Formal Loop Merging for Signal Transforms

Franz Franchetti Yevgen Voronenko Markus Püschel

Department of Electrical and Computer Engineering
Carnegie Mellon University
{franzf, yvoronen, pueschel}@ece.cmu.edu

Abstract

A critical optimization in the domain of linear signal transforms, such as the discrete Fourier transform (DFT), is loop merging, which increases data locality and reuse and thus performance. In particular, this includes the conversion of shuffle operations into array reindexings. To date, loop merging is well understood only for the DFT, and only for Cooley-Tukey FFT based algorithms, which excludes DFT sizes divisible by large primes. In this paper, we present a formal loop merging framework for general signal transforms and its implementation within the SPIRAL code generator. The framework consists of Σ-SPL, a mathematical language to express loops and index mappings; a rewriting system to merge loops in Σ-SPL; and a compiler that translates Σ-SPL into code. We apply the framework to DFT sizes that cannot be handled using only the Cooley-Tukey FFT and compare our method to FFTW 3.0.1 and the vendor library Intel MKL 7.2.1. Compared to FFTW our generated code is a factor of 2–4 faster under equal implementation conditions (same algorithms, same unrolling threshold). For some sizes we show a speed-up of a factor of 9 using Bluestein's algorithm. Further, we give a detailed comparison against the Intel vendor library MKL; our generated code is between 2 times faster and 4.5 times slower.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, optimization, code generation; F.2.1 [*Analysis of Algorithms and Problem Complexity*]: Numerical Algorithms and Problems—computation of transforms; C.3 [*Special Purpose and Application-Based Systems*]—signal processing systems

General Terms Algorithms, design, languages, measurement, performance, theory

Keywords Linear signal transform, discrete Fourier transform, DFT, loop optimization, domain-specific language, automatic performance tuning

1. Introduction

Linear digital signal processing (DSP) transforms are the computational workhorses in DSP, occurring in practically every DSP application or standard. The most prominent example is the discrete Fourier transform (DFT), which, beyond DSP, is arguably among the most important numerical algorithms used in science and engineering. Typically, DSP transforms are used in applications that process large data sets or operate under realtime constraints, which poses the need for very fast software implementations.

It is meanwhile well-known that the design and implementation of highest performance numerical kernels for modern computers is a very difficult problem due to deep memory hierarchies, complex microarchitectures, and special instruction sets. Exacerbating the problem, optimal code is often platform-dependent, which increases the development cost considerably. DSP transforms are no exception. On the positive side, fast algorithms for DSP transforms are "divide-and-conquer," which generally produces a structure well-suited for good performance on memory hierarchies. On the negative side, the conquer step in these algorithms is iterative, i.e., requires multiple passes through the data. In particular, some of these steps are complicated shuffle operations, which can deteriorate performance considerably. As a consequence, one of the keys to obtaining high performance is to *merge* these iterative steps to improve data locality and reuse. In particular, the shuffle operations should not be performed explicitly, but converted into a reindexing in the subsequent computation. For fully unrolled code this optimization is straightforward, since the array accesses can be precomputed and inlined. In contrast, merging shuffle operations with loops is very difficult.

The general problem of loop merging [1, 6] is NP-complete. Further, loop merging requires array dependence information, for which the most general methods, like [7], achieve exact results only if the array indices are affine expressions, and even for this class the analysis has exponential worst-case runtime. Since DFT algorithms other than the Cooley-Tukey fast Fourier transform (FFT) use non-affine index mappings, standard array dependence tests do not work.

In the domain of signal transforms, the problem of loop merging has been solved to date only for the DFT and only for one DFT method: the Cooley-Tukey FFT (see [10]). Examples include the fastest available DFT software provided by vendor libraries[1], by the adaptable library FFTW [2, 4], and by the code generator SPIRAL [9, 8]. Consequently, these libraries achieve very high performance for DFT sizes that factor into very small prime numbers, but, as we demonstrate in this paper, can be far suboptimal for other sizes.

Contribution of this paper. This paper addresses the problem of automatically fusing loops and shuffle operations for arbitrary linear DSP transform algorithms. We propose a domain-specific approach consisting of three main components:

- A new language to symbolically represent transform algorithms called Σ-SPL. Σ-SPL is an extension to SPL [11], which is at

[1] For vendor libraries we can only speculate which optimizations are used, since the source code is not available.

PLDI'05, June 12–15, 2005, Chicago, Illinois,USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006…$5.00.

the core of the original SPIRAL system. As SPL, Σ-SPL is of mathematical nature, but makes loops and index mappings explicit. Σ-SPL borrows concepts from the early paper [5] on FFT manipulations.

- A rule-based framework to perform loop fusions on the Σ-SPL representation of a transform algorithm.
- A compiler that translates Σ-SPL into code.

Next, we apply this framework to the DFT and the four most important FFT methods and identify the necessary optimization rules specific to these algorithms. For the Cooley-Tukey FFT, the optimizations are equivalent to those performed in FFTW or in the original SPIRAL. For other FFT algorithms, namely the prime-factor FFT, Rader FFT, Bluestein FFT, or their combinations, the optimizations are novel.

We implemented our approach as a rewriting system within the SPIRAL code generator, which enables us to automatically perform these optimization as part of the code generation process. For DFT sizes that cannot be exclusively computed using the Cooley-Tukey FFT, we generate DFT code that is 2–4 times faster than FFTW on a Pentium 4 and, for some sizes, up to a factor of 9 when using Bluestein's FFT algorithm, which is not used in the current version of FFTW. We also provide a detailed comparison against the Intel vendor library MKL, for which the source code is not available. Here we observe anything from a speed-up of a factor of 2 to a slow-down of a factor of 4.5.

Organization of this paper. In Section 2 we review the four most important recursive DFT algorithms, discuss the problems in implementing them efficiently and review FFTW and SPIRAL. Section 3 motivates and describes the new language Σ-SPL. Section 4 describes the actual loop optimizations performed in Σ-SPL, including examples, and explains the implementation as a rewriting system within SPIRAL. In Section 5 we show benchmarks of SPIRAL generated DFT code using the new optimizations against FFTW and the Intel MKL. Finally, we offer conclusions in Section 6.

2. Background

In this section we first introduce the four most important recursive methods for computing the DFT and discuss the problems that arise in their efficient implementation. Then we explain how in FFTW and SPIRAL only one of these methods, namely the Cooley-Tukey FFT, is implemented efficiently. The other three recursions, which are necessary to handle *all* DFT sizes, are either implemented far suboptimally or not at all. Further, the optimization process for these algorithms is not even theoretically well understood. In this paper we solve this problem to generate code that is considerably faster compared to previous methods.

DFT. The DFT is a matrix-vector multiplication. Namely, if x is a complex input vector of length n, then the DFT of x is the vector $y = \mathrm{DFT}_n\, x$, where DFT_n is the $n \times n$ matrix

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

Recursive FFTs: Overview. Direct computation of the DFT requires $O(n^2)$ operations. However, several recursive methods, called fast Fourier transforms (FFTs), are available that reduce the cost to $O(n \log(n))$. We consider only the four most important FFTs, which are called Cooley-Tukey, prime-factor (or Good-Thomas), Rader, and Bluestein. Each of these FFTs reduces the problem of computing a DFT of size n to computing several DFTs of different (and with one exception smaller) sizes. The applicability of each FFT depends on the size n:

- Cooley-Tukey requires that $n = km$ factors and reduces the DFT_n to $m\ \mathrm{DFT}_k$'s and $k\ \mathrm{DFT}_m$'s.
- Prime-factor requires that $n = km$ factors and that the factors are coprime, $\gcd(k, m) = 1$. Similar to Cooley-Tukey, the DFT_n is then computed using $m\ \mathrm{DFT}_k$'s and $k\ \mathrm{DFT}_m$'s.
- Rader requires that $n = p$ is prime and reduces the DFT_p to 2 DFT_{p-1}'s.
- Bluestein is applicable to all sizes and computes a DFT_n using 2 DFT_m's of *larger* size $m \ge 2n - 1$.

Besides the recursion, the first three FFTs involve shuffle operations or permutations of the following forms. For Cooley-Tukey ($n = km$),

$$i \mapsto ki \bmod (n-1), \tag{1}$$

which can also be written without the modulo operation using two arguments i and j in the affine form

$$jm + i \mapsto ik + j. \tag{2}$$

For Good-Thomas and Rader, respectively,

$$\begin{aligned} i &\mapsto (k\lfloor \tfrac{i}{k} \rfloor + m(i \bmod k)) \bmod n, & (3)\\ i &\mapsto g^i \bmod n, & (4) \end{aligned}$$

and their inverses, where g is a suitably chosen, fixed integer.

For high performance it is crucial to handle these permutations, and their combinations as a DFT is computed recursively, efficiently. This means that ideally the permutations should not be performed explicitly, but translated into an array reindexing in the subsequent computation. For fully unrolled code this optimization is rather straightforward, since all array accesses can be precomputed and inlined. This is done, for example, in FFTW's codelet generator [3]. For loop optimizations, the "affine" permutation (1) or (2) has been studied extensively in the compiler literature and is well understood in the context of FFTs, whereas the more expensive prime-factor and Rader FFT mappings (3) and (4) have not received much attention. One main contribution of this paper is to identify the compiler transformations necessary to optimize these permutations. These transformations are not performed on the actual code, where they would be prohibitively expensive, but at a higher level of abstraction provided by Σ-SPL introduced in this paper.

Recursive FFTs: Details. We provide the explicit form of the four FFTs mentioned above similar to [10] in the form of structured sparse matrix factorization using the mathematical language called SPL in SPIRAL.

