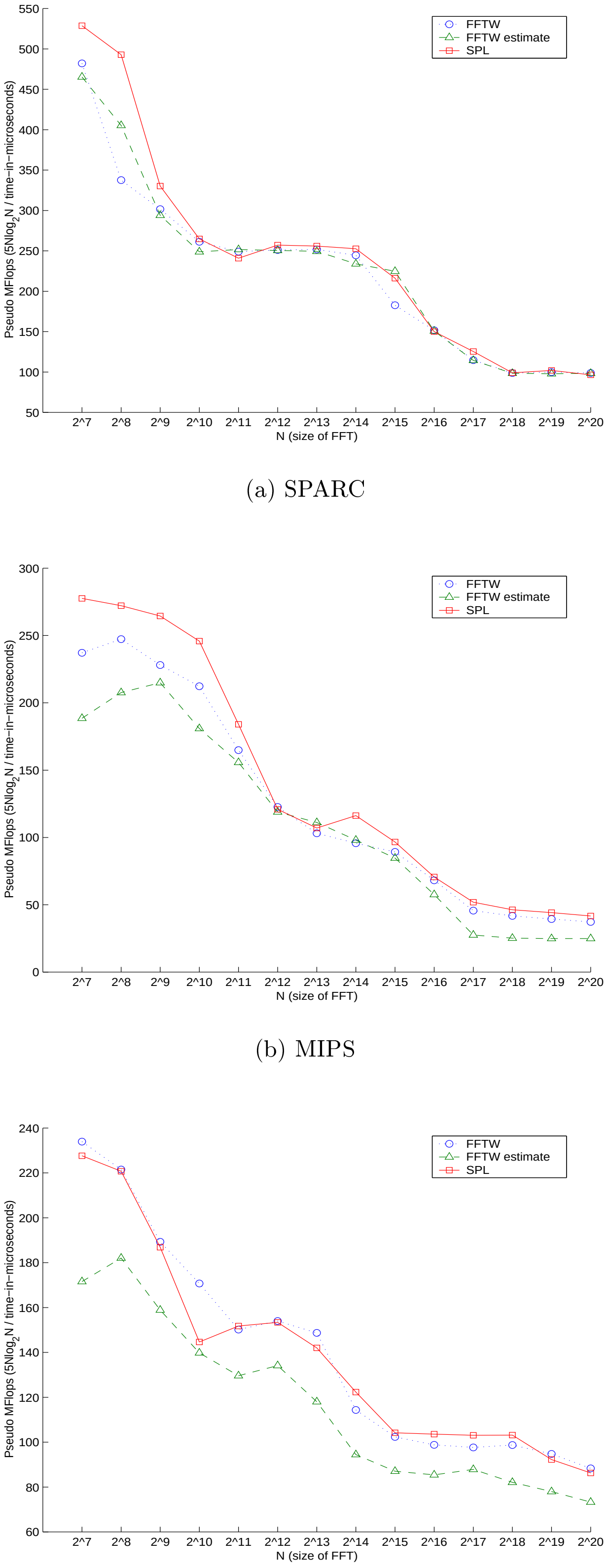
Figure 3: Performance for small-size FFTs)

4.2 Large Size FFTs

We decided to use straight-line code only for FFT sizes less than 64 in part to parallel FFTW but also because straight-line code for FFT sizes greater than 64 do not fit in the cache of the SPARC we used in our experiments.

For FFT sizes larger than 64, we factored the formula into FFTs of size 2, 4, 8, 16, 32, or 64, and implemented the computation using loops containing the straight-line codes generated for the small-size FFTs as discussed above. We used the best program resulting from the previous search under the assumption that a good formula for small size FFTs also could be a good sub-formula for larger size FFTs. It is possible we could have missed the actual best formula but following this approach significantly reduced the search space and made the search for large-size FFTs possible.

In this experiment, the search space was restricted to binary Cooley-Tukey style factorization, as expressed in Equation 5 (Section 2.1), and to right-most factorization. This means that when a FFT of size N is factored into two sub-FFTs of size N_1 and N_2, the sub-FFT of size N_1 will not be factored further. Only the second sub-FFT could be factored again. Again, this decision paralleled FFTW and



(a) SPARC

(b) MIPS

(c) Pentium II

Figure 4: Performance for large-size FFTs

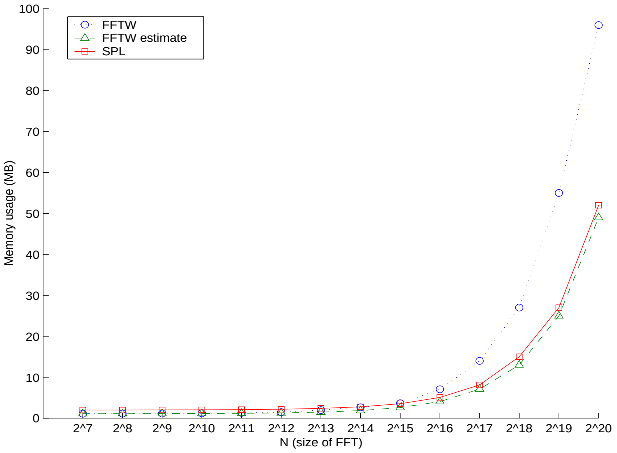


Figure 5: Memory consumption for large-size FFTs

reduced the size of the search space.

