Searching for the Best FFT Formulas with the SPL Compiler*

Jeremy Johnson[1], Robert W. Johnson[2], David Padua[3], and Jianxin Xiong[3]

[1] Drexel University, Philadelphia, PA 19104, jjohnson@mcs.drexel.edu
[2] MathStar Inc., Minneapolis, MN 55402, rwj@mathstar.com
[3] University of Illinois at Urbana-Champaign, Urbana, IL 61801, {padua,jxiong}@cs.uiuc.edu

Abstract. This paper discuss an approach to implementing and optimizing fast signal transforms based on a domain specific computer language, called SPL. SPL programs, which are essentially mathematical formulas, represent matrix factorizations, which provide fast algorithms for computing many important signal transforms. A special purpose compiler translates SPL programs into efficient FORTRAN programs. Since there are many formulas for a given transform, a fast implementation can be obtained by generating alternative formulas and searching for the one with the fastest execution time. This paper presents an application of this methodology to the implementation of the FFT.

1 Introduction

This paper discusses an approach to implementing and optimizing fast signal transforms, such as the fast Fourier transform (FFT), which can be expressed as matrix factorizations. The approach relies on a domain specific computer language, called SPL, for expressing matrix factorizations, and a special purpose compiler which translates matrix factorizations expressed in SPL into efficient FORTRAN programs. Since SPL programs are mathematical formulas representing matrix factorizations, it is possible to transform SPL programs into equivalent programs through the use of theorems from linear algebra. Moreover, given a matrix, it is possible, through the application of various mathematical transformations, to automatically generate a large number of mathematically equivalent SPL programs applying the given matrix to a vector. This allows us, through the use of the SPL compiler, to systematically search for the SPL program with the fastest execution time. When the matrix represents a signal transform, such as the FFT, this procedure allows us to find a fast implementation of the signal transform. The only difficulty with this approach is that the size of the search

* This work was partially supported by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting. A version of this paper, based on an earlier version of the SPL compiler, will appear in *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Springer-Verlag, 2001.

space grows exponentially and therefore more refined search techniques than exhaustive search must be utilized. This paper presents an application of this methodology to the implementation and optimization of the FFT. Various techniques are presented that allow us to find nearly optimal SPL programs while only searching through a small subspace of the allowed programs.

The approach presented in this paper was first outlined in [1] and further discussed in [7]. Our work is part of a larger effort, called SPIRAL [11], for automatically implementing and optimizing signal processing algorithms. This paper presents the first efficient implementation of the SPL compiler and hence this first true test of the methodology.

Our approach is similar to that used by FFTW [4, 3], however, FFTW is specific to the FFT, while our use of SPL allows us to implement and optimize a far more general set of programs. FFTW also utilizes a special purpose compiler and searches for efficient implementations. However, their compiler is based on built-in code sequences for a fixed set of FFT algorithms and is only used for generating small modules of straight-line code, called codelets. Furthermore, even for the FFT, we consider a much larger search space and use search techniques other than dynamic programming as is the case in FFTW. Similar to the work in FFTW, is the package described in [8] 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. Finally, in addition to searching over a space of mathematical formulas, this paper also considers a search over a space of compiler options and techniques. In particular, we allow compilation of SPL programs to perform various amounts of loop unrolling, and include in our search different amounts of unrolling.

Related to our approach is the ATLAS project, which uses empirical search techniques to automatically tune high performance implementation of the Basic Linear Algebra Subroutines (BLAS) [15]. The application of search techniques in determining appropriate compiler optimizations has been used in several efforts. Kisuki and Knijnenberg [9] presented similar ideas using the term of "iterative compilation". They experimented with loop tiling, loop unrolling and array padding with different parameters, as well as different search algorithms. Their results appeared to be good compared with static selection algorithms. One difference between their work and ours is that our search is carried on both different input programs and different compilation parameters, thus provides more flexibility. Massalin [10] presented a "superoptmizer" which tries to find the shortest machine code sequence to compute a function. They carried out an exhaustive search within the space of a subset of the machine instruction set to ensure that the final result is optimal. Since the size of their search space increases exponentially, the method only works for very short programs.

In the remainder of the paper we review the SPL language, the SPL compiler, SPL formula generation, and apply search techniques to systematically search for the optimal FFT implementation.