In the following, we use I_n to denote an $n \times n$ identity matrix, and

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

for the direct sum and tensor product of matrices, respectively.

The Cooley-Tukey, prime-factor, Rader, and Bluestein FFT, respectively, can be written as the following structured factorizations of the DFT matrix.

$$\begin{aligned} \mathrm{DFT}_n &= (\mathrm{DFT}_k \otimes \mathrm{I}_m) T_m^n (\mathrm{I}_k \otimes \mathrm{DFT}_m) L_k^n & (5)\\ \mathrm{DFT}_n &= V_n^{-1} (\mathrm{DFT}_k \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{DFT}_m) V_n, & (6)\\ \mathrm{DFT}_n &= W_n^{-1} (\mathrm{I}_1 \oplus \mathrm{DFT}_{p-1}) E_n (\mathrm{I}_1 \oplus \mathrm{DFT}_{p-1}) W_n, & (7)\\ \mathrm{DFT}_n &= B_n' D_m\, \mathrm{DFT}_m\, D_m'\, \mathrm{DFT}_m\, D_m'' B_n. & (8) \end{aligned}$$

Here, L, V, W are the permutation matrices corresponding to the permutations in (1), (3), (4), respectively; T, D, D', D'' are diagonal matrices, E is "almost" diagonal with 2 additional off-diagonal entries, B_n appends $m - n$ zeros to the input vector, and B_n' extracts the first n entries of a length m vector.

As said above, the first three recursions break down the DFT into smaller DFTs. The last, Bluestein, converts a DFT of size n into 2 larger DFTs of size $m \ge 2n - 1$. Usually, m is chosen in this case as a 2-power for fastest computation. This method does not involve shuffle operations and can serve as a fallback solution if the other methods become too slow.

We discuss Cooley-Tukey as an example. If $n = km$, then the DFT in (5) is computed in four steps (corresponding to the four factors in (5)). First, the input vector is shuffled according to the permutation matrix L_k^n; then, n DFT_m's are applied to subvectors of length m; then, the vector is scaled with T_k^n; and, finally, m DFT_k's are applied at stride m. This straightforward implementation would produce four loops and four passes through the data and leads to suboptimal performance.

The key to obtaining high performance with the above recursions is to reduce the number of passes through the data by fusing the loops to increase locality. In particular, the scaling steps (diagonal matrices) have to be fused with the adjacent loops arising from the tensor products, and the permutations should not be performed explicitly, but ideally be converted into reindexings in the subsequent loop. Since the above FFTs are applied recursively, diagonals and permutations occur at different levels of nesting, which makes these fusions a difficult problem. To date this problem has been solved only for algorithms that arise by exclusively using (5), for example, in FFTW and SPIRAL as discussed next. For other algorithms based on combinations of the other FFTs we present a solution in this paper.

FFTW. FFTW is a self-adaptable FFT library. For small DFT sizes, FFTW uses pregenerated, highly optimized, fully unrolled "codelets" [3]. The codelet generator uses a variety of FFT algorithms including Cooley-Tukey, prime-factor, and Rader. For large sizes, and thus loop code, FFTW has a built-in degree of freedom in recursing using Cooley-Tukey. A heuristic search selects at runtime the best recursion strategy (or algorithm), called "plan," for the given platform. The plan can then be used as many times as desired. FFTW implements Cooley-Tukey very efficiently. The permutation L is never explicitly performed, but passed as an argument in the recursion. This is possible, because of special properties of L. Similarly, scaling by T (the twiddle factors) is not performed as an extra step, but, also is passed down the recursion and finally performed by special "twiddle codelets." Thus, in a sense, the optimization of (5) is hardcoded into the infrastructure of FFTW. Besides (5), FFTW supports also (7), but there the permutations are performed explicitly, which results in poor performance. The other two FFTs are not supported.

SPIRAL. SPIRAL (see Figure 1) generates code for signal transforms of fixed size from scratch. In SPIRAL, the DFT recursions are called "rules" and included in the system in the form shown in (5)–(8). For a user-specified transform and size, SPIRAL applies these rules recursively until all transforms are of size 2 to generate one out of many possible algorithms, mathematically represented as SPL formula. The formula may then be optimized using formula manipulation. Next, the SPL compiler [11] translates the formula into optimized C code, using a template mechanism. The C code, in turn, is compiled and its runtime is measured. Based on the runtime, in a feedback loop, a search (or learning) engine triggers the generation of different algorithms. Iteration of this loop leads to a fast, platform-adapted implementation. For (5), SPIRAL performs optimizations equivalent to FFTW. Namely, when translating an SPL formula based on (5) into code, the SPL compiler fuses the twiddle factors and the stride permutation matrix a special purpose template for SPL expressions of the form $(\mathrm{I}_k \otimes A_m)L_k^{km}$ and $(A_k \otimes \mathrm{I}_m)T_m^{km}$. In other words, this optimization is hardcoded specifically for Cooley-Tukey based algorithms.

Summary. In summary, both FFTW and SPIRAL handle (5) as a special case, an approach that is neither easily extensible, nor one that gives any insight into how to optimize the other FFT recursions or other transforms. Since in SPIRAL the goal is to generate very fast code for all transform and all sizes, a new framework is needed, complementing but extending the original SPL and SPL compiler, to perform the necessary optimizations. This is the motivation for Σ-SPL presented in the next section. We then apply this framework to the FFTs (6), (7), and (8), to generate fast code for *all* DFT sizes.

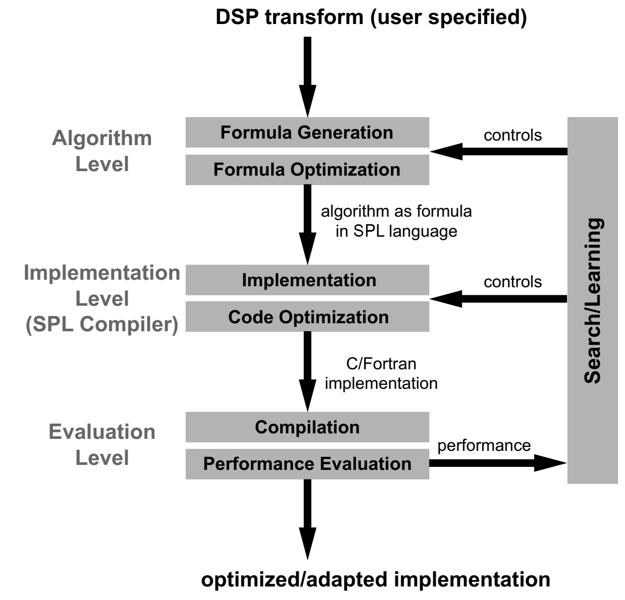

Figure 1. SPIRAL's architecture.

3. The Σ-SPL Language

In this section we define Σ-SPL after presenting the motivation and a short example.

The original SPL language used in SPIRAL describes transform algorithms as sparse structured matrix factorizations built from small matrices, permutation matrices, diagonal matrices, tensor products, and other constructs. SPL captures the data flow of an algorithm. However, all data reorganization steps are described explicitly as stages that perform passes through the data.

As an example, consider the SPL formula

$$(\mathrm{I}_m \otimes A_n)P, \quad P \text{ a permutation matrix,} \tag{9}$$

which is produced by the recursions (5) and (6). This formula describes two passes through the data. First, the data vector is shuffled according to P using a loop with mn iterations that just moves data. Second, a loop with m iterations applies the computational kernel A_n to subvectors of size n. One of our goals is to merge these two loops into one loop that implements the data reorganization given by P as readdressing of the input of the computational kernel A_n. This optimization cannot be expressed in SPL and is impractical, if not unfeasible, to perform on the corresponding C code. This is the motivation for introducing Σ-SPL, which makes the index mappings explicit and enables this optimization. We briefly show how this optimization is performed on (9) before we define Σ-SPL in detail.

Translating (9) into Σ-SPL yields

$$\left(\sum_{j=0}^{m-1} \mathrm{S}_{w_j}\, A_n\, \mathrm{G}_{r_j}\right) \mathrm{perm}\,(p), \tag{10}$$

where the so-called gather matrix G_{r_j} denotes the reading or loading of n input values according to the index mapping function r_j, and the scatter matrix S_{w_j} denotes the writing or storing of n output values according to the index mapping function w_j. Observe also that the permutation matrix is now expressed in terms of its defining permutation p. Intuitively, the permutation can be incorporated directly into the gather matrix by changing the gather index mapping. Below we give the optimized Σ-SPL formula for (9) with merged loops, obtained by applying a rewrite rule:

$$\sum_{j=0}^{m-1} \Big(\mathrm{S}_{w_j}\, A_n\, \mathrm{G}_{p \circ r_j} \Big). \tag{11}$$

The permutation was fused into the gather operation to yield the new gather index mapping $p \circ r_j$ ("$\circ$" is the composition of functions).

For efficient computation, it is further crucial to simplify the composed index mapping functions resulting from loop merging. This is also done by identifying a small set of rewrite rules that perform this simplification for the considered domain of transform algorithms. Identifying these rules for the FFTs (5)–(8) is one of the contributions of this paper.

To make explicit which loops in SPL formulas persist and which are merged, we divide the SPL constructs into two categories: *skeleton* and *decoration*.