The search strategy we used for large-size FFTs is a modified version of dynamic programming. Ordinary dynamic programming keeps the best result for each size. Our version keeps the three best results for each size. This strategy increases the chance of finding the best formulas because the best formula for one size is not necessarily also the best sub-formula for a larger size.

In FFTW, large-size FFTs are computed recursively using three components: the planner, the executor, and the codelets. The planner searches for an optimal factorization at run-time using dynamic programming. This factorization, called a plan, is then interpreted by the executor. The executor calls to the codelets in the order specified by the plan. FFTW also has an option to select plan by "estimating" instead of measuring the execution time. This option saves time and memory.

The performance results for FFT of size 2^7 to 2^{20} are shown in Figure 4. One line (labeled "SPL") represents the performance of the loop code generated by SPL compiler, a second line (labeled "FFTW") represents the performance of FFTW when the the plan is chosen by measuring the execution time, and a third line (labeled "FFTW estimate") represents the performance of FFTW when the plan is chosen using estimation. The time for planning in FFTW was excluded from the measurement.

As was the case for small-size FFTs, the performance of the code generated by the SPL compiler is similar to the performance of FFTW.

In addition to the performance, we also measured memory requirements. Figure 5 shows that the memory required to run the code generated by the SPL compiler is similar to the memory required by "FFTW estimate". More memory is required for FFTW to find a plan by measuring the execution time. However, this is not a significant issue, because in FFTW one can save the plan to a file and re-use it in later sessions.

An interesting observation of the performance curves in Figure 4 is that there are two large drops in each graph. For example, for the SPARC machine, they happened at size 2^9 and 2^{16}. By analyzing the size of individual segments of the executables, we found out that the drops are related to the

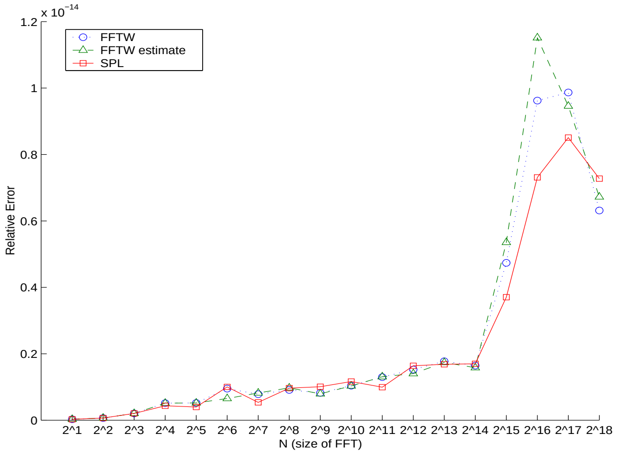


Figure 6: Accuracy of FFT computation

data cache. The size of data segment exceeded L1 cache limit (16KB) when the problem size reached 2^9, and exceeded L2 cache limit (2MB) when the problem size reached 2^{16}. The next limitation would be the physical memory size. Our SPARC machine has 128MB memory, about 90MB of which can be used to run a specific program. For this reason, from size 2^{21} to size 2^{22} we expect to see another performance drop. The curves of the other two machines have similar shapes.

It is not surprising that our loop code matched the performance of FFTW's recursive code. Our loop code was generated by applying recursive factorization rules and hence followed the same computation order and data access pattern of the recursive code used by FFTW. One possible drawback of the loop code is that code size may increase faster than recursive code because recursive code can re-use "codelets" while loop code has to duplicate them. Our analysis, however, showed that the increase of code size was very slow. The size of the text segment of the loop code for size 2^{20} was only 50 percent larger than that of size 2^7.

Finally, we measured the accuracy of the computation represented by these code by using the package `benchfft` [4]. Figure 6 shows the relative error of FFT of each size.

5. RELATED WORK

Our approach is similar to that used by FFTW [5, 6]. However, FFTW is specific to the FFT. Since the algorithm and implementation are mixed in the package, it's not easy to extend the ability to other algorithms. A recent work by [17] showed how to modify FFTW to support discrete cosine transform. However, doing this requires a good understanding of the internal mechanism of FFTW, especially the compiler that generates the codelets. This requirement is too much for an ordinary user. Our use of SPL allows us to implement and optimize a far more general set of programs. FFTW also utilizes a special purpose compiler. However, their compiler is based on built-in code sequences for a fixed set of FFT algorithms and is used for generating only codelets. FFTW uses runtime dynamic programming to search for efficient implementations, while our approach performs search at compile-time and allows more flexible selection of search algorithm and search space.

Similar to the work in FFTW is the package described in [11] for computing the Walsh-Hadamard transform (WHT). This work is closer in spirit to the work in this paper in that the different algorithms considered are expressed mathematically and the search is carried out over a space of formulas. Similar to FFTW, the WHT package is restricted to a specific transform, and a code generator restricted to the WHT is used rather than a compiler.