2 Matrix Factorizations and SPL

Many digital signal processing (DSP) transforms are mathematically given as a multiplication of a matrix M (the transform) with a vector x (the sampled signal). Examples are the discrete Fourier transform (DFT), trigonometric transforms, and the Hartley and Haar transforms, [12]. A fast algorithm for these transforms can be given by a factorization of M into a product of sparse matrices, [12, 14]. For example, the well-known Cooley/Tukey algorithm, [2], also known as fast Fourier transform (FFT), for computing an rs-point DFT, F_{rs}, can be written as

$$\mathrm{F}_{rs} = (\mathrm{F}_r \otimes \mathrm{I}_s)T(\mathrm{I}_r \otimes \mathrm{F}_s)L \tag{1}$$

where $\otimes$, denotes the tensor product, I_s is the identity matrix, T a diagonal matrix, and L a permutation matrix both depending on r and s.

SPL [6] is a domain-specific programming language for expressing and implementing matrix factorizations. It was originally developed to investigate and automate the implementation of FFT algorithms, [1]. As such, some of the built-in matrices are biased towards the expression of FFT algorithms, however, it is important to note that SPL is not restricted to the FFT. It contains features that allow the user to introduce notation suitable to any class of matrix expressions.

This section reviews some notation for expressing matrix factorizations and summarizes the SPL language.

2.1 Matrix Factorizations

In this section we show how a matrix factorization of the n-point DFT matrix, F_n, leads to a program to compute the matrix-vector product of F_n applied to a vector.

The n-point Fourier Transform is the linear computation

$$y(l) = \sum_{k=0}^{n-1} \omega_n^{lk} x(k) \qquad\qquad 0 \le l < n \tag{2}$$

where $\omega_n = e^{-2\pi i/n}$. Computation of the Fourier transform can be represented by the matrix-vector product $y = F_n x$, where F_n is the $n \times n$ matrix $[\omega_n^{lk}]_{0 \le l,k < n}$.

Computing the n-point Fourier transform as a matrix-vector product requires $O(n^2)$ operations. A faster algorithm can be obtained by factoring F_n into a product of simpler matrices.

Theorem 1. *Let $N = 2m$. Then*

$$F_N L_m^N = \begin{bmatrix} F_m & W_m F_m \\ F_m & -W_m F_m \end{bmatrix}, \tag{3}$$

where $W_m = diag(1, \omega_N, \ldots, \omega_N^{m-1})$ *and* L_m^N *is the permutation that rearranges the elements of a vector into* m *segments containing elements of the input separated by a stride of* m *and with the* i*-th segment starting with the* i*-th element of the input.*

This block decomposition can be written compactly using the tensor product (also called Kronecker product). Let A be an $m \times m$ matrix and B a $n \times n$ matrix then the tensor product of A and B, $A \otimes B$, is the $mn \times mn$ matrix defined by the block matrix product

$$A \otimes B = [a_{ij}B]_{1 \le i,j \le m} = \begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \ldots & a_{m,m}B \end{bmatrix}. \tag{4}$$

Corollary 1.

$$F_{2m} = (F_2 \otimes I_m)T_m^{2m}(I_2 \otimes F_m)L_2^{2m}, \tag{5}$$

where $T_m^{2m} = diag(I_m, W_m)$.

More generally,

Theorem 2 (Cooley-Tukey).

$$F_{rs} = (F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)L_r^{rs}, \tag{6}$$

where L_r^{rs} *is a permutation matrix and* T_s^{rs} *is a diagonal matrix. Let* e_i^n, $0 \le i < n$, *be a vector of size* n *with a one in the* i*-th location and zeroes elsewhere, and let* ω_{rs} *be a primitive* rs*-th root of unity. Then* $T_s^{rs}(e_i^r \otimes e_j^s) = \omega_{rs}^{ij}(e_i^r \otimes e_j^s)$ *and* $L_r^{rs}(e_i^r \otimes e_j^s) = (e_j^s \otimes e_i^r)$.

For example, the 4-point DFT can be factored like the following:

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

An algorithm to compute $y = F_N x$, where $N = 2m$, can be obtained using the previous factorization. The output y is obtained using the factorization $F_N = (F_2 \otimes I_m)T_m^N(I_2 \otimes F_m)L_2^N$ by first applying L_2^N to the input x, then applying $(I_2 \otimes F_m)$ to the resulting vector $L_2^N x$, and so on.