Skeleton. The main loop structure of an SPL formula is defined by its *skeleton*. Examples of skeleton objects include direct sums and tensor products with identity matrices:

$$A \oplus B, \quad A \otimes \mathrm{I}_n, \quad \text{and} \quad \mathrm{I}_n \otimes A.$$

When an SPL formula is mapped into Σ-SPL, skeleton objects are translated into iterative sums including gather/scatter operations, which makes the loop structure and the index mappings explicit (as in (10)). In our motivating example (9) the skeleton is $\mathrm{I}_m \otimes A_n$.

Decoration. Permutation matrices and diagonal matrices are called *decorations*. We introduce the container constructs $\mathrm{perm}\,(p)$ and $\mathrm{diag}\,(f)$ to express loops originating from decorations. Our optimization merges these objects into the skeleton such that the extra stages disappear. In (9) the decoration is $P = \mathrm{perm}\,(p)$ and the gather index mapping resulting from the loop merging in (11) is $p \circ r_j$.

With the above example as a motivation, we now provide the formal definition of index mappings that we use and then define Σ-SPL.

3.1 Index Mappings

An important concept in Σ-SPL is the index mapping, concisely expressed by a function mapping an interval into an interval. For example, as we saw in (10), gather, scatter, and permutation matrices are parameterized by these functions. Further, we express index mapping functions in terms of primitive functions and function operators to capture their structure and thus enable all necessary simplifications, which would be exceedingly difficult on the corresponding C code.

Index mapping functions. We start with some definitions. An integer interval is denoted by

$$\mathbb{I}_n = \{0 \ldots, n-1\}.$$

An index mapping function f with domain $\mathbb{I}_n$ and range $\mathbb{I}_N$ is denoted by

$$f : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto f(i).$$

We use the short-hand notation $f^{n \to N}$ to refer to an index mapping function of the form $f : \mathbb{I}_n \to \mathbb{I}_N$.

Index mapping functions may depend on parameters. Assuming the parameter is j, we would write

$$f_j : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto f_j(i).$$

A bijective index mapping function

$$p : \mathbb{I}_n \to \mathbb{I}_n;\ i \mapsto p(i)$$

defines a permutation on n elements and is denoted by $p^{n \circlearrowleft}$.

To capture the structure of index mapping functions we use a few primitive functions and operators defined next.

Primitive index mapping functions. We define the *identity* index mapping function as

$$\imath_n : \mathbb{I}_n \to \mathbb{I}_n;\ i \mapsto i$$

and the *constant* function on the domain $\mathbb{I}_1$, parameterized by $0 \le j < n$, as

$$(j)_n : \mathbb{I}_1 \to \mathbb{I}_n;\ i \mapsto j.$$

In other words, $(j)_n$ maps 0 to j. We also define the *add constant* function, for $k \le N - n$, as

$$(k)_+^{n \to N} : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto i + k.$$

Function operators. Structured index mapping functions are built from the above primitives using function operators.

For the two index mapping functions

$$f : \mathbb{I}_m \to \mathbb{I}_M;\ i \mapsto f(i) \quad \text{and} \quad g : \mathbb{I}_n \to \mathbb{I}_N;\ i \mapsto g(i)$$

with $n = M$, we define the *function composition* in the usual way:

$$g \circ f : \mathbb{I}_m \to \mathbb{I}_N;\ i \mapsto g(f(i)).$$

Further, we define the *tensor product* of index mapping functions as

$$f \otimes g : \mathbb{I}_{mn} \to \mathbb{I}_{MN};\ i \mapsto Nf\left(\left\lfloor \tfrac{i}{n} \right\rfloor\right) + g(i \bmod n).$$

Intuitively, if $\mathbb{I}_{nm}$ is organized as an $m \times n$ array, then $f \otimes g$ applies f to its rows and g to its columns to obtain an $M \times N$ array which represents $\mathbb{I}_{MN}$.

Tensor products of $\imath_n$ and $(j)_m$ correspond to multi-linear index mapping functions. The simplest case with only two terms expresses strided (vector) access and unit stride access, respectively. Namely,

$$\imath_n \otimes (j)_m : \mathbb{I}_n \to \mathbb{I}_{mn};\ i \mapsto im + j, \tag{12}$$

$$(j)_m \otimes \imath_n : \mathbb{I}_n \to \mathbb{I}_{mn};\ i \mapsto i + jn. \tag{13}$$

3.2 Σ-SPL

Σ-SPL extends the original SPL with four new parameterized matrices that are described by their defining functions:

$$\mathrm{G}_{r^{n \to N}},\ \mathrm{S}_{w^{n \to N}},\ \mathrm{perm}\left(p^{n \circlearrowleft}\right),\ \text{and}\ \mathrm{diag}\left(f^{n \to \mathbb{C}}\right),$$

where $f^{n \to \mathbb{C}}$ is a function from $\mathbb{I}_n$ to $\mathbb{C}$. Further, we introduce the new matrix operator *iterative sum*,

$$\sum_{j=0}^{n-1} A_j.$$

In Σ-SPL the summands A_j of an iterative sum are constrained such that actual additions (except the ones incurred by the A_j) are never performed. This means that for every output index k, there is at most one matrix A_j that has non-zero entries in row k.

These constructs are now explained in detail, together with their interpretation as actual C code, which is straightforward, and summarized in Table 1. For completeness, we include the matrix product, which is part of standard SPL.

Product of Matrices. For $y = ABx$ first $t = Bx$ is computed and then $y = At$, leading to the first compilation rule in Table 1.

Iterative sums of matrices. The interpretation of the iterative sum as a loop makes use of the distributivity law:

$$\left(\sum_{j=0}^{n-1} A_j \right) x = \sum_{j=0}^{n-1} (A_j x). \tag{14}$$

$\mathrm{Code}(AB, y, x) \rightarrow \mathrm{Code}(B, t, x)\,;\mathrm{Code}(A, y, t)\,;$

$\mathrm{Code}\left(\sum_{j=0}^{k-1} A_j, y, x\right) \rightarrow$
    for(j=0; j<k; j++) $\mathrm{Code}(A_j, y, x)$;

$\mathrm{Code}(\mathrm{G}_{f^{n\to N}}, y, x) \rightarrow$
    for(j=0; j<n; j++) y[j] = x[f(j)];

$\mathrm{Code}(\mathrm{S}_{f^{n\to N}}, y, x) \rightarrow$
    for(j=0; j<n; j++) y[f(j)] = x[j];

$\mathrm{Code}(\mathrm{perm}\left(p^{n\circlearrowleft}\right), y, x) \rightarrow$
    for(j=0; j<n; j++) y[j] = x[p(j)];

$\mathrm{Code}(\mathrm{diag}\left(f^{n\to\mathbb{C}}\right), y, x) \rightarrow$
    for(j=0; j<n; j++) y[j] = f(j)*x[j];

Table 1. Translating Σ-SPL constructs to code; x denotes the input and y the output vector.

Due to the constraint that the iterative sum actually does not incur any additional operations, it encodes a loop where each iteration produces a non-overlapping part of the final output vector. Each summand in the iterative sum typically consists of three factors that encode three parts of the final program: 1) A gather matrix (details below) specifying the addresses for loading the input, 2) a computational kernel specifying the actual computation, and 3) a scatter matrix (details below) specifying the addresses for storing the results. The code for (14) is produced by the second rule in Table 1.

Gather matrices. Let $e_k^n \in \mathbb{C}^{n\times 1}$ be the canonical basis vector with entry 1 in position k and entry 0 elsewhere. An index mapping function $f^{n\to N}$ generates the gather matrix ($[\cdot]^\top$ is the matrix transposition)

$$\mathrm{G}_{f^{n\to N}} := \left[e_{f(0)}^N \mid e_{f(1)}^N \mid \cdots \mid e_{f(n-1)}^N\right]^\top.$$

This implies that for two vectors $x = (x_0, \ldots, x_{N-1})^\top$ and $y = (x_0, \ldots, x_{n-1})^\top$,

$$y = \mathrm{G}_{f^{n\to N}}\, x \quad \Leftrightarrow \quad y_i = x_{f(i)},$$

which explains the corresponding code produced by the third rule in Table 1.

Scatter matrices. An index mapping function $f^{n\to N}$ generates the scatter matrix

$$\mathrm{S}_{f^{n\to N}} := \left[e_{f(0)}^N \mid e_{f(1)}^N \mid \cdots \mid e_{f(n-1)}^N\right].$$

Scatter and gather matrices generated by the same function are transposes of each other. This introduces for two vectors $x = (x_0, \ldots, x_{n-1})^\top$ and $y = (x_0, \ldots, x_{N-1})^\top$ the identity

$$y = \mathrm{S}_{f^{n\to N}}\, x \quad \Leftrightarrow \quad y_j = \begin{cases} x_i & \text{if } j = f(i) \\ 0 & \text{else} \end{cases}.$$

Code for the matrix-vector product $y = \mathrm{S}_{f^{n\to N}}\, x$ when used within an iterative sum is shown by the fourth rule in Table 1. The elements set to zero can be omitted in this case since they do not contribute to the result.

Permutation matrices. A permutation matrix corresponding to its defining permutation $p^{n\circlearrowleft}$ is written as

$$\mathrm{perm}\left(p^{n\circlearrowleft}\right) := \left[e_{p(0)}^n \mid e_{p(1)}^n \mid \cdots \mid e_{p(n-1)}^n\right]^\top.$$

Permutation matrices are special cases of gather matrices with the constraint that the index mapping function must be bijective. Thus, the algorithm to implement gather matrices is used to implement permutation matrices.