Another closely related work is EXTENT [3], which uses tensor product to represent block recursive algorithms and uses a translator to generate programs in high-level languages. Our work differs from EXTENT in several way: at first, we provide a language for describing algorithms, while in EXTENT an "application subsystem" has to be added for every algorithm. Secondly, our use of templates enable us to extend the algorithm defining language and to change the code generation strategy without modifying the system; while in EXTENT, code generation is built into the translator so that it doesn't have this flexibility; thirdly, in EXTENT performance data is fed back to the user for manual performance tuning; while we use search to automate the tuning procedure.

Also related to our approach is the ATLAS project, whose goal is portable high performance implementation of the Basic Linear Algebra Subroutines (BLAS) [18]. They isolate machine specific operations to several routines that deal with performing optimized on-chip, cache contained matrix multiply. General matrix multiply is built from these basic routines. The basic routines are created by a code generator that searches for the correct blocking and loop unrolling factors based on timing information. PHiPAC[2] presents some guidelines for writing compiler friendly high performance C programs and makes use of code generators to create special purpose (for matrix multiply) C code that follows these guidelines. Their code generators are parameterized so that they can tune performance by searching over these parameters. Our work differs from theirs in that they are focusing on general linear algebra routines, while we are interested in different signal transform algorithms. Furthermore, their code generators are hand-coded, while we can automate the entire procedure through the use of a domain-specific language and compiler.

We use compile-time search to find optimal implementations. Kisuki and Knijnenberg [12] presented similar ideas using the term "iterative compilation". They experimented with loop tiling, loop unrolling and array padding with different parameters, as well as different search algorithms. Their results appear to be good compared with static selection algorithms. One difference between their work and ours is that we allow a wider range of search objects because, in addition to compiler options, input programs can be in the search space.

6. CONCLUSIONS

We have presented a domain-specific language, SPL, and compiler for representing and implementing signal processing algorithms. The SPL compiler applies several optimizations and transformations and is extensible in that new operators and their semantics can be introduced using a pattern matching mechanism and templates. When the compiler is combined with an algorithm generator and a search engine, as is the case in SPIRAL, it is possible to automatically search for optimal implementations. This paper shows that when the compiler is incorporated into a search of FFT factorizations, it produces library routines that are competitive in their performance with those generated by FFTW. The SPL compiler, however, is more general in that it can generate libraries not only for FFT computations, as is the case with FFTW, but also for many other classes of algorithms. The results presented in the paper illustrate the power of domain-specific languages and compilers. By focusing on a particular domain, the process of generating highly-tuned code can be automated to a degree that is not possible in more general contexts, at least with today's technology. The signal processing domain is particularly attractive because of the many applications of signal processing algorithms and the degree of interest that exist today in these applications. However, signal processing is not necessarily better suited for an approach like the one discussed in this paper than other problem domains. It is likely, therefore, that we will see many more domain-specific projects in other areas in the near future.

7. ACKNOWLEDGMENTS

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

8. 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, June 1996. Presented at the DARPA ACMP PI meeting.

[2] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c code methodology. In *Proc. ICS1997*, 1997. `http://www.icsi.berkeley.edu/~bilmes/phipac/`.

[3] D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. EXTENT: A portable programming environment for designing and implementing high-performance block recursive algorithms. In *Supercomputing 1994*, pages 49–58, 1994.

[4] M. Frigo. BenchFFT. `http://www.fftw.org/benchfft/`.

[5] M. Frigo. A fast fourier transform compiler. In *PLDI '99*, pages 169–180, 1999.

[6] 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`.

[7] M. T. H. Henrik V. Sorenson, C. Sidney Burrus. *Fast Fourier Transform Database*. PWS Publishing Company, Boston, 1995.

[8] J. Johnson, R. Johnson, D. Padua, and J. Xiong. TPL: Tensor Product Language, 1999. `http://www.ece.cmu.edu/~spiral/tpl.html`.

[9] J. R. Johnson and R. W. Johnson. Challenges of computing the fast Fourier transform. In *Proc. Optimized Portable Application Libraries (OPAL) Workshop, A DARPA/NSF sponsored workshop in Kansas City, June. 2–3*, 1997.

[10] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9(4):449–500, 1990.

[11] J. R. Johnson and M. Püschel. In search of the optimal Walsh-Hadamard transform. In *Proc. ICASSP 2000*, 2000.

[12] T. Kisuki, P.M.W.Knijnenberg, M.F.P.O'Boyle, and H.A.G.Wijshoff. Iterative compilation in program optimization. In *Proc. CPC2000*, pages 35–44, 2000.

[13] 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.ece.cmu.edu/~spiral`.

[14] K. R. Rao and P. Yip. *Discrete Cosine Transform*. Academic Press, 1990.

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

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

[17] R. Vuduc and J. Demmel. Code generators for automatic tuning of numerical kernels: Experiences with fftw. In *ICFP 2000*, 2000.

[18] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS), 1998. `http://www.netlib.org/atlas/`.