2.2 SPL

A SPL program is essentially a sequence of mathematical formulas built up from a parameterized set of special matrices and algebraic operators such as matrix composition, direct sum, and the tensor product. The SPL language uses a prefix notation similar to lisp to represent formulas. For example, the expressions `(compose A B)` and `(tensor A B)` correspond to the matrix product $A \cdot B$ and the tensor product $A \otimes B$ respectively. The language also includes special symbols such as `(F n)` and `(I m)` to represent the discrete Fourier transform matrix F_n and the identity matrix I_m. In addition to the built-in symbols the user can assign a SPL formula to a symbol to be used in other expressions. Moreover, new parameterized symbols can be defined so that SPL programs can refer to other sets of parameterized matrices.

For example, the Cooley-Tukey factorization of F_{rs} for any integer values r and s is represented by the SPL expression

```
(compose (tensor (F r)(I s))(T rs s)(tensor (I r)(F s))(L rs r))
```

Note that all SPL programs represent linear computations of a fixed size. Expressions with parameters such as the one above are used to instantiate families of SPL programs.

The power of SPL for expressing alternative algorithms is illustrated by the following list of formulas corresponding to different variants of the FFT, [5, 13, 14]. Each of the formulas was obtained using the Cooley-Tukey factorization and elementary properties of the tensor product. In the following formulas, the symbol R_{2^k} denotes the k-bit bit-reversal permutation.

Apply Cooley/Tukey inductively

$$F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes F_4) L_2^8 \tag{7}$$

Recursive FFT

$$F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes ((F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4)) L_2^8 \tag{8}$$

Iterative FFT (Cooley/Tukey)

$$F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2) R_8 \tag{9}$$

Vector FFT (Stockham)

$$F_8 = (F_2 \otimes I_4) T_4^8 L_2^8 (F_2 \otimes I_4)(T_2^4 \otimes I_2)(L_2^4 \otimes I_2)(F_2 \otimes I_4) \tag{10}$$

Vector FFT (Korn/Lambiotte)

$$F_8 = (F_2 \otimes I_4) T_4^8 L_2^8 (F_2 \otimes I_4)(T_2^4 \otimes I_2) L_2^8 (F_2 \otimes I_4) L_2^8 R_8 \tag{11}$$

Parallel FFT (Pease)

$$F_8 = L_2^8 (I_4 \otimes F_2) L_4^8 T_4^8 L_2^8 L_2^8 (I_4 \otimes F_2) L_4^8 (T_2^4 \otimes I_2) L_2^8 L_2^8 (I_4 \otimes F_2) R_8 \tag{12}$$

These formulas are easily translated into SPL programs. The following SPL program corresponds to the formula for the iterative FFT on 8 points.

```
(define R8 (permutation (0 4 2 6 1 5 3 7)))
(compose (tensor (F 2) (I 4)) (T 8 4)
         (tensor (I 2) (F 2) (I 2)) (tensor (I 2) (T 4 2))
         (tensor (I 4) (F 2)) R8)
```

In general, a SPL program consists of the following constructs. This list refers to SPL 3.01 (see [6] for future enhancements and the most current version).

```
(1) matrix operations
        (tensor formula formula ...)
        (compose formula formula ...)
        (direct_sum formula formula ...)
(2) direct matrix description:
        (matrix (a11 a12 ...) (a21 a22 ...) ... )
        (diagonal (d1 d2 ...))
        (permutation (p1 p2 ...))
(3) parameterized matrices:
        (I n)
        (F n)
        (T mn n) or (T mn n, start:step:end)
        (L mn n)
(4) parameterized scalars:
        W(m n)        ; e^(2*pi*i*n/m)
        WR(m n)       ; real part of W(m n)
        WI(m n)       ; imaginary part of W(m n)
        C(m n)        ; cos(pi*n/m)
        S(m n)        ; sin(pi*n/m)
(5) symbol definition:
        (define name formula)
        (template formula (i-code-list))
```

Templates are used to define new parameterized matrices and by the SPL compiler to determine the code for different formulas in SPL. Templates are defined using an language independent syntax for code called i-code.