Diagonal matrices. A function $f^{n\to\mathbb{C}} : \mathbb{I}_n \to \mathbb{C}$ defines the $n \times n$ diagonal matrix

$$\mathrm{diag}\left(f^{n\to\mathbb{C}}\right) := \mathrm{diag}\left(f(0), \ldots, f(n-1)\right).$$

The translation of the matrix-vector product $y = \mathrm{diag}\left(f^{n\to\mathbb{C}}\right)x$ to code is shown by the last rule in Table 1.

Example. As explained in the beginning of this section, we use iterative sums to make the loop structure of skeleton objects in SPL explicit. The summands are sparse matrices that depend on the summation index. By construction, the iterative sum does not lead to any additional arithmetic operations, since the elements of the original matrix are distributed among different matrix summands, and all the other summand elements are set to zero. For example,

$$\mathrm{I}_2 \otimes \mathrm{F}_2 \quad , \quad \mathrm{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

becomes in Σ-SPL the sum

$$\begin{bmatrix} \mathrm{F}_2 & 0_2 \\ 0_2 & \mathrm{F}_2 \end{bmatrix} = \begin{bmatrix} \mathrm{F}_2 & 0_2 \\ 0_2 & 0_2 \end{bmatrix} + \begin{bmatrix} 0_2 & 0_2 \\ 0_2 & \mathrm{F}_2 \end{bmatrix}. \qquad (15)$$

With the definition of the scatter and gather matrices

$$S_0 = \left[\,\mathrm{I}_2 \,|\, 0_2\,\right]^\top \quad \text{and} \quad S_1 = \left[\,0_2 \,|\, \mathrm{I}_2\,\right]^\top,$$
$$G_0 = \left[\,\mathrm{I}_2 \,|\, 0_2\,\right] \quad \text{and} \quad G_1 = \left[\,0_2 \,|\, \mathrm{I}_2\,\right],$$

(15) can be written as the iterative sum

$$\sum_{j=0}^{1} S_j\, \mathrm{F}_2\, G_j. \qquad (16)$$

However, in (16) the subscripts of S and G are integers and not functions. Using the identity function, the constant function, and the tensor product for functions (see Section 3.1), we express the gather and scatter matrices as

$$S_j = \mathrm{S}_{(j)_2 \otimes \imath_2} \quad \text{and} \quad G_j = \mathrm{G}_{(j)_2 \otimes \imath_2}.$$

The matrix-vector product $y = (\mathrm{I}_2 \otimes \mathrm{F}_2)x$ now becomes in Σ-SPL

$$y = \sum_{j=0}^{1} \mathrm{S}_{(j)_2 \otimes \imath_2}\, \mathrm{F}_2\, \mathrm{G}_{(j)_2 \otimes \imath_2}\, x.$$

Using the rules in Table 1 we obtain the following unoptimized program:

```
// Input: _Complex double x[4], output: y[4]
_Complex double t0[2], t1[2];
for (int j=0;j<2;j++) {
  for (int i=0; i<2; i++) t0[i] = x[i+2*j];
  t1[0] = t0[0] + t0[1];
  t1[1] = t0[0] - t0[1];
  for (int i=0; i<2; i++) y[i+2*j] = t1[i];
}
```

After standard optimizations (performed by the standard SPL compiler), such as loop unrolling, array scalarization, and copy propagation, we obtain the following optimized program:

```
// Input: _Complex double x[4], output: y[4]
for (int j=0;j<2;j++) {
  y[2*j]   = x[2*j] + x[2*j+1];
  y[2*j+1] = x[2*j] - x[2*j+1];
}
```

4. The Σ-SPL Rewriting System

In this section we describe the new loop optimization procedure and its implementation in SPIRAL. The optimizations are implemented as a series of rewriting systems operating on SPL and Σ-SPL expression trees. An overview of the different steps is shown in Figure 2, which is inserted in the formula optimization block in

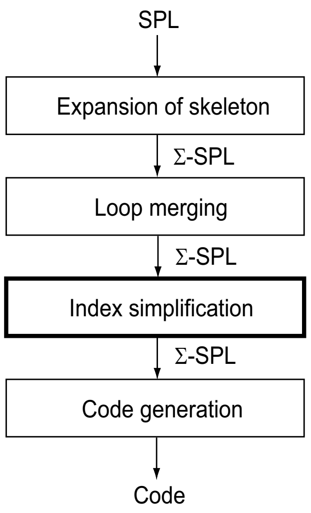

Figure 2. Translating SPL to optimized code.

SPIRAL (see Figure 1). The steps are generic for all transforms and algorithms except the index simplification (bold box), which requires the inclusion of rules specific to the class of algorithms considered. We start with an overview; then we explain the steps in detail.

Expansion of skeleton. In the first step we translate an SPL formula (as generated within SPIRAL) into a corresponding Σ-SPL formula. The skeleton (see Section 3) is expanded into iterative sums and the decorations are expressed in terms of their defining functions.

Loop merging. A generic set of rules merges the decorations into adjacent iterative sums, thus effectively merging loops. In this process index mapping functions are symbolically composed, and thus become complicated or costly to compute.

Index simplification. In this stage the index mapping functions are simplified, which is possible due to their symbolic representation and a set of symbolic rules. The rules of this stage are transform dependent and encode the domain specific knowledge how to handle specific permutations. Most identities are based on number theoretic properties of the permutations. Identifying these rules for a given transform is usually a research problem.

Code generation. After the structural optimization the Σ-SPL compiler is used to translate the final expression into initial code (see Table 1), which is then further optimized as in the original SPL compiler [11].

Running example. In the following detailed explanation, we use the following simple formula, a special case of (9), as a running example

$$(\mathrm{I}_m \otimes \mathrm{DFT}_n)\, L^{mn}_m, \quad L^{mn}_m = \mathrm{perm}\left(\ell^{mn}_m\right), \tag{17}$$

with $\ell^{mn}_m : \mathbb{I}_{mn} \to \mathbb{I}_{mn}$, mapping

$$i \mapsto \begin{cases} (im) \bmod (mn-1) & \text{if } i < mn-1, \\ mn-1 & \text{if } i = mn-1. \end{cases} \tag{18}$$

The direct translation of (17) into code for $m = 5$ and $n = 2$ would lead to the following code:

```
// Input: _Complex double x[10], output: y[10]
_Complex double t[10];
// explicit stride permutation L^10_5
for (int i=0; i<10; i++)
   t[i] = x[i<9 ? (i*5)%9 : 9];
// kernel loop I_5 x DFT_2
for (int i=0; i<5; i++) {
   y[2*i]   = t[2*i] + t[2*i+1];
   y[2*i+1] = t[2*i] - t[2*i+1];
}
```

The code contains two loops, originating from the permutation matrix L^{mn}_m and from the computational kernel $\mathrm{I}_m \otimes \mathrm{DFT}_n$. Note that optimized methods to implement the stride permutation L^{mn}_m using two nested loops can be found in [10]. However, we do not resort to such an implementation as this treats the stride permutation as a special case and we aim at solving the general problem. The goal in this example is to merge the two loops into one.

$$\begin{aligned} A \oplus B \quad &\to \quad \mathrm{S}_{(0)^{m\to m+m'}_{+}}\, A\, \mathrm{G}_{(0)^{n\to n+n'}_{+}} + \\ &\qquad \mathrm{S}_{(m)^{m'\to m+m'}_{+}}\, A\, \mathrm{G}_{(n)^{n'\to n+n'}_{+}} && (19)\\ A \otimes \mathrm{I}_k \quad &\to \quad \sum_{j=0}^{k-1} \mathrm{S}_{\imath_m \otimes (j)_k}\, A\, \mathrm{G}_{\imath_n \otimes (j)_k} && (20)\\ \mathrm{I}_k \otimes A \quad &\to \quad \sum_{j=0}^{k-1} \mathrm{S}_{(j)_k \otimes \imath_m}\, A\, \mathrm{G}_{(j)_k \otimes \imath_n} && (21) \end{aligned}$$

Table 2. Rules to expand the skeleton.

4.1 Expansion of Skeleton

This stage translates SPL formulas, generated by SPIRAL, into Σ-SPL formulas. It translates all skeleton objects in the SPL formula into iterative sums and makes the defining functions of decorations explicit. Table 2 summarizes the rewrite rules used in this step, assuming $A \in \mathbb{C}^{m\times n},\ B \in \mathbb{C}^{m'\times n'}$.

The tensor product structure of matrices is in Σ-SPL captured through the tensor product structure of the gather and scatter index mapping functions. The add constant function is used in the conversion of direct sums of matrices.

In our example (17), the rewriting system applies the rule (21) to obtain the Σ-SPL expression

$$\left(\sum_{j=0}^{m-1} \mathrm{S}_{(j)_m \otimes \imath_n}\, \mathrm{DFT}_n\, \mathrm{G}_{(j)_m \otimes \imath_n}\right) \mathrm{perm}\left(\ell^{mn}_m\right). \tag{22}$$

4.2 Loop Merging

The goal of loop merging is to propagate all index mapping functions for a nested sum into the parameter of one single gather and scatter matrix in the innermost sum, and to propagate all diagonal matrices into the innermost computational kernel. Table 3 summarizes the necessary rules. Loop merging consists of two steps: moving matrices into iterative sums, and actually merging or commuting matrices.

First, matrices are moved inside iterative sums by applying rules (24) and (25) that implement the distributivity law. Note, that iterative sums are not moved into other iterative sums.

Second, the system merges gather, scatter, and permutation matrices using rules (26)–(29) and pulls diagonal matrices into the computational kernel using (30) and (31). This step simply composes defining functions, possibly creating complicated terms that must be simplified in the next step.

