Efficient Utilization of SIMD Extensions

Franz Franchetti, Stefan Kral, Juergen Lorenz, Christoph W. Ueberhuber

(Invited Paper)

***Abstract*— This paper targets automatic performance tuning of numerical kernels in the presence of multi-layered memory hierarchies and SIMD parallelism. The studied SIMD instruction set extensions include Intel's SSE family, AMD's 3DNow!, Motorola's AltiVec, and IBM's BlueGene/L SIMD instructions.**

FFTW, ATLAS, and SPIRAL demonstrate that near-optimal performance of numerical kernels across a variety of modern computers featuring deep memory hierarchies can be achieved only by means of automatic performance tuning. These software packages generate and optimize ANSI C code and feed it into the target machine's general purpose C compiler to maintain portability.

The scalar C code produced by performance tuning systems poses a severe challenge for vectorizing compilers. The particular code structure hampers automatic vectorization and thus inhibits satisfactory performance on processors featuring short vector extensions.

This paper describes special purpose compiler technology that supports automatic performance tuning on machines with vector instructions. The work described includes (*i*) symbolic vectorization of DSP transforms, (*ii*) straight-line code vectorization for numerical kernels, and (*iii*) compiler backends for straight-line code with vector instructions.

Methods from all three areas were combined with FFTW, SPIRAL, and ATLAS to optimize both for memory hierarchy and vector instructions. Experiments show that the presented methods lead to substantial speed-ups (up to 1.8 for two-way and 3.3 for four-way vector extensions) over the best scalar C codes generated by the original systems as well as roughly matching the performance of hand-tuned vendor libraries.

***Index Terms*— Short vector SIMD, automatic vectorization, symbolic vectorization, digital signal processing, FFT.**

I. INTRODUCTION

In order to turn Moore's law (exponential growth of the numbers of transistors per area unit) into actual performance gain, modern microprocessor architectures include performance boosting features like multi-level caches (resulting in a deep memory hierarchy), data prefetching, multiple execution units and superscalar processor cores, as well as special instructions for compute-intensive applications.

One important goal in compiler research is to map high-level language programs (written, for instance, in C or Fortran) to machine code that efficiently utilizes all performance boosting features of current processors. However, some of these features are difficult to exploit without detailed knowledge of the characteristics of the underlying algorithm. This is particularly the case when conflicting optimization goals aim at different hardware features and only the algorithm's structure provides hints on how to resolve this issue.

A. Short Vector SIMD Extensions

Short vector instructions have been introduced to support compute intensive multimedia applications on general purpose processors. Originally, these instructions targeted integer computation but later were also expanded to include single and double precision floating-point computation, which makes them useful in scientific computing.

The main idea of short vector SIMD instructions is to have multiple floating-point units operating in parallel, however, restricting them to work on newly introduced vector registers only. All floating-point SIMD instruction set extensions feature constrained vector memory access, in-register data shuffling, and parallel computation. They are all based on packing two or four floating-point numbers into their vector registers. Examples include Intel's SSE family, AMD's 3DNow!, Motorola's AltiVec, and the vector extensions implemented in IBM's BlueGene/L[1] processors.

To utilize short vector SIMD extensions transparently within C and Fortran programs, automatic vectorization is a must. Vectorization is a well-studied topic: Loop vectorization originates from vector computers and the vectorization of basic blocks is related to instruction-level parallelism. However, when these well-established techniques are targeted at short vector extensions, they introduce a considerable overhead in the resulting codes. Only algorithms having very regular structure (e.g., algorithms in 3D graphics) can be mapped to short vector instructions in this way without significant overhead; in particular numerical kernels often give rise to a non-negligible amount of overhead. It is therefore not surprising that only modest speed-ups are gained by vectorizing compilers [44].

To bypass this performance bottleneck, most compilers targeting short vector extensions provide proprietary programming interfaces allowing the programmer to insert SIMD instructions manually. This approach exposes architectural features and requires programmers to deal with low-level issues like data alignment, and to select appropriate vector instructions. The use of such language extensions reduces portability of source code (by the introduction of vector extension specific optimization), and requires considerably higher programming expertise. Straightforward application of the new instructions usually leads to disappointingly low speed-ups and may even result in actual slowdowns.

Manuscript received November 15, 2003. This work was supported by the Austrian Science Funds FWF.