3 SPL Compiler

The execution of the SPL compiler consists of six stages: (1) parsing, (2) semantic binding, (3) type control, (4) optimization, (5) scheduling, and (6) code generation. The parser builds an abstract syntax tree (AST), which is then converted to intermediate code (i-code) using templates to define the semantics of different SPL expressions. The i-code is expanded to produce type dependent code (e.g. double precision real or complex) and loops are unrolled depending on compiler parameters. After intermediate code is generated, various optimizations such as constant folding, copy propagation, common sub-expression elimination, and algebraic simplification are performed. Optionally, data dependency analysis is used to rearrange the code to improve locality. Finally, the intermediate code is converted to FORTRAN, leaving machine dependent compilation stages (in particular register allocation and instruction scheduling) to a standard compiler. (The code generator could easily be modified to produce C code or even native assembly code directly instead for FORTRAN.) Different options to the compiler, command-line flags or compiler directives, control various aspects of the compiler, such as the data types, whether loops are unrolled, and whether the optional scheduler is used. Some compiler optimizations and instruction scheduling can also be obtained at the SPL level, by transforming the input formulas.

The nodes in the AST corresponding to a SPL formula can be interpreted as programs for computing a specific matrix-vector product. Code is generated by combining and manipulating these code segments, to obtain a program for the matrix-vector product specified by the SPL program. For example, a composition $A \cdot B$ is compiled into the sequence $t = B \cdot x; y = A \cdot t$ mapping input signal x into output signal y with intermediate signal vector t. In the same way, the direct sum compiles into operations acting on parts of the input signal in parallel. The tensor product of code sequences for computing A and B can be obtained using the equation $A \otimes B = L_m^{mn}(I_n \otimes A)L_n^{mn}(I_m \otimes B)$.

For example, the code produced for the SPL program corresponding to the 4-point Cooley/Tukey algorithm is

```
subroutine F4(y,x)
implicit complex*16(f)
implicit integer(r)
complex*16 y(4),x(4)
f0   =  x(1) + x(3)
f1   =  x(1) - x(3)
f2   =  x(2) + x(4)
f3   =  x(2) - x(4)
f4   = (0d0,-1d0)*f3
y(1) =  f0 + f2
y(3) =  f0 - f2
y(2) =  f1 + f4
y(4) =  f1 - f4
end
```

In this example, two compiler directives were added: one giving the name `F4` to the subroutine and one causing complex arithmetic to be used. Looking at this example, one already sees several optimizations that the compiler makes (e.g. multiplications by 1 and -1 are removed). More significantly, multiplication by the permutation matrix L_2^4 is performed as re-addressing in the array accesses. Another important point is that scalar variables were used for temporaries rather than an array. This has significant consequences on the FORTRAN compiler's effectiveness at register allocation and instruction scheduling.

Changing the code type to real, `#codetype real`, breaks up complex numbers into real and imaginary parts which gives the chance for further optimizations. In the case above the (complex) multiplication vanishes.

```
subroutine F4(y,x)
implicit real*8(r)
real*8 y(8),x(8)
r0   =  x(1) + x(5)
r1   =  x(2) + x(6)
r2   =  x(1) - x(5)
r3   =  x(2) - x(6)
r4   =  x(3) + x(7)
r5   =  x(4) + x(8)
r6   =  x(3) - x(7)
r7   =  x(4) - x(8)
y(1) =  r0 + r4
y(2) =  r1 + r5
y(5) =  r0 - r4
y(6) =  r1 - r5
y(3) =  r2 + r7
y(4) =  r3 - r6
y(7) =  r2 - r7
y(8) =  r3 + r6
end
```

In the previous example, we produced straight-line code. The SPL compiler is also capable of producing code with loops. For example, the formula $I_n \otimes A$ has a straight-forward interpretation as a loop with n iterations, where each iteration applies A to a segment of the input vector. The SPL compiler is instructed to generate code using this interpretation by the following template in which `ANY` matches any integer and `any` matches any SPL expression.

```
; (tensor Im An) parameters:
; p.0: mn, p.1:size(I)=m, p.2:ANY=m, p.3:size(A)=n, p.4:A
(template (tensor (I ANY) any)
  (r.0 = p.3-1
   do p.1
     y(0:1:r.0 p.3) = call p.4(x(0:1:r.0 p.3))
   end))
```