After loop merging, (22) is transformed into

$$\sum_{j=0}^{m-1} \left(\mathrm{S}_{(j)_m \otimes \imath_n}\, \mathrm{DFT}_n\, \mathrm{G}_{\ell^{mn}_m \circ \left((j)_m \otimes \imath_n\right)}\right). \tag{23}$$

$$\left(\sum_{j=0}^{m-1} A_j\right) M \quad\rightarrow\quad \left(\sum_{j=0}^{m-1} A_j M\right) \tag{24}$$

$$M\left(\sum_{j=0}^{m-1} A_j\right) \quad\rightarrow\quad \left(\sum_{j=0}^{m-1} M A_j\right) \tag{25}$$

$$\mathrm{G}_{s^{n\to N_1}}\,\mathrm{G}_{r^{N_1\to N}} \quad\rightarrow\quad \mathrm{G}_{r\circ s} \tag{26}$$

$$\mathrm{S}_{v^{N_1\to N}}\,\mathrm{S}_{w^{n\to N_1}} \quad\rightarrow\quad \mathrm{S}_{v\circ w} \tag{27}$$

$$\mathrm{G}_{r^{n\to N}}\,\mathrm{perm}\left(\pi^{N\circlearrowleft}\right) \quad\rightarrow\quad \mathrm{G}_{\pi\circ r} \tag{28}$$

$$\mathrm{perm}\left(\pi^{N\circlearrowleft}\right)\mathrm{S}_{w^{n\to N}} \quad\rightarrow\quad \mathrm{S}_{\pi^{-1}\circ w} \tag{29}$$

$$\mathrm{G}_{r^{n\to N}}\,\mathrm{diag}\left(f^{N\to\mathbb{C}}\right) \quad\rightarrow\quad \mathrm{diag}\left(f\circ r\right)\mathrm{G}_r \tag{30}$$

$$\mathrm{diag}\left(f^{N\to\mathbb{C}}\right)\mathrm{S}_{w^{n\to N}} \quad\rightarrow\quad \mathrm{S}_w\,\mathrm{diag}\left(f\circ w\right) \tag{31}$$

Table 3. Loop merging rules.

4.3 Index Mapping Simplification

As said before, this is the only step that depends on the considered transform and its algorithms. We consider the DFT and the FFTs (5)–(7). These recursions involve permutations that involve integer power computations, modulo operations, and conditional computations. In addition to the stride permutation (18) in the Cooley-Tukey decomposition (5), the prime-factor decomposition (6) uses the permutation matrix $V_{r,s} = \mathrm{perm}\left(v^{r,s}\right)$, for $\gcd(r,s) = 1$, with

$$v^{r,s} \;:\; \mathbb{I}_{rs} \;\to\; \mathbb{I}_{rs};\; i \;\mapsto\; \left(s\left\lfloor \tfrac{i}{s}\right\rfloor + r(i \bmod s)\right) \bmod rs.$$

For prime n, the Rader decomposition (7) requires an exponentiation permutation matrix $W_{r,s} = \mathrm{perm}\left(w^n_{1,g}\right)$, where g is a generator of the multiplicative group $\mathbb{Z}_n^\times$, and

$$w^n_{\varphi,g} : \mathbb{I}_n \to \mathbb{I}_n;\; i \mapsto \begin{cases} 0 & \text{if } i = 0,\\ \varphi g^{i-1} \bmod n & \text{else.}\end{cases}$$

Thus, our rewriting system requires powerful index mapping function simplifications, as the previous steps merge the already complicated index mappings from each stage into one *very* complicated index mapping function in the innermost nested sum.

We express the index mappings using symbolic functions like $\imath_n$, $(j)_m$, and $(k)_+^{n\to N}$ as well as the tensor product and function composition to enable simplification. In order to handle (6) and (7) we have to introduce three helper functions,

$$h^{n\to N}_{b,s} : \mathbb{I}_n \to \mathbb{I}_N;\; i \mapsto b + is, \quad s|N, \tag{32}$$

$$\overline{h}^{n\to N}_{b,s} : \mathbb{I}_n \to \mathbb{I}_N;\; i \mapsto b + is \bmod N, \quad s|N, \tag{33}$$

$$\overline{w}^{n\to N}_{\varphi,g} : \mathbb{I}_n \to \mathbb{I}_N;\; i \mapsto \varphi g^i \bmod N. \tag{34}$$

The power of using symbolic functions and operators becomes apparent next. Namely, we can identify a rather small set of context insensitive simplification rules to simplify all index functions arising from combining the FFT rules (5)–(7). The required simplifications cannot be done solely using basic integer identities as conflicting identities and special number-theoretical constraints (e.g., a variable is required to be the generator of a cyclic group of order n) would be required. Our function symbols capture these conditions by construction while our rules encode the constraints.

Table 4 summarizes the most important index simplification rules. Rules (35)–(39) are used to simplify Cooley-Tukey FFT decompositions. Rules (40)–(42) are used for decompositions based on the Cooley-Tukey FFT and other FFTs. Rules (43)–(46) are used for the Rader FFT and (47) for the prime-factor FFT.

$$\ell^{mn}_m \circ \left(\left(j\right)_m \otimes f^{k\to n}\right) \quad\rightarrow\quad f^{k\to n}\otimes\left(j\right)_m . \tag{35}$$

$$\left(\ell^{mn}_m\right)^{-1} \quad\rightarrow\quad \ell^{mn}_n \tag{36}$$

$$\left(f^{1\to m}\otimes h\right)\circ g \quad\rightarrow\quad f\otimes\left(h\circ g\right) \tag{37}$$

$$\left(h\otimes g^{1\to n}\right)\circ f \quad\rightarrow\quad \left(h\circ f\right)\otimes g \tag{38}$$

$$\left(f_0\otimes f_1\right)\circ\left(g_0\otimes g_1\right) \quad\rightarrow\quad \left(f_0\circ g_0\right)\otimes\left(f_1\circ g_1\right) \tag{39}$$

$$\imath_n \quad\rightarrow\quad h^{n\to n}_{0,1} \tag{40}$$

$$f^{m\to M}\otimes g^{1\to N} \quad\rightarrow\quad h^{M\to MN}_{g(0),N}\circ f \tag{41}$$

$$g^{1\to N}\otimes f^{m\to M} \quad\rightarrow\quad h^{M\to MN}_{Mg(0),1}\circ f \tag{42}$$

$$w^N_{\varphi,g}\circ(0)^{1\to N}_+ \quad\rightarrow\quad (0)^{1\to N}_+ \tag{43}$$

$$w^N_{\varphi,g}\circ(N-1)^{N-1\to N}_+ \quad\rightarrow\quad \overline{w}^{N-1\to N}_{\varphi,g} \tag{44}$$

$$\overline{w}^{N'\to N}_{\varphi,g}\circ h^{n\to N'}_{b,s} \quad\rightarrow\quad \overline{w}^{n\to N}_{\varphi g^b, g^s} \tag{45}$$

$$\overline{w}^{N'\to N}_{\varphi,g}\circ \overline{h}^{n\to N'}_{b,s} \quad\rightarrow\quad \overline{w}^{n\to N}_{\varphi g^b, g^s} \tag{46}$$

$$v^{r,s}\circ h^{s\to rs}_{b,1} \quad\rightarrow\quad \overline{h}^{s\to rs}_{b,r} \tag{47}$$

Table 4. Index function simplification rules.

The identification of these rules is one of the main contributions of this paper.

In our example, the index function simplification applies rule (35) to (23) to obtain

$$\sum_{j=0}^{m-1}\left(\mathrm{S}_{(j)_m\otimes\imath_n}\,\mathrm{DFT}_n\,\mathrm{G}_{\imath_n\otimes(j)_m}\right). \tag{48}$$

4.4 Code Generation

The Σ-SPL compiler first generates unoptimized code for an Σ-SPL formula using the context insensitive mappings given in Table 1. As in the original SPL compiler [11], an unrolling parameter B to the Σ-SPL compiler controls which loops in a Σ-SPL formula will be fully unrolled and thus become basic blocks in which all decorations and index mappings are inlined. Our approach relies on further basic block level optimizations to produce efficient code. These are a superset of the optimizations implemented in the original SPL compiler, including 1) full loop unrolling of computational kernels, 2) array scalarization, 3) constant folding, 4) algebraic strength reduction, 5) copy propagation, 6) dead code elimination, 7) common subexpression elimination, 8) loop invariant code motion, and 9) induction-variable optimizations.

Translating our running example (48) into optimized code using the Σ-SPL compiler leads to the following:

```
// Input: _Complex double x[10], output: y[10]
for (int j=0; j<5; j++) {
  y[2*j]   = x[j] + x[j+5];
  y[2*j+1] = x[j] - x[j+5];
}
```

4.5 Rader and Prime-Factor Example

To demonstrate how our framework applies to more complicated SPL formulas, we show the steps for compiling a part of a 3-level recursion for a non-power-of-2 size DFT that uses all three different recursion rules (5)–(7), namely a DFT_{pq} with q prime, and $q-1 = rs$. Initially the prime-factor decomposition (6) is applied for the size pq, then the Rader decomposition (7) decomposes the prime size q into $q-1$. Finally, a Cooley-Tukey step (5) is used to decompose $q-1$ into rs. We only discuss a fragment of the resulting SPL formula, namely

$$(\mathrm{I}_p \otimes (\mathrm{I}_1 \oplus (\mathrm{I}_r \otimes \mathrm{DFT}_s) L_r^{rs})\, W_q)\, V_{p,q},$$

which is also shown in (49) in Table 5. This formula has three different permutations, and a naive implementation would require three explicit shuffle operations, leading to three extra passes through the data vector.

