Short Vector Code Generation for the Discrete Fourier Transform

Franz Franchetti[*]
Applied and Numerical Mathematics
Technical University of Vienna, Austria
franz.franchetti@tuwien.ac.at

Markus Püschel
Electrical and Computer Engineering
Carnegie Mellon University
pueschel@ece.cmu.edu

Abstract

In this paper we use a mathematical approach to automatically generate high performance short vector code for the discrete Fourier transform (DFT). We represent the well-known Cooley-Tukey fast Fourier transform in a mathematical notation and formally derive a "short vector variant". Using this recursion we generate for a given DFT a large number of different algorithms, represented as formulas, and translate them into short vector code. Then we present a vector code specific dynamic programming method that searches in the space of different implementations for the fastest on the given architecture. We implemented this approach as part of the SPIRAL library generator. On Pentium III and 4, our automatically generated SSE and SSE2 vector code compares favorably with the hand-tuned Intel vendor library.

1. Introduction

Short Vector Extensions. The computational simplicity of multimedia applications has spawned the invention of *short vector SIMD (single instruction, multiple data) extensions*, which are included in most recent instruction set architectures. These extensions provide datatypes and instructions to operate in parallel on short vectors (currently of length 2 or 4) of floating point numbers. Table 1 gives an overview on these extensions. Examples include SSE (4-way single precision) provided by Pentium III/4 and Athlon XP, and SSE2 (2-way double precision) provided by Pentium 4.

Short vector instructions provide a large potential speed-up (factors of 2 or 4) for performance-critical applications, but pose a difficult challenge for software developers for the following reasons:

- Automatic vectorization of C code by compilers is very limited for all but the most simple programs and yields only moderate speed-up. Most code cannot be vectorized at all.
- No common programming interface (API) exists for using these instructions. The programmer has either to resort to assembly programming or to the use of C language extensions (called *intrinsics* or *built-in functions*) provided by the vendor. These intrinsics are not standardized, neither across compilers nor across architectures. Both strategies require a high level of programming expertise and yield code that is not portable.
- The performance of short vector instructions is very sensitive to the data access pattern during computation. Unaligned and non-unit stride access can deteriorate performance.

Vendor	Name	n-way	Prec.	Processor
Intel	SSE	4-way	single	Pentium III Pentium 4
Intel	SSE2	2-way	double	Pentium 4
Intel	IPF	2-way	single	Itanium Itanium 2
AMD	3DNow!	2-way	single	K6
AMD	Enhanced 3DNow!	2-way	single	K7, Athlon XP Athlon MP
AMD	3DNow! Professional	4-way	single	Athlon XP Athlon MP
Motorola	AltiVec	4-way	single	MPC 74xx G4

Table 1. Short vector SIMD extensions.

DFT Vector Code Generation. In this paper we address the problem of creating high performance short vector code for the discrete Fourier transform (DFT), which is ubiquitously used in signal processing and across scientific disciplines and has in many applications a virtually unlimited need for performance. Our approach is based on SPIRAL [6], a library generator for digital signal processing (DSP) transforms. SPIRAL generates for a given transform many different al-

[*]This work was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF and by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.

gorithms, represented as mathematical formulas. These formulas are translated into programs, which are timed on the given platform. By intelligently searching in the space of these formulas and their implementations, SPIRAL automatically finds an algorithm and its implementation that is adapted to the given architecture. In [2] we extended SPIRAL to generate SSE vector code for the DFT and other transforms. We presented

- a short vector API of C macros that can be efficiently implemented on all current short vector architectures; and
- a set of basic formula building blocks that can be efficiently implemented using the API.

In this paper we present the additional tools that are necessary to generate very fast DFT implementations across platforms and across vector extensions.

- We formally derive a *short vector* version of the famous Cooley-Tukey fast Fourier transform (FFT) for a complex input vector in the interleaved format (real and imaginary part alternately). The new variant consists exclusively of building blocks that are efficiently vectorizable on all current short vector SIMD architectures, i.e., it can be implemented using vector memory access, vector arithmetic, and a small number of in-register permutations.
- We present two vector code specific dynamic programming search methods.

We included these methods into SPIRAL to *automatically generate* short vector DFT code for different platforms using the SSE and SSE2 instruction set. Our generated code is competitive with or faster than the hand-tuned Intel vendor library MKL 5.1. and yields a speed-up compared to the best available C code (from FFTW 2.1.3 [3] or SPIRAL) of up to a factor of 3.3 for SSE, and up to a factor of 1.8 for SSE2 code. We also show that the best algorithm found depends on the platform and on the data format (i.e., scalar, SSE2, and SSE), and that automatic compiler vectorization yields suboptimal performance. In summary, our approach provides portable short vector code *and* portable high performance.

Related Work. Because of the problems sketched above, there are only few research efforts on short vector DFT code. Reference [9] provides a DFT implementation using SSE, and is included in our benchmarks. FFTW 2.1.3 [3] provides very efficient C code, but no short vector code. The best currently available SSE code is provided by the vendor library MKL 5.1. FFTW-GEL [5] provides short vector code, but the vectorization technique is restricted to two-way short vector extensions. FFTW-GEL for 3DNow! and SSE2 is included in our benchmarks. Finally, we want to note that the "original" vector computers used a decade or longer ago had a typical vector length of at least 64, and a high startup cost for using vector instructions [4]. As a consequence, the algorithms designed for these platforms are not suitable for short vector implementations. The corresponding libraries were implemented in assembly and can thus not be used for benchmarking against our generated code.

Organization. In Section 2 we give an overview of SPIRAL and explain the mathematical framework that we use to represent and manipulate DFT algorithms. Section 3 contains the main advances: a recursion method for the DFT suitable for vectorization, and methods to search the space of algorithms that can be derived from it. We benchmark our generated DFT code in Section 4 and conclude with Section 5.

2. Background: SPIRAL and the Mathematics of Transforms

In this section we provide the background for our approach to automatic generation and platform adaptation of short vector code for the discrete Fourier transform (DFT). First, we briefly introduce SPIRAL, a code generator for DSP transforms, which provides the context and the methodology for our approach. Then we explain the formal mathematical notation that we use throughout the paper to represent and manipulate DFT algorithms.