The following example shows how to use the SPL compiler to combine straight-line code with loops using formula manipulation and loop unrolling (loop unrolling is controlled by the compiler directive `#unroll` or by specifying a command line option `-Bn` which indicates that code for matrices of dimension n or less are to be unrolled). Using a simple property of the tensor product, $I_{64} \otimes F_2 = I_{32} \otimes (I_2 \otimes F_2)$. With the option `-R -l-B4`, the SPL compiler produces straight-line code for loop body which computes $(I_2 \otimes F_2)$.

```
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
```

SPL clearly provides a convenient way of expressing and implementing matrix factorizations; however, it can only be considered as a serious programming tool, if the generated code is competitive with the best code available. One strength of the SPL compiler is its ability to produce long sequences of straight-line code. In order to obtain maximal efficiency small signal transforms should be implemented with straight-line code thus avoiding the overhead of loop control or recursion. One of the fastest available packages for computing the FFT, FFTW, [4], utilizes this idea; however, the code generation facilities of FFTW are restricted to several FFT algorithms. SPL offers far greater generality and the efficiency of the resulting code is comparable to FFTW. Table 1 compares code for small FFTs generated by the SPL compiler to the FFT codelets of FFTW on a Sun Ultra 5, 333 MHz.

n	1	2	3	4	5	6
FFTW	0.04	0.07	0.23	0.53	1.50	4.34
SPL FFT	0.03	0.05	0.22	0.50	1.45	4.29

Table 1. Runtime of FFT(2^n) in μs, FFTW vs. SPL

4 SPL Formula Generator

SPL programs can be derived by applying various mathematical theorems such as the Cooley-Tukey theorem. Such mathematical theorems can be thought of as SPL program transformations. For a given matrix, it is possible to generate many different SPL programs for performing the same mathematical computation. While mathematically equivalent, different SPL programs may, when translated by the SPL compiler, lead to FORTRAN programs with vastly different performance. Optimizing a given linear computation can be thought of as a search problem. Simply generate all possible SPL programs, compile them, and select the one which leads to the fastest execution time. By "all" possible algorithms, we mean all of those SPL programs that can be generated by applying a fixed set of mathematical theorems.

In this section we consider a generalization of the Cooley-Tukey theorem which allows us to generate many different FFT algorithms. This space of FFT algorithms allows us to explore various combinations of iterative and recursive FFT algorithms.

The following theorem is an easy corollary of the Cooley-Tukey theorem.

Theorem 3. *Let $N = 2^n$ and let $N = N_1 \cdots N_t$, with $N_i = 2^{n_i}$, be a factorization of N. Using the notation $N(i) = N_1 \cdots N_i$, $\bar{N}(i) = N/N(i)$, $N(0) = 1$, and $N(t) = N$, Then*

$$F_N = \prod_{i=t}^{1} \left\{ (I_{N(i-1)} \otimes F_{N_i} \otimes I_{\bar{N}(i)})(I_{N(i-1)} \otimes T_{\bar{N}(i)}^{\bar{N}(i-1)}) \right\} R \tag{13}$$

where

$$R = \prod_{i=1}^{t} (I_{N(i-1)} \otimes L_{N_i}^{\bar{N}(i-1)}) \tag{14}$$

is a generalization of the bit-reversal permutation.

Applying this factorization recursively to each F_{N_i} until the base case equal to F_2 leads to an algorithm for computing F_N. The special case when $t = 2$ and $N_1 = 2$, and $N_2 = N/2$ leads to the standard recursive FFT. Other recursive breakdown strategies are obtained by choosing different values for N_1 and N_2. The special case when $N_1 = \cdots N_t = 2$ leads to the standard iterative, radix two, FFT. The space of formulas we consider allows us to explore various amounts of recursion and iteration along with different breakdown strategies.

Corresponding to this factorization is a partition of the integer $n = n_1 + \cdots + n_t$. A similar partition is obtained for each n_i, and the resulting algorithm can be represented as a tree, whose nodes are labeled by the integers n_i. The tree, called a partition tree, has the property that the value at any node is equal to the sum of the values of its children. Partition trees were introduced in [8].