Our rewriting system merges these permutation matrices into the innermost loop and then simplifies the index mapping function, effectively reducing the number of necessary mod operations to approximately 3 mods per 2 data points and improving the locality of the computation at the same time. We discuss the intermediate steps following Figure 2.

Expansion of skeleton. Rules (21) and (19) expand the skeleton to produce the unoptimized Σ-SPL formula (50) in Table 5. A direct compilation for $p = 4$, $q = 7$, $r = 3$, and $s = 2$ would result in the following code.

```
00 // Input: _Complex double x[28], output: y[28]
01 _Complex double t1[28];
02 // permutation v^(4,7)
03 for(int i5 = 0; i5 <= 27; i5++)
04   t1[i5] = x[(3*i5 + 8*(i5%7))%28];
05 // iterative sum
06 for(int i1 = 0; i1 <= 3; i1++) {
07   _Complex double t3[7], t4[7], t5[7];
08   // gather
09   for(int i6 = 0; i6 <= 6; i6++)
10     t5[i6] = t1[7*i1 + i6];
11   // permutation w^7
12  for(int i8 = 0; i8 <= 6; i8++)
13    t4[i8] = t5[i8 ? pow(3, i8-1)%7 : 0];
14  // gather, permutation i_1, scatter
15  t3[0] = t4[0];
16  { _Complex double t10[6], t11[6], t12[6];
17    // gather
18    for(int i13 = 0; i13 <= 5; i13++)
19      t12[i13] = t4[i13 + 1];
20    // permutation l^6_3
21    for(int i14 = 0; i14 <= 5; i14++)
22      t11[i14] = t12[i14/2 + 3*(i14%2)];
23    // iterative sum
24    for(int i3 = 0; i3 <= 2; i3++) {
25      _Complex double t14[2], t15[2];
26      // gather
27      for(int i15 = 0; i15 <= 1; i15++)
28        t15[i15] = t11[2*i3 + i15];
29      // t14 = DFT_2*t15
30      t14[0] = t15[0] + t15[1];
31      t14[1] = t15[0] - t15[1];
32      // scatter
33      for(int i17 = 0; i17 <= 1; i17++)
34        t10[2*i3 + i17] = t14[i17];
35    }
36    // scatter
37    for(int i19 = 0; i19 <= 5; i19++)
38      t3[i19 + 1] = t10[i19];
39  }
40  // scatter
41  for(int i20 = 0; i20 <= 6; i20++)
42    y[7*i1 + i20] = t3[i20];
43 }
```

The code contains explicit data permutations (e.g., lines 3/4 and 12/13), has multiple iterative stages as reflected by multiple temporary vectors, and contains expensive power (line 13) and mod operations. For example, the index computation in line 4 involves two nested mod operations.

Loop merging. The rules in Table 3 are used to merge the loops and thus move the decorations into the innermost sum. The resulting Σ-SPL formula is shown in (51) in Table 5. Note that the occurring index functions are very complicated.

Index simplification. Using only the rules in Table 4 the gather and scatter index mapping functions are simplified to produce the optimized formula shown in (52) in Table 5.

Code generation. Now the code can be generated for some specific values of p, q, r and s using the Σ-SPL compiler. For the same parameters as above, $p = 4$, $q = 7$, $r = 3$, $s = 2$, we get:

```
00  // Input: _Complex double x[28], output: y[28]
01  int p1, b1;
02  for(int j1 = 0; j1 <= 3; j1++) {
03     y[7*j1] = x[(7*j1%28)];
04     p1 = 1; b1 = 7*j1;
05     for(int j0 = 0; j0 <= 2; j0++) {
06        y[b1 + 2*j0 + 1] =
07           x[(b1 + 4*p1)%28] + x[(b1 + 24*p1)%28];
08        y[b1 + 2*j0 + 2] =
09           x[(b1 + 4*p1)%28] - x[(b1 + 24*p1)%28];
10        p1 = (p1*3)%7;
11     }
12  }
```

Each index computation involves now at most one mod operation. Data is never explicitly copied between buffers and the computation is fully recursive. Note that simple optimizations like precomputing `b1+4*p1` in lines 7 and 9 in each iteration are left to the compiler.

4.6 Verification

A very important advantage of doing optimization at the formula level is the possibility of exact verification. Since a Σ-SPL formula still represents a sparse matrix factorization, it can always be converted, in the SPIRAL environment, into the dense transform matrix it represents and compared against the transform definition. For relatively small sizes it is thus feasible to perform verification after each rule application, which makes it possible to pinpoint the rule that leads to a possibly invalid formula—an important feature in our context of rather involved manipulations.

Beyond formula and rule verification, the presented approach benefits from SPIRAL's general code verification routines that are automatically applicable [8].

5. Experimental Results

In this section we show runtime results of SPIRAL generated DFT code using our new loop optimization approach. As said before, our focus is on DFT sizes that cannot be computed using exclusively the Cooley-Tukey FFT (5), since for these sizes, close-to-optimal code is already available from vendor libraries, FFTW, or the original SPIRAL. In other words, we focus on DFT sizes that require at least one Rader step (7) that is not unrolled in the recursion. This is the case if and only if the DFT size n has a prime divisor $p|n$ that is larger than the unrolling threshold B, i.e., $p > B$. Further, since the Rader step is far more expensive than a prime-factor or Cooley-Tukey step, we divide the set of all numbers into *levels*, depending on how many Rader steps are needed. This is captured in the following definition.

Definition 1 Let $B > 0$ be given (the unrolling threshold). With respect to B, a prime number p is called *level* 0 if $p \leq B$, and *level* i, $i > 0$, if the largest level of all prime divisors of $p-1$ is $i-1$. An integer is called level i, if the largest level of all its prime divisors is i.

For example, if $B = 16$, then $n = 11, n = 33 = 3 \cdot 11$ are level 0, $n = 17, n = 323 = 17 \cdot 19$ are level 1, and $n = 47$ is level 2, since $23|(47-1)$ and 23 is level 1.

$$(\mathrm{I}_p \otimes (\mathrm{I}_1 \oplus (\mathrm{I}_r \otimes \mathrm{DFT}_s) L_r^{rs})\, W_q)\, V_{p,q} \tag{49}$$

$$\sum_{j_1=0}^{p-1} \Bigg(\mathrm{S}_{(j_1)_p \otimes \imath_q} \Bigg(\mathrm{S}_{(0)_+^{1\to q}} \,\mathrm{perm}(\imath_1)\, \mathrm{G}_{(0)_+^{1\to q}} + \mathrm{S}_{(1)_+^{q-1\to q}} \Bigg(\sum_{j_0=0}^{r-1} \mathrm{S}_{(j_0)_r \otimes \imath_s} \,\mathrm{DFT}_s\, \mathrm{G}_{(j_0)_r \otimes \imath_s} \Bigg) \mathrm{perm}(\ell_r^{q-1})\, \mathrm{G}_{(1)_+^{q-1\to q}} \Bigg) \mathrm{perm}(w_{1,g}^q)\, \mathrm{G}_{(j_1)_p \otimes \imath_q} \Bigg) \mathrm{perm}(v^{p,q}) \tag{50}$$

$$\sum_{j_1=0}^{p-1} \Bigg(\mathrm{S}_{\left((j_1)_p \otimes \imath_q\right) \circ (0)_+^{1\to q} \circ \imath_1} \,\mathrm{G}_{v^{p,q} \circ \left((j_1)_p \otimes \imath_q\right) \circ w_{1,g}^q \circ (0)_+^{1\to q}} + \sum_{j_0=0}^{r-1} \mathrm{S}_{\left((j_1)_p \otimes \imath_q\right) \circ (1)_+^{q-1\to q} \circ \left((j_0)_r \otimes \imath_s\right)} \,\mathrm{DFT}_s\, \mathrm{G}_{v^{p,q} \circ \left((j_1)_p \otimes \imath_q\right) \circ w_{1,g}^q \circ (1)_+^{q-1\to q} \circ \ell_r^{rs} \circ \left((j_0)_r \otimes \imath_s\right)} \Bigg) \tag{51}$$

$$\sum_{\substack{j_1=0 \\ b_1=qj_1}}^{p-1} \Bigg(\mathrm{S}_{h_{0,q}^{p\to pq} \circ (j_1)_p} \,\mathrm{G}_{\overline{h}_{0,q}^{p\to pq} \circ (j_1)_p} + \sum_{\substack{j_0=0 \\ \phi_1=g^{j_0}}}^{r-1} \mathrm{S}_{h_{qj_1+sj_0+1,1}^{s\to pq}} \,\mathrm{DFT}_s\, \mathrm{G}_{\overline{h}_{b_1,p}^{q\to pq} \circ \overline{w}_{\phi_1,g^s}^{s\to q}} \Bigg) \tag{52}$$

Table 5. Loop merging example for the formula fragment (49), arising from a combination of prime-factor, Rader, and Cooley-Tukey FFT. The steps follow Figure 2: After expansion of the skeleton (50); after loop merging (51); and the final Σ-SPL formula after index simplification (52).

Intuitively, if a DFT of size n is recursively expanded into a tree using the three recursions (5), (6), and (7), then the level of n is the number of levels in this tree that contain a Rader step. Since Rader is expensive, a higher level will imply a lower relative performance.