2.1. SPIRAL

SPIRAL is a generator for high performance code for DSP transforms like the DFT, the discrete cosine transforms (DCTs), and many others [6]. SPIRAL uses a mathematical approach that translates the implementation problem into a search in the space of structurally different algorithms and their possible implementations to generate code that is adapted to the given computing platform. At the heart of SPIRAL's approach is a concise mathematical language that represents the many different algorithms for a transform as formulas. These formulas are automatically generated and automatically translated into code, thus enabling automated search for the best.

The architecture of SPIRAL is shown in Figure 1. The user specifies a transform she wants to implement, e.g., a DFT of size 1024. The **Formula Generator** module expands the transform into one (or several) formulas, i.e., algorithms, represented in the SPIRAL proprietary language SPL (signal processing language). The **Formula Translator** (also called SPL compiler) translates the formula into a C or Fortran program. The runtime of the generated program is fed back into a **Search Engine** that controls the generation of the next formula and possible implementation choices, such as the degree of loop unrolling. Iteration of this process yields a platform-adapted implementation. Search methods in SPIRAL include dynamic programming and evolutionary algorithms. By including the mathematics in the system, SPIRAL can optimize, akin to a human expert programmer, on the implementation level *and* the algorithmic level to find the best match to the given platform. Further information on SPIRAL can be found in [6, 8, 7, 12, 10].

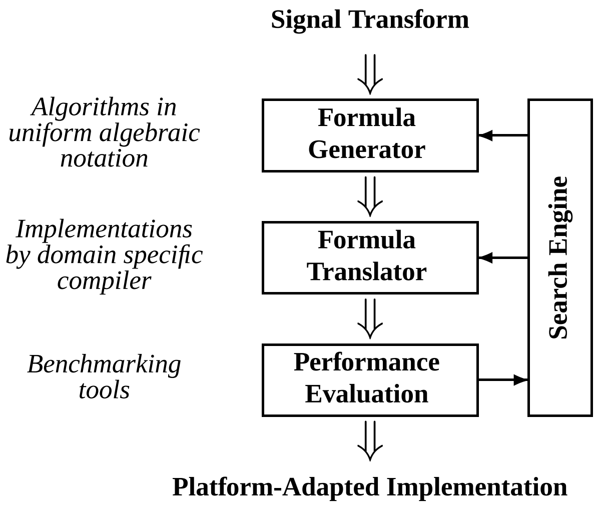

Signal Transform
Algorithms in uniform algebraic notation
Formula Generator
Implementations by domain specific compiler
Formula Translator
Benchmarking tools
Performance Evaluation
Search Engine
Platform-Adapted Implementation

Figure 1. The architecture of SPIRAL.

In [2] we made first steps in extending SPIRAL to generator short vector code. In this paper we show how to use this methodology to obtain very fast DFT code.

2.2. Mathematical Framework

In this section we describe SPIRAL's mathematical framework, which is the foundation of our approach. The key point is the representation of algorithms as *mathematical formulas*. SPIRAL generates these formulas (as SPL programs) and translates them into code (see Section 2.1). To obtain fast short vector code, this approach is not sufficient. We will show in Section 3.2 how to *formally manipulate* DFT formulas using mathematical identities to obtain the structure necessary for the generation of high performance vector code.

DSP Transforms and Algorithms. A (linear) DSP transform is a multiplication of the sampled signal $x \in \mathbb{C}^n$ by a transform matrix M of size $n \times n$, $x \mapsto M \cdot x$. In this paper we are mainly concerned with the DFT, which, for size n, is given by the matrix

$$\mathrm{DFT}_n = [\omega_n^{k\ell} \mid \leq k, \ell < n], \quad \omega_n = e^{2\pi\sqrt{-1}/n}.$$

Fast algorithms for DSP transform can be represented as structured sparse factorizations of the transform matrix. The famous Cooley-Tukey fast Fourier transform (FFT) is a recursion method that computes a DFT_{mn} from smaller DFT_m's and DFT_n's. It can be written as

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}_n^{mn} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{mn} \quad (1)$$

(see [11]), where we denote with I_n the $n \times n$ identity matrix, by T_n^{mn} the complex *twiddle* diagonal matrix, and by $\mathrm{L}_n^{mn} : im + j \mapsto jn + i,\ 0 \leq j < m,\ 0 \leq i < n$ the *stride permutation* matrix; L_n^{mn} reads an input at stride n and stores it at stride 1. Particularly important is the *tensor* or *Kronecker product* of matrices, and the *direct sum*, respectively defined as

$$A \otimes B = \begin{bmatrix} a_{1,1} \cdot B & \dots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{n,1} \cdot B & \dots & a_{n,n} \cdot B \end{bmatrix} \quad (2)$$

with $A = [a_{k,\ell}]_{0 \leq k,\ell \leq n}$, and

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

The latter will occur later. These definitions show that all factors in (1) are sparse, and that the factorization (1) reduces the arithmetic cost of computing a DFT. We call Equation (1) a *breakdown rule* or simply *rule*. Recursive application of rules until all occurring transforms are base cases yields a *formula* which defines a fast algorithm. For example, a formula for a DFT_8 can be derived by applying (1) twice (corresponding to $8 = 4 \cdot 2$, followed by $4 = 2 \cdot 2$):

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

In this paper we consider only DFTs of 2-power size $n = 2^k$ and only rule (1) for formula generation. However, the presented framework covers all linear DSP transforms [8]. The degree of freedom in choosing a factorization of the transform size in each step leads to a large number of formulas with about equal arithmetic cost, but different structure, which leads to different runtimes when implemented. SPIRAL uses search in the space of formulas for optimization and platform adaptation.

Formulas and Programs. Formulas are a natural representation of algorithms from a mathematical point of view, but also have an intuitive interpretation as programs, and can thus be translated into code (by SPIRAL's SPL compiler [12]). For example, multiplying a tensor product $\mathrm{I}_n \otimes A$ or $A \otimes \mathrm{I}_n$ to an input vector x leads to loop code (if desired), or, a permutation determines in which order the elements of x are accessed. This access pattern is one of the main problems when generating vector code. We use formula manipulation to solve this problem.