The number of trees, T_n, with root equal to k is given by the recurrence

$$T_n = \begin{cases} 1 & n = 1 \\ \prod_{n=n_1+\cdots+n_t} T_{n_1} \cdots T_{n_t} & n > 1 \end{cases} \tag{15}$$

The solution to this recurrence is $\Theta(\alpha^n / n^{3/2})$, where $\alpha = 3+2\sqrt{2} \approx 5.83$. Table 2 lists the initial values for T_n.

n	# of formulas	n	# of formulas	n	# of formulas	n	# of formulas
1	1	5	45	9	20793	13	13649969
2	1	6	197	10	103049	14	71039373
3	3	7	903	11	518859	15	372693519
4	11	8	4279	12	2646723	16	1968801519

Table 2. Size of the formula space

Figure 1 shows the distribution of runtime (in microseconds) for all 20793 formulas for F_{2^9}. The left side plots performance in ms verses formula and the right side shows the distribution of runtimes. The fastest formula is about five times faster than the slowest one and 2.25 times faster than the mean. Also observe that only 1.7% of the formulas fall into the bin with the highest performance. This illustrates the wide range in performance and that their truly is an optimization problem. Moreover, optimal formulas are rare.

All runtimes presented in this paper were obtained on an Ultra 5 workstation, with a 248MHz UltraIIi CPU and 512MB memory. The software environment was Solaris 7, SUN Fortran 77 5.0, and SPL Compiler 3.28. The table shows that there is a wide range in runtime (approximately an order of magnitude), and that a significant gain in performance can be obtained by using the proposed optimization procedure. The only difficulty in this approach to optimization, is that the number of formulas considered grows exponentially and hence an exhaustive search is not feasible for large values of n. In the next section we will present some techniques for reducing the search time.

5 Search

In this section we search for the optimal FFT implementation using the SPL compiler and the FFT formulas generated in the previous section. Several search strategies are applied in an effort to reduce the number of formulas considered, while still obtaining a nearly optimal implementation. In addition to searching

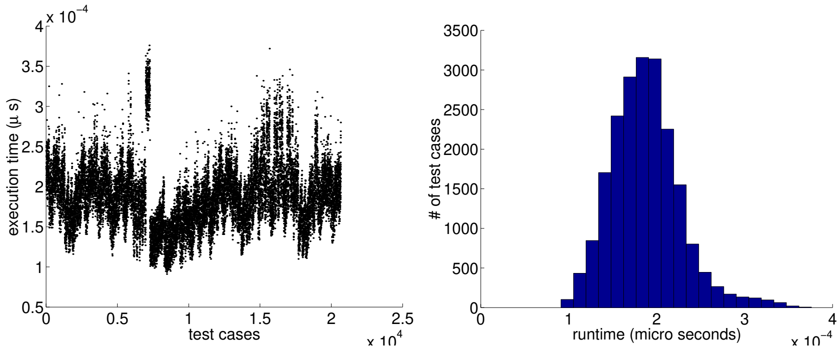

Fig. 1. Execution time(μs) and distribution, n=9, Exhaustive

over the space of FFT formulas presented, we also search over several compiler strategies related to loop unrolling.

For each formula, there is a tradeoff between generating straight line code and loop code. Straight line code involves fewer instructions than loop code but its size increases with N. This leads to performance degradation due to cache and memory limitations. Except for small sized problems, partial unrolling is probably the best choice. The SPL compiler can generate several versions of code with different degrees of loop unrolling. The controlling parameter is called the *unrolling threshold*, which is denoted as T_u in this paper. Given a formula, the code generation is carried out top-down over the tree. If the size of a node is less than or equal to T_u, then the entire subtree rooted at this node is implemented by straight-line code; otherwise, the code corresponding to this node will be implemented as a loop.

It is impractical to try all the possible values of T_u. Preliminary experiments show that both compile time and execution time increase dramatically when T_u is greater than 7 on the Ultra 5 workstation. This value could be different on other machines or for other problems, but we believe that the upper bound of the value of interest can always be identified and is much smaller than N when N is large enough. Therefore, in our experiment, only values of $T_u \leq 7$ are considered. The entire search space is the Cartesian product of the formula space and the set of values of T_u.

Since exhaustive search is impractical it is necessary to use a search strategy that enables the optimal formual, or a nearly optimial formula, to be found without considering such a large space of formulas. As a first step one might randomly choose a subset of formulas to consider. However, Fig. 1 suggests that this approach is unlikely to find an optimal formula.