All authors are with the Vienna University of Technology.

[1]The 360 Tfbp/s supercomputer BlueGene/L features 65,536 CPUs and supersedes the *Earth Simulator* by an order of magnitude.

In performance optimization it is often required to trade operation counts for structure, i.e., do suboptimal computation to allow for better vectorization and globally change the algorithm's structure. A second trade-off is a satisfactory utilization of SIMD instructions versus vectorization overhead. Therefore, speed-up values close to their theoretical upper bound are often realized only by expert programmers dealing with very simple algorithms.

B. Automatic Performance Tuning

An important optimization target is the memory hierarchy of the computer used. Optimization for data locality is a must to achieve satisfactory performance on today's computer systems. Most production quality C or Fortran compilers include memory hierarchy optimization, typically by applying aggressive loop reorganization. These compilers also use profiling data gathered by running the actual program for further optimization. A vast amount of literature exists on the topic of loop optimization and feedback directed compilation (see, for instance, [47]).

However, due to the complexity of modern architectures, even the most sophisticated compiler techniques aiming at the memory hierarchy are incapable of achieving performance on a level close to that of automatic performance tuning systems like FFTW [22]–[24], SPIRAL [39], [43], [53], PHIPAC [9], and ATLAS [13], [51]. These systems use an empirical approach to automatically adapt numerical kernels to a given computer system. They search in the space of possible implementations and optimizations of their algorithm base and use actual runtimes of the generated programs as the cost function in the optimization process.

In terms of the performance of numerical kernels, an important question is how automatic performance tuning systems can be adapted to support short vector SIMD extensions. For portability reasons, these systems automatically generate and optimize C or Fortran code. The obvious solution is to let a vectorizing compiler take care of the SIMD extensions while the automatic performance tuning system is responsible for the optimization with respect to the memory hierarchy. However, this approach turns out to be too simple-minded. Two reasons for this include:

(i) Optimization dealing with loop vectorization conflicts with optimizations related to the memory hierarchy. Efficient vectorization of loops requires long vector lengths while data locality calls for a minimization of vector lengths.

(ii) Automatic performance tuning systems carry out loop unrolling and algebraic simplification, which results in large basic blocks with complicated structure. Such basic blocks are the worst case for extracting instruction-level parallelism. As a result, vectorizing compilers in tandem with automatic performance tuning systems often produce poorly performing code.

C. Contributions of this Paper

This paper introduces methods that enable state-of-the-art automatic performance tuning systems to utilize floating-point short vector SIMD extensions efficiently. Support is provided at three different levels:

1) *Code Generator Level.* This paper introduces and describes SIMD-aware vectorization algorithms for DSP transforms designed to fit the needs of automatic performance tuning systems.
2) *Vectorization of Automatically Generated Code.* The Vienna MAP vectorizer utilizes the known structure of computational kernels generated by self-tuning numerical software to translate these kernels into efficient vector code.
3) *Replacement of C Compiler.* The Vienna MAP backend translates vectorized code into high-performance vector assembly code.

The results from all three levels are connected to, and partly included, in the state-of-the-art automatic performance tuning systems SPIRAL, FFTW, and ATLAS. Moreover, FFTW-GEL [31], a specialized FFTW version including the Vienna MAP vectorizer and backend, is provided.

The main results of this paper can be summarized as follows: Our methods for utilizing SIMD extensions achieve the same level of performance as hand-tuned vendor libraries while providing performance portability. Combined with leading-edge self-tuning numerical software—FFTW, SPIRAL, and ATLAS—we have therefore produced:

(i) the fastest real and complex FFTs running on x86 machines (Pentium III, Pentium 4, and AMD processors) for certain vector lengths; (ii) the only FFT routines supporting IBM's Power PC 440 FP2 double FPU used in BlueGene/L machines; (iii) the only automatically tuned vectorized implementations of important DSP algorithms (including discrete sine and cosine transforms, Walsh-Hadamard transforms, and multidimensional transforms); (iv) the only fully automatically vectorized ATLAS kernels.

By comparing industry-standard non-adaptive implementations with the SIMD-aware automatic performance tuning that we provide in tandem with our partners' software, an overall speed-up of an order of magnitude has been achieved (this is comparable to five years of advance in hardware development).