Formula Manipulation. A given formula can be manipulated using mathematical identities to derive correct variations with different structure, i.e., different data flow. Important examples for formula manipulations include (the subscript of A, B denotes the matrix size)

$$(\mathrm{L}_m^{mn})^{-1} = \mathrm{L}_n^{mn} \quad (3)$$

and

$$\begin{aligned} \mathrm{L}_n^{mn}(A_m \otimes B_n)\,\mathrm{L}_m^{mn} &= (A_m \otimes B_n)^{\mathrm{L}_m^{mn}} \\ &= (B_n \otimes A_m), \end{aligned} \tag{4}$$

where $M^P = P^{-1}MP$ denotes *conjugation*. Further identities can be found in Table 2. We will use these identities in Section 3 to derive a short vector variant of the Cooley-Tukey FFT (1).

$$\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n \tag{5}$$

$$(\mathrm{L}_m^{mn})^{-1} = \mathrm{L}_n^{mn} \tag{6}$$

$$\mathrm{L}_m^{kmn}\,\mathrm{L}_n^{kmn} = \mathrm{L}_n^{kmn}\,\mathrm{L}_m^{kmn} = \mathrm{L}_{mn}^{kmn} \tag{7}$$

$$\mathrm{L}_n^{kmn} = (\mathrm{L}_n^{kn} \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}_n^{mn}) \tag{8}$$

$$\mathrm{L}_{km}^{kmn} = (\mathrm{I}_k \otimes \mathrm{L}_m^{mn})(\mathrm{L}_k^{kn} \otimes \mathrm{I}_m) \tag{9}$$

$$\mathrm{L}_n^{mn}(A_m \otimes B_n)\,\mathrm{L}_m^{mn} = (A_m \otimes B_n)^{\mathrm{L}_m^{mn}} = (B_n \otimes A_m) \tag{10}$$

$$\begin{aligned} (\mathrm{I}_{sk} \otimes A_{ms\times n})\,\mathrm{L}_{sk}^{skn} = \Big(\mathrm{I}_k \otimes (\mathrm{L}_s^{ms} \otimes \mathrm{I}_s) \\ \Big(\mathrm{I}_m \otimes \mathrm{L}_s^{s^2}\Big)\,(A_{ms\times n} \otimes \mathrm{I}_s)\Big)\,\Big(\mathrm{L}_k^{kn} \otimes \mathrm{I}_s\Big) \end{aligned} \tag{11}$$

Table 2. Formula manipulations.

Complex Arithmetic. Above we represented DFT algorithms as formulas built from matrices with *complex* entries. However, vector instructions provide only *real* arithmetic of vectors with their elements stored contiguously in memory. Thus, to map formulas to vector code, we have to translate complex formulas into real ones. As data format we choose the commonly used *interleaved complex format* (alternately real and imaginary part) and express it formally as a mapping of formulas. We use the fact that the complex multiplication $(u+iv)\cdot(y+iz)$ is equivalent to the real multiplication $\left[\begin{smallmatrix} u & -v \\ v & u \end{smallmatrix}\right]\cdot\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right]$. Thus, the complex matrix-vector multiplication $M \cdot x \in \mathbb{C}^n$ corresponds to $\overline{M}\cdot\tilde{x} \in \mathbb{R}^{2n}$, where $\overline{M}$ arises from M by replacing every entry $u+iv$ by the corresponding (2×2)-matrix above, and $\tilde{x}$ is in interleaved complex format. To evaluate the $\overline{(\cdot)}$ of a formula we use the set of identities given in Table 3. For example, $\overline{A\cdot B} = \overline{A}\cdot\overline{B}$.

3. Generating Fast Short Vector DFT Code

To generate very fast short vector code for the DFT we need to provide a set of tools that extend all 3 modules of SPIRAL (see Figure 1). We order them conceptually from low to high level:

- A *short vector API* implemented as C macros that can be implemented on any short vector architecture. The API was developed as part of [2].
- A set of *formula building blocks* that can be mapped to efficient vector code on all current vector extensions using this API and that is sufficient to implement the DFT.

$$\overline{A\cdot B} = \overline{A}\cdot\overline{B} \tag{12}$$

$$\overline{A} = A \otimes \mathrm{I}_2, \quad A \text{ real} \tag{13}$$

$$\overline{A_m \otimes \mathrm{I}_n} = (\overline{A_m} \otimes \mathrm{I}_n)^{(\mathrm{I}_m \otimes \mathrm{L}_2^{2n})} \tag{14}$$

$$\overline{A_m \otimes \mathrm{I}_\nu} = (\overline{A_m} \otimes \mathrm{I}_\nu)^{(\mathrm{I}_m \otimes \mathrm{L}_2^{2\nu})} \tag{15}$$

$$\overline{A_m \otimes \mathrm{I}_n} = (\overline{A_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu)^{\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}_2^{2\nu}\right)} \tag{16}$$

$$\overline{D}' = \overline{D}^{\left(\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}_\nu^{2\nu}\right)},\ D = \operatorname{diag}(c_0,\ldots,c_{n-1}) \tag{17}$$

Table 3. Identities for the bar operator $\overline{(\cdot)}$.

- A *short vector FFT rule*, derived from (1), that is exclusively built from these building blocks, and thus can be used to generate a large class of vectorizable formulas (see Section 2.2) that constitute the algorithm search space.
- A *vector code specific search method* to find a fast implementation.

We explain the latter three bullets in the following.

3.1. Formula Building Blocks

Our portable SIMD API is designed such that the only architectural parameter on the formula level is the the vector length ν of the short vector SIMD extension. In this section we present a set of basic formula constructs, parameterized by ν, that can be efficiently implemented using the API, and thus on every short vector architecture. In other words, the structure of the building blocks allows their implementation using exclusively *aligned* vector memory access, vector arithmetic, vector permutations, and certain register permutations.

Tensor Product. The simplest construct that can be naturally mapped to vector code is any tensor product of the form

$$A \otimes \mathrm{I}_\nu, \quad A \in \mathbb{R}^{m\times n}. \tag{18}$$

The corresponding code is obtained by replacing every scalar operation in a program for the formula A by a ν-way vector operation. A is subsequently called *vector terminal*, since the construct solves the vectorization problem independent of A. In particular, $\overline{\mathrm{DFT}}_n \otimes \mathrm{I}_\nu$ can be completely vectorized, no matter how DFT_n is further expanded.