If an unrolling threshold B is chosen, then the DFTs that can be computed exclusively using Cooley-Tukey are precisely those with a level 0 size. For $B = 16$, the numbers $1 \leq n \leq 1024$ divide into levels as follows; only about one quarter are level 0.

level	0	1	2	3	4
how many	245	536	205	36	1

Experimental setup. We ran the experiments on a Pentium 4, processor number 560, 3.6 GHz, under Windows XP using the Intel C++ compiler 8.0 with flags /Qc99 /O3 /Qrestrict /QxKWP. We considered only double precision code and measured the runtime of Intel's MKL 7.2.1, FFTW 3.0.1, and SPIRAL generated code using the new optimization approach. For best performance it is necessary to use short vector instructions. The considered Pentium 4 provides for double precision the instruction set extensions SSE2 and SSE3. Both provide 2-way vector instructions and SSE3 is a superset of SSE2, designed specifically to map complex arithmetic efficiently. The MKL provides code optimized for Pentium 4 using SSE2 or SSE3, FFTW provides scalar and SSE2 code, and with SPIRAL we generated scalar code and SSE3 code. The SSE3 code was obtained by generating complex C99 code and using compiler vectorization (flag /QxKWP). The advantage of this method is that it can be applied regardless of the DFT size. The disadvantage is that for sizes that are divisible by 4 (square of the vector length) we obtain roughly 20–30% slower code than with the SSE2 code generated using our vectorization method in [8]. We chose SSE3 since it fits seamlessly into the new Σ-SPL framework and does not require additional vectorization effort, thus allowing us to focus on the analysis of the new optimization approach.

5.1 SPIRAL vs. FFTW

In Figure 3 we compare the runtime of FFTW and SPIRAL generated code. Since the default installation of FFTW provides unrolled basic blocks (codelets) up to size 16 (and for 32 and 64), we chose for SPIRAL too, an unrolling threshold of $B = 16$ for fair comparison. As said above, we consider sizes of level 1 and higher with respect to this B.

For FFTW (dashed lines), we measured scalar and SSE2 code (provided by FFTW), in both cases using FFTW's search for the best recursion tree. For SPIRAL, we generated scalar code and SSE3 code (solid black lines), in both cases only using the FFTs (5)–(7). Finally, we also generated code with SPIRAL using Bluestein (8) (solid gray line), again as complex C99 code using compiler vectorization for SSE3.

Level 1 sizes. For sizes of level 1 (Figure 3, top) and scalar code (bullets), SPIRAL code is between 1.25 and 2 times faster than FFTW. FFTW's SSE2 code is only marginally faster than its scalar code since most of the time is spent in the costly Rader permutations. SPIRAL's generated SSE3 code (vectorized by the compiler from explicit complex C99 code) consistently gains about another 50% in performance, independent of the size, yielding a total improvement of a factor of 2–3 over FFTW. The Bluestein method (gray line) is not competitive for almost all level 1 numbers. Note that the Bluestein method (8) is about constant within intervals $2^k < n < 2^{k+1}$, since all DFTs of these sizes are computed using two DFTs of size 2^{k+2}.

Level 2 sizes. For level 2 sizes, we observe, as expected because of the two Rader steps, a larger gap between FFTW and SPIRAL generated code, which can now be as much as a factor of 3 for scalar code, and a factor of 4 for vector code. Bluestein is still inferior for most sizes.

Level 3 sizes. For level 3 sizes, the gap between FFTW and SPIRAL generated code further widens, and here Bluestein is faster in all cases, gaining up to a factor of 9 over FFTW.

Summary. The experiments validate that our method for loop merging produces significant performance gains for DFT sizes of level 1 and higher. Further, by generating explicit complex code and vectorizing for SSE3, we obtain another consistent performance gain of 50%. The experiments show that for DFT sizes of level 1 and 2 the recursive FFTs are preferable, whereas for levels 3 and higher Bluestein is faster. The search mechanism in SPIRAL will determine this automatically.

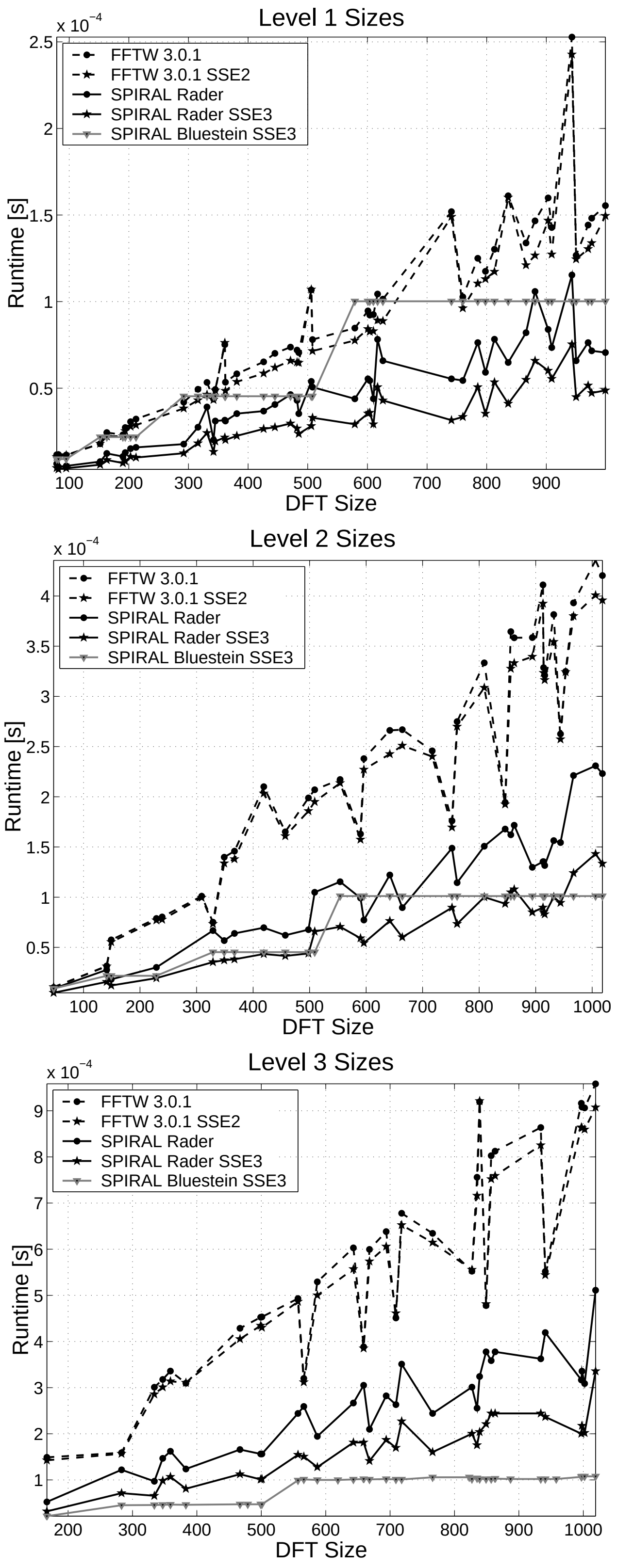

Figure 3. DFT runtime results for sizes between 100 and 1000 of level 1 (top), 2 (middle), and 3 (bottom): FFTW (dotted) and SPIRAL generated (solid). Lower is better.

5.2 SPIRAL vs. Intel MKL

We compare SPIRAL against Intel's MKL. This time we only consider vector code: MKL uses SSE2 or SSE3, and our generated code uses SSE3 (as said above, this is achieved by generating complex C99 code and using compiler vectorization). The source code of MKL is not available, and MKL's internal structure and the algorithms used are not documented. For this reason we first analyze MKL's runtime behavior.

MKL DFT runtime behavior. In the first experiment we timed MKL for all DFT sizes up to 1024. Figure 4 shows the result. The points, and thus the DFT sizes, are partitioned into three classes: Class 1 includes all sizes whose largest prime factor is smaller than 32 (black triangles); class 2 includes all sizes whose largest prime factor is between 32 and 150 (white squares); and class 3 includes all sizes whose largest prime factor exceeds 150 (gray bullets). Note that this is not a division into levels (see Definition 1). Class 1 is the set of level 0 numbers for unrolling threshold $B = 32$; class 2 is the set of level 0 numbers for $B = 150$, which are not in class 1; and class 3 is the set of all numbers of level 1 and higher for $B = 150$.

From Figure 4 we speculate that MKL uses Cooley-Tukey (and maybe prime-factor), in tandem with a quadratic cost algorithm for prime size kernels smaller than 150 (each of the parabolic strands in class 2 seems to have quadratic or near-quadratic behavior as we confirmed through polynomial curve fitting), and Bluestein for class 3 numbers. In other words, it seems that Rader is not used.

Since class 1 sizes are precisely the level 0 numbers for the (reasonable) unrolling threshold $B = 32$, we compare against MKL only for class 2 and class 3 sizes, and we choose $B = 32$ as unrolling threshold in SPIRAL. Since we generate vector code, a larger threshold would be possible, but for 150 the basic blocks become too large.

Class 2. Figure 5 shows the ratio of the runtimes of SPIRAL generated code and MKL for class 2 sizes. A value < 1 signifies that SPIRAL's code is faster. As we see from this plot, SPIRAL code is between 30% faster and 4.5 times slower. The median of all ratios is 1.35, i.e., 35% slower. To better understand for which DFT sizes we fare worse, we further partition into subclasses. If MKL does successfully use an $O(n^2)$ algorithm for small prime sizes, one would expect that we fare worse for sizes that require several Rader steps independent of whether they are unrolled or not due to their relatively high arithmetic cost. For this reason we partition the size in Figure 5 into those of level 1 (black triangles), level 2 (white squares), and level 3 and higher (gray bullets) for unrolling threshold 5 (the 5 was chosen empirically). As expected our code compares best for level 1 (median 1.02), a little worse for level 2 (median 1.36), and worst for level 3 and higher (median 2.78).