D. Synopsis

The paper begins with a survey of the state-of-the-art in automatic performance tuning, compiler backends, and short vector SIMD vectorization. Section II reviews compiler technology that plays an important role in the context of this paper. The idea of automatic performance tuning is introduced in this section and the automatic performance tuning systems ATLAS, FFTW, and SPIRAL are reviewed. Section III provides details of current short vector SIMD extensions, including vectorization techniques and their problems when applied to short vector SIMD extensions.

The remaining sections present the novel contributions of this paper. Section IV introduces symbolic vectorization of DSP transforms. Section V explains the vectorization techniques applied in the Vienna MAP vectorizer. Section VI introduces the Vienna MAP backend. All these novel technologies are assessed experimentally in Section VII.

II. Compiler Technology and Automatic Performance Tuning

This section introduces and evaluates compiler techniques relevant to high-performance scientific computing. In addition, the concept of automatic performance tuning is introduced.

A. High-Performance Compilers

Today's production compilers for high-performance computing feature sophisticated optimization techniques to cope with the complexity of current computer systems. Optimization targets the utilization of resources within the CPU as well as memory locality, special instructions, and multiple processors.

Standard loop optimization techniques targeting locality of references and CPU resource utilization include loop pipelining, partial unrolling, exchanging, splitting, and fusing [40]. Other techniques to enhance CPU resource utilization include register allocation [2], instruction scheduling, and strength reduction [40]. Features like data prefetching, multiple CPUs, and vector instructions are addressed by various algorithms. Feedback directed compilation [47] utilizes the actual runtime behavior of a program in the optimization process. An excellent survey of compiler techniques may be found in [7].

Of these techniques, the quality of register allocation is particularly important for the compilation of numerical straight-line code. [25] assesses the simple, yet effective *farthest first* algorithm [47] in a compiler backend for MIPS processors. Experiments demonstrate that this alternative spilling strategy is superior to the standard techniques (which are based on graph coloring [2]) when compiling numerical code.

B. Automatic Performance Tuning

For numerical kernels, which demand the highest possible level of performance, traditional optimizing compilers are not sufficient when used in isolation on portable source code.

Automatic performance tuning as a new software optimization paradigm targets this problem. It is a problem specific approach to performance optimization beyond general purpose compiler optimization. Automatic performance tuning systems systems feature code generators that utilize domain knowledge to generate many alternative implementations for a given algorithm, exploiting degrees of freedom in the problem to be solved. Actual runtimes of these alternative implementations drive a search process that automatically selects the best performing algorithm for a given machine.

ATLAS. ATLAS [13], [50], [51] uses empirical techniques to produce efficient performance portable implementations of the BLAS (*basic linear algebra subprograms*). ATLAS automatically generates various implementations of a given BLAS operation and searches for optimal loop tiling, blocking, software pipelining, register blocking, instruction scheduling, etc. to find the best way of performing this operation on a given computer.

FFTW. The first effort to automatically generate high-performance FFT code was FFTW [23]. Typically, it produces code that runs faster than publicly available FFT codes and compares well to vendor libraries. A dynamic programming approach relying on a recursive implementation of the Cooley-Tukey FFT algorithm [49] provides for the adaptation of the FFT computation of a given size to a given target machine at runtime. The actual computation is done within routines called (twiddle and no-twiddle) *codelets* produced by a program generator named `genfft` [22]. The newly released FFTW 3 [24] includes results of this paper.

SPIRAL. SPIRAL [39], [43] is a code generator for high performance DSP transforms. For a given transform to be implemented, the rule-based *formula generator* produces one out of many possible fast algorithms, represented by mathematical formulas in the signal processing language SPL. The *formula translator* (SPL compiler) [53] translates these formulas into C or Fortran code. Based on measured program runtimes, the *search engine* generates alternative formulas and triggers their implementation. Searching for good implementations by iterating this process tries to bring the best out of a given machine.

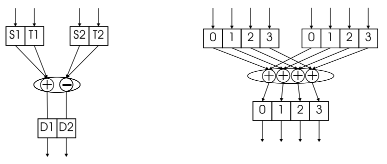


Fig. 1. **Packed Operations.** Intra-operand and parallel SIMD operations.