Permutations. All permutations of the form

$$P \otimes Q = (P \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes Q), \tag{19}$$

where $Q \in \left\{\mathrm{I}_\nu,\, \mathrm{L}_2^{2\nu},\, \mathrm{L}_\nu^{2\nu},\, \mathrm{L}_\nu^{\nu^2}\right\}$ and P is arbitrary, are included. The permutation $P \otimes \mathrm{I}_m$ operates on blocks of length $m \in \{\nu, 2\nu, \nu^2\}$, i.e., $\nu \mid m$, and can thus be realized using vector variables. $\mathrm{I}_n \otimes Q$ is implemented by applying Q to n blocks of m/ν vector variables. The register-to-register permutations required by the different Q's can be implemented on all SIMD architectures, on some by specialized instructions. The actual implementation of Q is hidden by the portable SIMD API.

Complex Diagonals. The last basic construct covers complex diagonal matrices $D = \operatorname{diag}(c_0, \dots, c_{\nu-1})$, $c_k = a_k + ib_k$, which cannot directly be mapped efficiently onto short vector instructions (note that $\overline{D}$ is a direct sum of 2×2 matrices, which has no obvious vector structure). We conjugate complex diagonals ($A^P = P^{-1}AP$) to obtain

$$\overline{D}' = \overline{D}^{\mathrm{L}^{2\nu}_{\nu}} = \begin{bmatrix} \operatorname{diag}(a_0, \dots, a_{\nu-1}) & -\operatorname{diag}(b_0, \dots, b_{\nu-1}) \\ \operatorname{diag}(b_0, \dots, b_{\nu-1}) & \operatorname{diag}(a_0, \dots, a_{\nu-1}) \end{bmatrix} \tag{20}$$

The construct in (20) has the same structure as (18) with $A \in \mathbb{R}^{2\times 2}$, only the the nonzero entries vary, and can thus be implemented using ν-way vector arithmetic. To implement the complex twiddle diagonal $\overline{\mathrm{T}}^{mn}_{n}$ in (1), we divide it into subdiagonals $\overline{D_i}$ of length ν and apply (20). Formally,

$$\overline{\mathrm{T}}'^{mn}_{n} = \overline{\mathrm{T}}^{mn\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_{\nu}\right)}_{n} = \bigoplus_{i=0}^{\frac{mn}{\nu}-1} \overline{D}'_i, \quad \nu \mid mn \tag{21}$$

In the remainder of this paper, we will refer to the constructs defined by (18), (19), (20), and (21) as *vector constructs*.

3.2. Short Vector FFT Rule

In this section we derive a short vector variant of the Cooley-Tukey FFT rule given in (1) (in real arithmetic, i.e., with $\overline{(\cdot)}$ applied). The derived rule is exclusively built from the constructs presented in Section 3.1 and is thus parameterized by the vector length ν, which ensures complete vectorization on all current short vector SIMD architectures using our vector API. Further, the rule expands a given DFT in one step into vector terminals that can be further expanded using the ordinary scalar rule (1), which gives rise to a large space of vectorizable formulas. We will show in Section 4 that by searching this space we can generate very fast code.

3.2.1 Derivation

The starting point for the derivation is (1) mapped to real arithmetic, i.e.,

$$\overline{\mathrm{DFT}}_{mn} = \overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}^{mn}_{n} (\mathrm{I}_m \otimes \mathrm{DFT}_n) \mathrm{L}^{mn}_{m}} \tag{22}$$

and we assume that

$$\nu \mid m \quad \text{and} \quad \nu \mid n.$$

The standard way of translating (1) into real code using the complex interleaved format corresponds to a straightforward application of the identities in Table 3, starting with distributing $\overline{(\cdot)}$ over the factors in (22) to get

$$\begin{aligned} \overline{\mathrm{DFT}}_{mn} =& (\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_n)^{\left(\mathrm{I}_m \otimes \mathrm{L}^{2n}_{2}\right)} \\ & \overline{\mathrm{T}}^{mn}_{n} (\mathrm{I}_m \otimes \overline{\mathrm{DFT}}_n)(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_2) \end{aligned} \tag{23}$$

This formula is not built exclusively from vector constructs (see Section 3.1). Formally, when mapping it to vector instructions the following difficulties occur:

- $\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_2$ does not match (19) for $\nu \neq 2$.
- $\mathrm{I}_m \otimes \mathrm{L}^{2n}_{2}$ and $\mathrm{I}_m \otimes \mathrm{L}^{2n}_{n}$ do not match (19) ($n \neq \nu$).
- $\overline{\mathrm{DFT}}_m \otimes \mathrm{I}_n$ and $\mathrm{I}_m \otimes \overline{\mathrm{DFT}}_n$ do not match (18) (they are no vector terminals).
- $\overline{\mathrm{T}}^{mn}_{n}$ does not match (21).

Thus, this formula cannot be directly mapped to short vector SIMD hardware without further manipulation. Using the identities in Table 2 and (17), it is possible to modify (23) to obtain a better structure. However, the problem of $\mathrm{L}^{mn}_{n} \otimes \mathrm{I}_2$ remains and forces sub-vector access. We have tried different variants of (23) and obtained moderate to good runtime results in [1, 2].

The key to complete and efficient vectorization, a better overall structure, and higher performance, is to pursue a different derivation that starts by applying the bar operator in (22) differently to the derivation of (23). We present the derivation in detail.

We start by distributing the bar operator in (22) over only three factors, using identity (12), and obtain

$$\overline{\mathrm{DFT}}_{mn} = \underbrace{\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)}}_{(a)}\, \underbrace{\overline{\mathrm{T}}^{mn}_{n}}_{(b)}\, \underbrace{\overline{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_{m}}}_{(c)} \tag{24}$$

To further manipulate (a), we can use any of the identities (14)–(16) in Table 3. It turns out that (16) is best, since it leads later to a cancellation of permutations and thus a simpler structure. We obtain

$$\overline{(\mathrm{DFT}_m \otimes \mathrm{I}_n)} = \left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu\right)^{\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_{2}\right)}.$$

Note that the construct $\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}}$ is a vector terminal, i.e., it matches (18), and can thus be further expanded by the *scalar* rule (1).