Instead we experimented with several versions of heuristic search based on local transformations of formulas. The first approach, called *single point search* (SPS), starts with a random formula and repeatedly finds a neighboring formula while checking to see if the neighbor leads to better performance. The second approach, called *multiple point search* (MPS), starts with a smaller set of formulas (in our case 4) and searches locally in a neighborhood of each formula in the initial set.

Both approaches rely on a function *nexttree* to find a "neighbor" to an existing formula. The rules for finding a "neighbor" in SPS are shown in Fig.2. The rules used by MPS are the same, except $R7(Random)$ is not used. The "neighbors" are similar in their tree structure and, as a result, their performance values tend to be close (except when $R7$ is applied).

```
R1 (Merge):      merge two subtrees;
R2 (Split):      split one node into two;
R3 (Resize):     resize two subtrees by one;
R4 (Swap):       swap two subtrees;
R5 (Unrollmore): increase the unrolling threshold by one;
R6 (Unrollless): decrease the unrolling threshold by one;
R7 (Random):     randomly choose another tree;
R8 (Chgchild):   apply these rules to a subtree;
```

Fig. 2. Rules for finding a neighbor

The two approaches were compared to random search and exhaustive search when it was feasible. Data is presented for experiments with $n = 9$ and $n = 16$. The results for other sizes have similar characteristics.

The first observation is that different search strategies behave differently. Figures 3 to 5 show the execution time and the distribution of each search case for random search, SPS, and MPS when $n = 9$. For random search, the execution time graph shows a random pattern and the distribution looks like a normal distribution. For MPS, the execution time converges to a small range near the best and the distribution concentrates on the fast side. It shows that the definition of "neighbor" in MPS is effective in controlling the range of the search. For SPS, the run-time also shows some kind of random pattern. However, from the distribution we can see, the run-times are more concentrated in the fast side. The random pattern is due to the use of R7(Random) in finding neighbors.

The second important observation is that a nearly optimal formula can be found while considering substantially fewer formulas than the total size of the search space. Recall that the size of the search space is $O(5.83^n)$. The number of samples in SPS and MPS to $O(n^2)$. Figure 6 shows at each step the best performance obtained since the beginning of the test. This illustrates how quickly the search methods converge on a formula with good performance.

The best formulas found by SPS and MPS are very close to the best one found in the exhaustive search (we only verified this for $n \leq 9$). For the random search, although a pretty good formula can be found quickly, the performance of the best formula found is not as good as those found by SPS and MPS. The difference gets larger when n increases.