Class 3. Figure 6 shows the runtime of MKL's DFTs and SPIRAL generated DFT code of problem sizes in class 3. Obviously, MKL (gray bullets) uses Bluestein (8). Class 3 sizes, as class 2 sizes, are level 1 or higher with respect to our unrolling threshold 32. For level 1 sizes (black triangles), SPIRAL generated code runs up to twice as fast as MKL code, and is faster for most sizes. For level 2 sizes (white squares) MKL is faster for most numbers; however, SPIRAL's Bluestein (gray line) limits the slowdown to 30%, which is roughly the loss of our SSE3 vectorization versus the better SSE2 vectorization in [8] for the 2-power FFTs used in Bluestein.

Summary. SPIRAL generated code tends to be faster than MKL DFT code for level 1 sizes with respect to unrolling threshold 32 and is comparable (with the disadvantage of SSE3 vectorization leading to 20 % slowdown) for level 0 and level 2 sizes. For level 0 sizes with prime factors of level 3 or higher with respect to unrolling threshold 5, however, SPIRAL generated code is far suboptimal due to multiple Rader steps (7) in basic blocks.

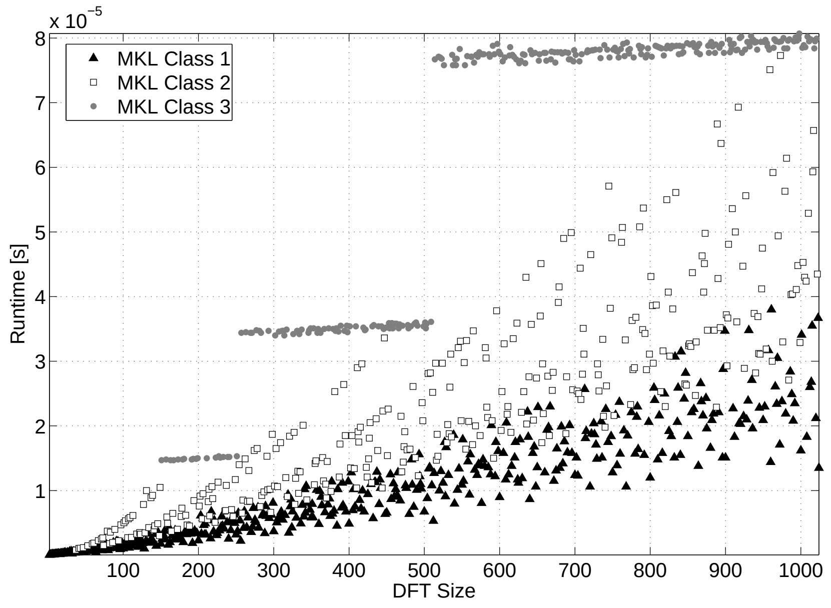

Figure 4. Runtimes of Intel MKL's DFT for sizes up to 1024. The sizes are partitioned into 3 classes (explained in the text).

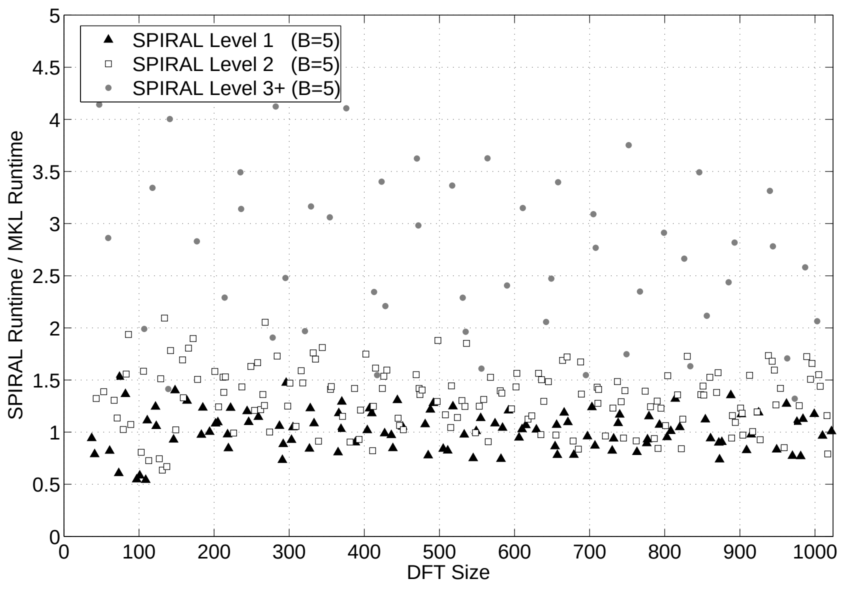

Figure 5. Runtime ratios of SPIRAL's DFT and MKL's DFT for class 2 sizes, partitioned into 3 classes (explained in the text).

6. Conclusions

We presented an extension of the SPL language and SPL compiler used in SPIRAL to enable loop optimization for signal transform algorithms at the formula level. The approach concurs with the SPIRAL philosophy which aims to perform optimizations at the "right level" of abstraction. The "right level" depends on the type of optimization and is usually a research question. For the loop optimizations considered in this paper, we found that the right level is between SPL and the actual code as reflected by Σ-SPL, which, unlike SPL, represents loops and index mappings explicitly and compactly. We want to emphasize that the main goal was not to optimize the discussed FFT algorithms, but to develop a general framework that can perform these loop optimizations for the entire domain of linear transforms considered by SPIRAL. Our approach is specific to this domain, but general within this domain.

Of course, our approach requires us to identify for each considered transform and set of transform algorithms the proper set of index function simplification rules, which, in this paper, we did for the FFTs (5)–(8). As a result we could generate DFT code for sizes of level 1 and 2 that is considerably faster than FFTW under equal choice of algorithms and unrolling threshold. For sizes of level 3 and higher, we achieved an even higher speed-up, which, however was based on the Bluestein FFT, which we believe could be easily incorporated into FFTW.

The comparison to Intel's MKL produced mixed results, between considerable improvements for some sizes and considerably

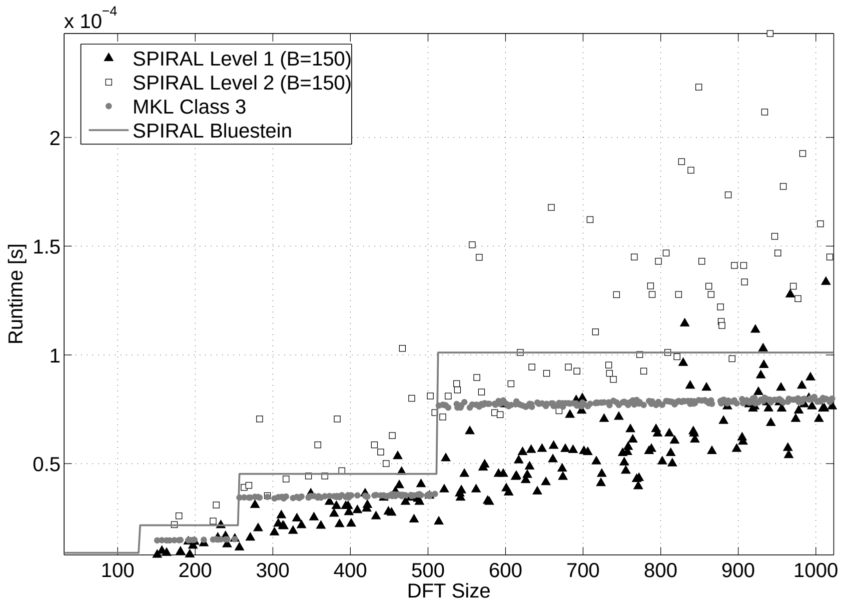

Figure 6. Runtimes of SPIRAL's DFT and MKL's DFT for class 3 sizes (where MKL uses Bluestein). The SPIRAL runtimes are partitioned into 2 classes (explained in the text). SPIRAL's Bluestein is also shown.

slower code for other sizes. Since we believe that MKL does not use Rader, but some $O(n^2)$ algorithm for small primes, we plan to incorporate the corresponding rules into SPIRAL to more carefully study the trade-offs. No matter which strategy for small primes is chosen, it is interesting to record that the Bluestein FFT is crucial for high performance if all DFT sizes are to be implemented.

Finally, it was interesting to note that by generating explicit complex C99 code and using compiler vectorization for SSE3, we obtained, independent of the DFT size, a consistent performance improvement of about 50%.

Acknowledgments

This work was supported by NSF through awards 0234293, 0310941, and 0325687. Franz Franchetti was supported by the Austrian Science Fund FWF's Erwin Schroedinger Fellowship J2322. The authors also wish to thank the anonymous reviewers for their suggestions that helped to improve the paper.

References

[1] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[2] FFTW web site. `www.fftw.org`.

[3] M. Frigo. A fast Fourier transform compiler. In *Proc. PLDI*, pages 169–180, 1999.

[4] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005. Special issue on *Program Generation, Optimization, and Adaptation*.

[5] 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. Circuits and Systems*, 9, 1990.

[6] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, Ore., 1993. Berlin: Springer Verlag.

[7] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Trans. Progr. Lang. Syst.*, 20(3):635–678, 1998.

[8] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232–275, 2005. Special issue on *Program Generation, Optimization, and Adaptation*.

[9] SPIRAL web site. `www.spiral.net`.

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

[11] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proc. PLDI*, pages 298–308, 2001.