Construct (b) in (24) is a complex diagonal matrix and is transformed by applying the identities (6) and (17) to get

$$\overline{\mathrm{T}}^{mn}_{n} = \overline{\mathrm{T}}'^{mn\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}^{2\nu}_{2}\right)}_{n}. \tag{25}$$

Construct (c) requires the most complicated transformation among the three factors in (24). We first manipulate the complex formula, and then apply the bar operator. By factoring the stride permutation (identities (5), (8)), partially flipping the tensor product (identity (10)), and

$$\overline{\mathrm{DFT}}_{mn} = \left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}_{\nu}^{2\nu}\right)\left(\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu\right)\overline{\mathrm{T}}'^{mn}_{n}\left(\mathrm{I}_{\frac{m}{\nu}} \otimes \left(\mathrm{L}_{\nu}^{2n} \otimes \mathrm{I}_\nu\right)\left(\mathrm{I}_{\frac{2n}{\nu}} \otimes \mathrm{L}_{\nu}^{\nu^2}\right)\left(\overline{\mathrm{DFT}}_n \otimes \mathrm{I}_\nu\right)\right)\left(\mathrm{L}^{\frac{mn}{\nu}}_{\frac{m}{\nu}} \otimes \mathrm{L}_2^{2\nu}\right) \qquad (26)$$

Table 4. Short vector variant of the Cooley-Tukey FFT, $\nu \mid m, n$.

using several other identities in Table 3, we obtain a special case of (11),

$$(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn} = \Big(\mathrm{I}_{\frac{m}{\nu}} \otimes (\mathrm{L}_\nu^n \otimes \mathrm{I}_\nu)$$
$$\left(\mathrm{I}_{\frac{n}{\nu}} \otimes \mathrm{L}_\nu^{\nu^2}\right)(\mathrm{DFT}_n \otimes \mathrm{I}_\nu)\Big)\left(\mathrm{L}^{\frac{mn}{\nu}}_{\frac{m}{\nu}} \otimes \mathrm{I}_\nu\right).$$

The permutation on the right side of $\mathrm{DFT}_n \otimes \mathrm{I}_\nu$ permutes blocks while the permutations on the left side is a product of a permutation on blocks and a permutation within blocks. In a second step, the bar operator is applied to obtain a real formula. Using now identities from Table 3 we obtain the desired structure

$$\overline{(\mathrm{I}_m \otimes \mathrm{DFT}_n)\,\mathrm{L}_m^{mn}} = \left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathrm{L}_\nu^{2\nu}\right)\Big(\mathrm{I}_{\frac{m}{\nu}} \otimes (\mathrm{L}_\nu^{2n} \otimes \mathrm{I}_\nu)$$
$$\left(\mathrm{I}_{\frac{2n}{\nu}} \otimes \mathrm{L}_\nu^{\nu^2}\right)(\overline{\mathrm{DFT}}_n \otimes \mathrm{I}_\nu)\Big)\left(\mathrm{L}^{\frac{mn}{\nu}}_{\frac{m}{\nu}} \otimes \mathrm{L}_2^{2\nu}\right)$$

This equation now consists exclusively of vector constructs. Again, $\overline{\mathrm{DFT}}_n$ is a vector terminal, which can be further expanded by the scalar rule (1).

Now all factors in (24) are built exclusively from vector constructs. By multiplying the derived factors, some permutations introduced as conjugations cancel out using identity (6). This cancellation simplifies data flow, and thus improves performance, and is another reason for the particular choice of transformations above. The final short vector rule (26) for the DFT is displayed in Table 4.

In summary we obtain the main result of this paper: an FFT variant that consists exclusively of vectorizable constructs and decomposes a given $\overline{\mathrm{DFT}}_{mn}$, for $\nu \mid m, n$, into vector terminals $\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}}$ and $\overline{\mathrm{DFT}}_n$, which both can be further expanded using scalar rules, e.g., by (1). Thus, all formulas derived this way can be efficiently vectorized. We note that a formal transposition of (26) yields a different fully vectorizable rule.

Next, we explain how the formulas generated from (26) are mapped into vector code.

3.2.2 Implementation

For the code generation we use the SIMD version of SPIRAL's SPL compiler developed in [2]. All formulas generated using the vector FFT rule (26) match the general construct this compiler can translate into vector code and the generated code uses *exclusively* vector memory access. We briefly describe how the SIMD SPL compiler translates the occurring constructs. For further information we refer the reader to [2]. The generated code is C code using the macros of our SIMD API.

Vector Terminals. Vector code for vector terminals $A \otimes \mathrm{I}_\nu$ is generated by generating scalar code for A and replacing every scalar operation by the corresponding vector operation.

Permutations. Any permutation in a final formula that is not part of a vector terminal matches equation (19). The short vector SPL compiler fuses these permutations with memory access operations required by the subsequent computation. Finally, they are implemented via variable renaming and calls to combined memory access and reordering macros that are part of the portable SIMD API. No additional (scalar) memory access is caused by this method.

Twiddle Matrix. The construct $\overline{\mathrm{T}}'^{mn}_{n}$ is handled as a pre- or post-processing scaling operation of memory access operations.

3.3. Vector Code Specific Search Methods

SPIRAL provides different search methods to find the best algorithm for a given computing platform, including dynamic programming (DP), STEER (an evolutionary algorithm) [10], a hill climbing search, and exhaustive search. For scalar DFT implementations, it turns out that in general DP is a good choice since it terminates fast (using only (1), DP times at the order of $O(n^2)$ formulas, where n is the transform size) and finds close to the best implementations found among all search methods [8].

But in the case of short vector SIMD implementations, it turns out that DP fails to find the best algorithms, as the optimal subproblem becomes highly context dependent. We explain this in the following and present two variations of DP that we included in SPIRAL search engine to overcome this problem. In Section 4 we experimentally evaluate the different DP variants.

Standard DP. DP searches for the best implementation for a transform recursively by applying all possible rules but expanding the obtained child transforms using the previously found optimal formulas, and picking the fastest among the obtained formulas. Since the child transforms are smaller than the original construct, this process terminates. In the case of a DFT_{2^n}, using only rule (1), DP recursively generates the best implementations for all DFT_{2^k}, $k \le n$.