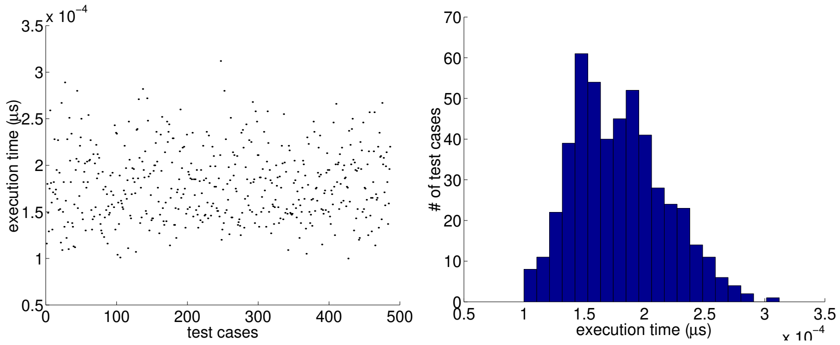

Fig. 3. Execution time(μs) and distribution, n=9, Random

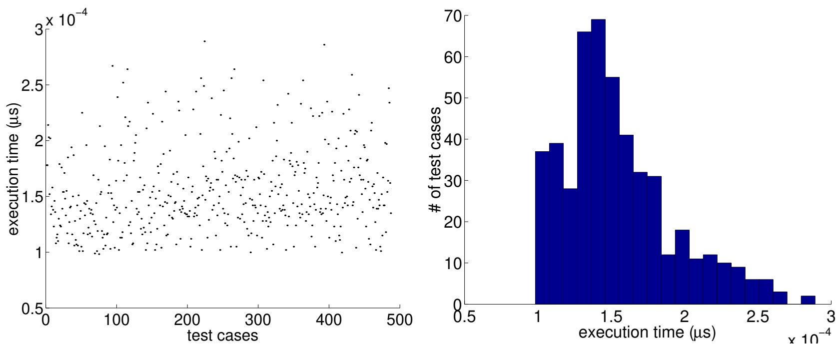

Fig. 4. Execution time(μs) and distribution, n=9, SPS

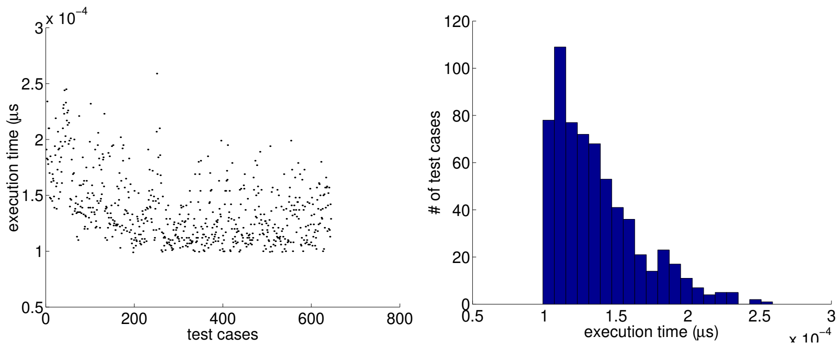

Fig. 5. Execution time(μs) and distribution, n=9, MPS

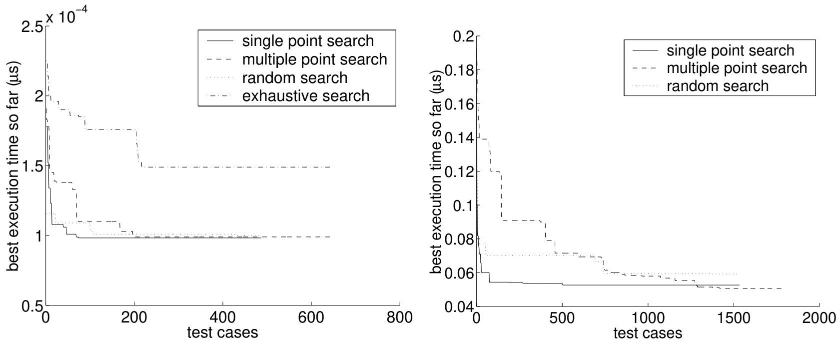

Fig. 6. Best execution time since the beginning (μs), n=9(left), 16(right)

These experiments show that search techniques can be used to obtain fast implementations of signal processing algorithms like the FFT. The mathematical approach presented in the paper sets the framework for a systematic search, and while an exhaustive search is infeasible, simple heuristic approaches allow nearly optimal implementations to be found in a reasonable amount of time.

Acknowledgements

The authors would like to thank the referees for their comments and suggestions.

References

1. L. Auslander, J. R. Johnson, and R. W. Johnson. Automatic implementation of FFT algorithms. Technical Report 96-01, Dept. of Math. and Computer Science, Drexel University, Philadelphia, PA, June 1996. Presented at the DARPA ACMP PI meeting.
2. J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. of Computation*, 19:297–301, 1965.
3. M. Frigo. A fast fourier transform compiler. In *PLDI '99*, pages 169–180, 1999.
4. 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`.
5. Johnson J., Johnson R., D Rodriguez, and R. Tolimieri. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures . *IEEE Trans. Circuits Sys.*, 9, 1990.
6. J. Johnson, R. Johnson, D. Padua, and J. Xiong. SPL: Signal Processing Language, 1999. `http://www.ece.cmu.edu/~spiral/SPL.html`.
7. J. R. Johnson and R. W. Johnson. Automatic generation and implementation of FFT algorithms. In *SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
8. J. R. Johnson and M. Püschel. In search of the optimal Walsh-Hadamard transform. In *Proc. ICASSP 2000*, 2000.

9. 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.
10. H. Massalin. Superoptimizer – a look at the smallest program. In *Proc. ASPLOS II*, pages 122–126, 1987.
11. J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso. SPIRAL: Portable Library of Optimized SP Algorithms, 1998. `http://www.ece.cmu.edu/~spiral/`.
12. K. R. Rao and P. Yip. *Discrete Cosine Transform*. Academic Press, 1990.
13. R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer, 2nd edition, 1997.
14. C. Van Loan. *Computational Framework of the Fast Fourier Transform*. Siam, 1992.
15. R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software (ATLAS), 1998. `http://www.netlib.org/atlas/`.