The method works well for scalar code, but for vector code the method is flawed. First rule (26) is applied on the top level, leading to different child transform configurations. Calling DP recursively would inevitably apply this rule again, even though the children are vector terminals, i.e., should be expanded using rule (1). As a result, the wrong breakdown strategies are found.

Vector DP. The first obvious change is to disable the vector rule (26) for vector terminals. This already leads to reasonable structured formulas. But there is a second problem: DP optimizes all vector terminals like $\overline{\mathrm{DFT}_k}$ as scalar constructs thus not taking into account the context $\overline{\mathrm{DFT}_k} \otimes \mathrm{I}_\nu$ of $\overline{\mathrm{DFT}_k}$. Thus, we make a second modification by expanding $\overline{\mathrm{DFT}_k}$ by scalar rules but always measuring the runtime of the vector code generated from $\overline{\mathrm{DFT}_k} \otimes \mathrm{I}_\nu$. For the other construct containing a vector terminal in (26), $\overline{\mathrm{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_\nu$, also $\overline{\mathrm{DFT}_m} \otimes \mathrm{I}_\nu$ is measured, independent of $\frac{n}{\nu}$.

Stride Sensitive Vector DP. This variant is directly matched to rule (26). For a given DFT_n, this search variant first creates all possible pairs (n_1, n_2) with $n = n_1 n_2$. For any pair (n_1, n_2), it searches for the best implementation of the vector terminals required by equation (26) using Vector DP. But when searching for the best DFT_{n_2} by a call to Vector DP a variant is used that finally measures $\overline{\mathrm{DFT}_{n_2} \otimes \mathrm{I}_{\frac{n_1}{\nu}}} \otimes \mathrm{I}_\nu$ instead of $\overline{\mathrm{DFT}_{n_2}} \otimes \mathrm{I}_\nu$, which makes the search sensitive to the stride $\frac{n_1}{\nu}$. The best DFT_{n_1} is found by a call to standard Vector DP. This exactly optimizes the required vector terminals, including the stride. This DP variant requires much more runtime measurements compared to the other two DP variants and thus saving the results for earlier measured pairs (n_1, n_2) speeds up the search crucially. Running the search without this memorization leads to a context and stride sensitive version ("nohash" variant), but the additional search time does not pay off.

Implementation Degrees of Freedom. In addition to the formula space, we consider two implementation degrees of freedom arising for a DFT formula generated from (26). Both degrees of freedom are machine specific, and not formula or transform size specific. Thus they can be checked and fixed at install time.

Replicated Constants. Any constant in the code generated for a vector terminal becomes a vector constant of ν times the same number in the final code. Depending on the machine, either loading the constant with vector memory access (thus storing ν numbers) or loading the scalar and using a vector fill (splat) operation may be better.

Constant Fusion. The other degree of freedom is connected to the twiddle matrix in equation (26): the multiplications can be fused with either of the vector terminals, which changes the locality of memory accesses for loading the constants and changes whether expensive arithmetics (complex scaling) is done immediately after loading or prior to storing data elements.

4. Experimental Results

In this section we present experimental results for our automatically generated short vector code for the DFT of size $n = 2^k$. We benchmark against the best available DFT implementations across different architectures, compare our different search methods, and show the structure of the best algorithms found.

To validate our approach we chose the SSE and SSE2 extensions on binary compatible, yet architectural very different platforms: (i) Intel Pentium III with SDRAM running at 1 GHz; (ii) Intel Pentium 4 with RDRAM running at 2.53 GHz; and (iii) AMD Athlon XP 2100+ with DDR-SDRAM running at 1733 MHz. These machines feature different chip sets, system busses, and memory technology, and the processors are based on different cores and have different cache architectures.

The theoretical speed-up achievable by vectorization (ignoring effects like smaller program size when using vector instructions) is a factor of four for SSE on Pentium III and Pentium 4. For SSE on Athlon XP and SSE 2 on Pentium 4 the limit is a factor of two.

By finding differently structured algorithms on different machines, we achieve high performance across architectures and across short vector extensions, which demonstrate the success of our approach in providing portable performance independent of the vector length ν and other architectural details.

We benchmarked our generated vector code against state-of-the-art C code by FFTW 2.1.3 [3] or generated by SPIRAL, against compiler vectorized C code generated by SPIRAL, and against short vector code provided by the Intel Math Kernel Library MKL 5.1[1]. Further, we included FFTW-GEL, which supports SSE2 and 3DNow! [5], and the DFT runtime results from [9]. In all cases we use the Intel C++ Compiler 6.0.

The MKL features separate versions optimized for Pentium III and 4. We note that the MKL uses in-place computation and memory prefetching instructions, which gives it an advantage over our code, which computes the DFT out-of-place and without prefetching.

FFTW-GEL is an extension of FFTW that supports 2-way vector code, made possible by a short vector "codelet" generator for small DFT sizes. The vectorization technique is restricted to 2-way vectors and generates straight-line assembly code.

All results are given in "pseudo flop/s" computed as $5n\log_2(n)/(\text{runtime in s})$ for a DFT of size n. $5n\log_2(n)$ is an upper bound for the arithmetic cost of FFT algorithms, thus the numbers are slightly higher

[1]http://developer.intel.com/software/products/mkl

than real flop/s, but the relation is preserved. SPIRAL generated scalar code was found using a DP search, SPIRAL generated vector code (using our extensions) using the best result of the two different vector DPs in Section 3.3. In both cases we included the global limit for unrolling (the size of subblocks to be unrolled) into the search.

We now present and discuss the results in detail.

Pentium 4. On this machine (2.53 GHz) we achieved the best performance per cycle and the highest speed-ups compared to scalar SPIRAL generated code (or FFTW). Using SSE (see Figure 2(a)), we achieved up to 6.25 pseudo Gflop/s and a speed-up of up to 3.1. Using SSE2 (see Figure 2(b)), we achieved up to 3.3 pseudo Gflop/s and a speed-up of up to 1.83. The performance of our code is best within L1 cache and only slightly decreases outside L1. For SSE2 we also included an exhaustive search for small DFT sizes, which yielded significant improvement only for $n = 64$. Analysis of our generated programs shows that the best found code features very small loop bodies (as opposed to medium and large unrolled blocks that typically lead to high performance) and very regular code structure. This is due to the Pentium 4's new features, namely (i) its new core with a very long pipeline, (ii) its new instruction cache that caches instructions *after* they are decoded (trace cache), and (iii) its small, but very fast data caches. Our generated SSE code outperforms the MKL for sizes $n < 2^9$ and is about equal for $n \geq 2^9$. Our generated SSE2 code compares favorably to the MKL across all considered sizes. For SSE2, FFTW-GEL achieves about the same performance as our code. However, it cannot be used for SSE as FFTW-GEL's vectorization is restricted to two-way short vector extensions. We include the results reported in [9] obtained on a similar machine which was running at 1.4 GHz. As we could not get the source code, we scaled the reported results up to the frequency of our test machine (of course, these performance numbers are only a very rough but instructive estimate). See Figure 2(a) and (b) for details.

Pentium III. We achieved up to 1.7 pseudo Gflop/s and a speed-up of up to 3.1 on a 1 GHz machine with a Coppermine core (see Figure 2(c)). The best implementations featured moderate sized loop bodies. On this machine, our code delivers the highest speed-ups for larger sizes. The Intel MKL offers lower performance compared to our codes (when comparing to the Pentium 4), which reflects Intel's additional tuning effort for the Pentium 4. The new short vector FFT rule presented in this paper removes the performance degradation on the Pentium III for larger sizes in [2] and sped up our implementation significantly. It now runs at a high performance level across all tested problem sizes. See Figure 2(c) for details.

Athlon XP. We achieved up to 2.8 pseudo Gflop/s and speed-ups of up to 1.7 on an 1733 MHz machine (see Figure 2(d)). The best implementations featured large loop bodies. On this machine, the performance of our code decreases at the L1 boundary, where the Intel MKL can keep the performance level. Analysis shows, however, that the performance level achieved by our codes for 2^{n-1} is the same as the Intel MKL achieves for 2^n. This is in part due to the in-place computation by the Intel MKL which results in smaller memory requirements. Although the 3DNow! professional extension (binary compatible to 3DNow! and SSE) features 4-way SIMD, the maximum obtainable speed-up is a factor of two, as the Athlon XP's two floating-point units then both operate as two-way SIMD units.

FFTW-GEL achieves up to 35 % higher performance as our approach; it gains the advantage from the following facts: (i) FFTW-GEL generates assembly code. (ii) FFTW-GEL directly utilizes the two-way native 3DNow! (iii) FFTW-GEL features an AMD specific assembler backend. On the other hand, our generated code faces the following disadvantages: (i) We utilize 3DNow! professional's compatibility to SSE. AMD supports SSE instructions for compatibility reasons, however, is not specific about the *performance* of its SSE implementation. (ii) AMD does not supply its own compiler (extension). We have to resort to the Intel compiler on the AMD machine which produces fair but not optimal code, however, is still among the best compilers with SSE support available for the Athlon XP. Barring these differences, we speculate that, for 2-way vectorization, both approaches are equally successful (as in Figure 2(b)). Comparing the performance of our code with the Intel MKL on the Athlon XP shows that we achieve high performance within the boundaries of AMD's SSE implementation. However, using AMD 3DNow!, higher performance can be obtained as shown by FFTW-GEL.

Compiler Vectorization. Automatic *compiler* vectorization in tandem with SPIRAL code generation provides a fair evaluation of the limits of this technique. By running a DP search, SPIRAL can find algorithms that are best structured for compiler vectorization. Further, the code generated by SPIRAL is of simple structure (e.g., contains no pointers or variable loop limits). Even though the compiler can improve on SPIRAL's scalar code, the performance is far from optimal (see "SPIRAL vect" in Figure 2(a)–(d)). As an aside, due to its structure, FFTW cannot be compiler vectorized.

Evaluation of the Search Methods. One of the main observations was that DP works well on some machines while on others it misses the best implementation considerably, thus requiring the modified versions introduced in Section 3.3. Specifically, on the Pentium III and Athlon XP, Standard DP finds implementations very

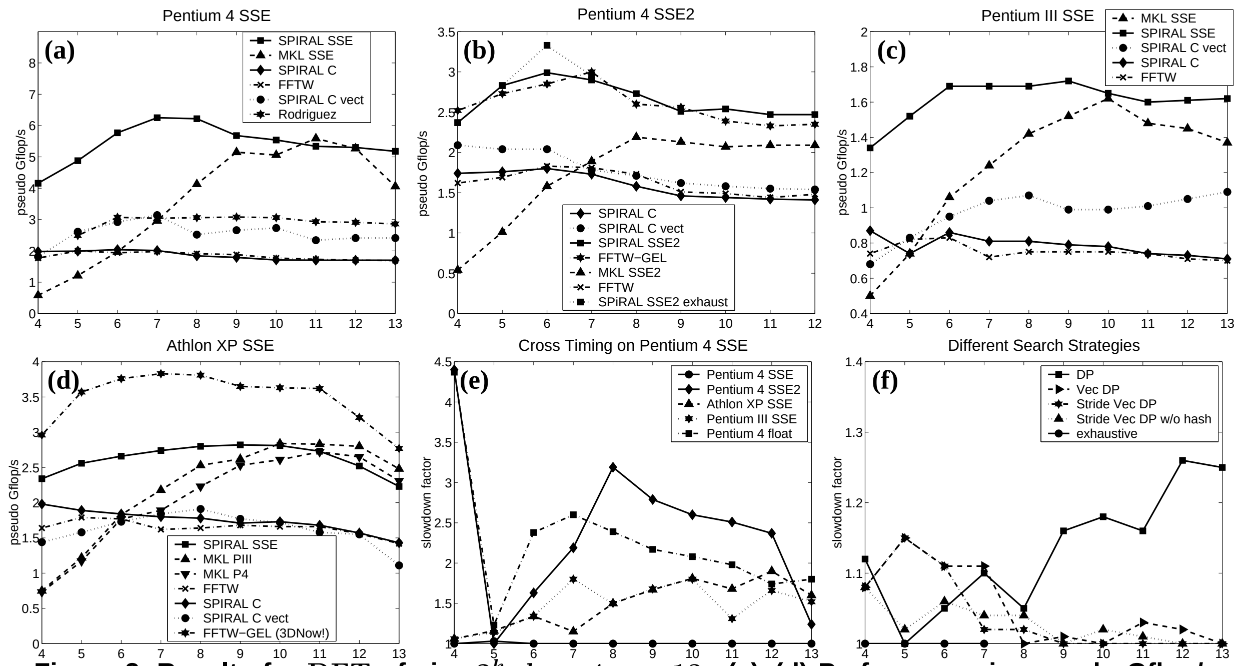


Figure 2. Results for DFT of size 2^k, $k = 4, \ldots, 13$. (a)–(d) Performance in pseudo Gflop/s $= 5\,k\,2^k/(10^9 \times \text{runtime in s})$. Higher is better. SSE is single and SSE2 is double precision. (e) Cross timing of the best algorithms found for different architectures, all measured on the Pentium 4, implemented using SSE. (f) Comparison of the best algorithms found by different search methods. For (e) and (f) the slowdown factor compared to the best is shown.

close to the best found. But on the Pentium 4 the vector-aware DPs are required to get the best performance. Figure 2(f) shows this behavior, by displaying the slowdown factor of the best code found by different search methods compared to the best: Standard DP misses the best result by up to 25 %. Further, for $n \leq 2^7$, exhaustive search leads to the best result. For $n > 2^7$ Vector DP finds code of similar performance as Stride Sensitive Vector DP. Thus, the additional search time required by the Stride Sensitive Vector DP variants does not pay off. A combination of exhaustive search (where possible) and Vector DP (for larger sizes) is the most economical search method to obtain fast short vector code.

Crosstiming. A natural question that arises is how the best algorithms found for one architecture perform on the other platforms. As an example, we show in Figure 2(e) the slow-down factor of the best found DFT formulas for Pentium III/SSE, Athlon XP/SSE, Pentium 4/scalar, Pentium 4/SSE2, and Pentium 4/SSE, when implemented using SSE on Pentium 4 (using the best compiler optimization). As expected, the Pentium 4 SSE version performs best and is the baseline. Both the scalar and the SSE2 formula perform very bad. But interestingly, also the formulas for SSE on Pentium III and on Athlon XP are up to 60 % slower than the Pentium 4 SSE version. We obtained a similar behavior on the other machines. This experiment clearly shows the need for platform-adaptation to obtain optimal performance.

Best Found Algorithms. Our approach delivers high performance on all tested systems. The structure of the best implementations, however, depends heavily on the target machine. As an example, Table 5(f) shows for a DFT of size 2^{10}, the structure of the best found formulas, displayed as trees representing the breakdown strategy (the expansion of smaller nodes is sometimes omitted).

Generally speaking, two completely different types of algorithms were found: 1. Algorithms with rather balanced trees; and 2. Algorithms with unbalanced trees tending to be right-expanded. The first type occurs when the working set fits into the L1 cache and for codes generated using compiler vectorization. The second type occurs for out-of-cache sizes; the actual structure depends on whether scalar or vector code is generated. For all ruletrees, parameters and structural details on deeper levels depend on the actual target machine, and are briefly discussed next.

Scalar Code. Right-expanded trees with machine-dependent sizes of the left leaves are found to be the most efficient formulas when using only the scalar FPU for out-of-cache sizes. These trees provide the best data locality. For in-cache sizes, balanced trees are found. For example, In Table 5, due to the larger caches (compared to Pentium 4) of the Pentium III and Athlon XP, still balanced trees are found.

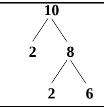
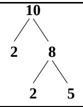
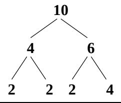
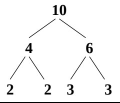
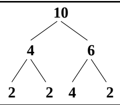
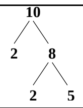
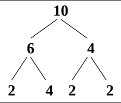
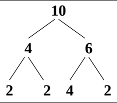
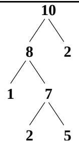
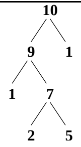
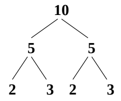
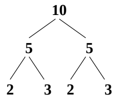
	Pentium 4 (single)	Pentium 4 (double)	Pentium III (single)	Athlon XP (single)
generated scalar code				
generated scalar code compiler vectorized				
generated vector code single implies SSE double implies SSE2				

Table 5. The best found DFT algorithms for $n = 2^{10}$, represented as breakdown trees.

Compiler Vectorized Scalar Code. Vectorizing compilers tend to favor large loops with many iterations. Thus, the best found trees feature a top-level split that recurses into about equally large sub-problems. In Table 5, on the Pentium 4 for double precision, the code was not vectorized and thus a right-expanded tree is found.

Short Vector SIMD Code. Due to structural differences in the standard Cooley-Tukey rule (optimizing for locality) and the Short Vector Cooley-Tukey rules (trying to keep locality while supporting vector memory access), in the first recursion step the right child problem is small compared to the left child problem and the left child problem is subsequently right-expanded. This leads to good data locality for vector memory access. In Table 5, due to the larger cache sizes, on the Pentium III and Athlon XP again balanced trees are found.

5. Conclusion

We formally derived a novel variant of the Cooley-Tukey FFT that can be used to implement a complex DFT using exclusively short vector instructions. We included the FFT variant as breakdown rule into SPIRAL and automatically generated high performance DFT short vector implementations across different architectures including Pentium III/4 and Athlon XP. We achieved speed-ups compared to the best available scalar code that are close (70–80 %) to the vector length ν, and matched or exceeded the performance of the best available DFT vendor library. We showed that highest performance code is platform-specific, which confirms the need for platform-adaptation and thus code generation technologies.

We conclude by thanking Prof. Überhuber (Technical University of Vienna) and Prof. José Moura (Carnegie Mellon University) for initiating and supporting the authors collaboration.

References

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

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

[3] 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.

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

[5] S. Kral. FFTW-GEL, 2002. http://www.fftw.org/∼skral.

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

[7] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast Automatic Generation of DSP Algorithms. In *Proc. ICCS 2001*, pages 97–106. Springer, 2001.

[8] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *Journal of High Performance Computing and Applications*, 2003. To appear.

[9] P. Rodriguez. A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures. In *Proc. ICASSP*, 2002.

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

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

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