TABLE I

FLOATING-POINT SIMD INSTRUCTION SET EXTENSIONS. FOR EACH EXTENSION LISTED, THE VECTOR LENGTH, THE CALCULATION PRECISION, AND A LIST OF SUPPORTING SYSTEMS IS PROVIDED.

Name	Type	Processors
3DNow!	2x single	AMD Athlon, Opteron
SSE	4x single	Intel P III, P4, AMD Athlon XP, Opteron
SSE2	2x double	Intel P4, AMD Opteron
SSE3	2x double	Intel P4e
Altivec	4x single	Motorola PPC G4, IBM PPC G5
Double FPU	2x double	IBM PPC 440 FP2 (BlueGene/L)

III. Utilizing Short Vector SIMD Extensions Efficiently

Modern processors from embedded computing to supercomputers feature short vector SIMD (single instruction, multiple data) extensions. These extensions operate on vectors of basic integer and floating-point data types. The vectors allow for fine-grained parallelism. To keep the complexity of the microprocessor design low, only restricted sets of operations are supported, including instructions for memory access, arithmetic operations, and data shuffling. Although initially targeting multi-media applications, short vector SIMD technology became a major determinant in scientific computing by providing vector double-precision arithmetic.

SIMD ISA extensions supporting floating-point operations include Intel's streaming SIMD extensions (SSE, SSE 2, and SSE 3), AMD's 3DNow! and its extension called "enhanced 3DNow!", Motorola's AltiVec, and the SIMD extensions of

TABLE II
MAXIMAL THEORETICAL NUMBER OF SINGLE (s) AND DOUBLE PRECISION (d) OPERATIONS THAT CAN BE RETIRED PER CYCLE EITHER ON THE STANDARD FLOATING-POINT UNIT (FPU) OR ON THE VECTOR PROCESSING UNIT (VPU) OF GENERAL PURPOSE PROCESSORS.

Processor	dFPU	sFPU	dVPU	sVPU
AMD Athlon	2	2	n/a	4
AMD Opteron	2	2	2	4
Intel Pentium III	1	1	n/a	4
Intel Pentium 4	1	1	2	4
Motorola PowerPC G4	2	2	n/a	8
IBM PowerPC G5	4	4	n/a	8
IBM PowerPC 440 FP2	2	2	4	n/a

IBM's double floating-point unit for BlueGene/L machines. Table I gives an overview over the SIMD ISA extensions supported by machines targeted in this paper. Fig.1 presents typical examples of two-way and four-way SIMD operations.

A. Architectural Features of SIMD Extensions

Depending on the actual architecture and the precision used, floating-point SIMD extensions may boost the potential peak performance by up to a factor of four (see Table II). However, speed-up by itself does not indicate absolute performance. Implementations utilizing SIMD technology leading to good speed-up and at the same time achieving a good absolute performance are rare.

Although short vector SIMD technology is similar in concept to the technology implemented in VLIW processors, vector processors, or super scalar processors, SIMD extensions have restrictions that distinguish them from these related concepts. These restrictions include:

1) *Restricted Parallelism.* Short vector SIMD floating-point instructions and data shuffling instructions are more general than those found on vector processors, but the respective vector length is much shorter, thus distinguishing SIMD processors from vector computers. The usage of vector registers makes SIMD technology more restrictive than VLIW and superscalar designs.
An important issue is that an application's data flow may require many shuffle operations to allow for parallel computation of arithmetic instructions. Complicated data flow requires more shuffle instructions, which can lead to actual performance degradation.
2) *Memory Access Peculiarities.* Memory access on processors with short vector SIMD extensions is dominated by the memory hierarchy. Systems featuring such processors behave like modern superscalar processors and thus require optimization for the memory hierarchy. This is in contrast to the behavior of conventional vector computers which do not feature a memory hierarchy.
Another major restriction is that only properly aligned data can be loaded into vector registers efficiently. Unaligned or non-unit-stride memory access can therefore lead to extremely poor performance.

Due to the inherent restrictions of SIMD extensions, only very regular kernels may lead to satisfactory speed-up. Even when SIMD optimizations in theory provide large speed-up (up to a factor of 4 on Pentium 4), even highly-tuned applications may see only a fraction of this. Nevertheless, not using SIMD extensions is throwing away a good part of the performance of your machine.

The techniques introduced in this paper overcome the restrictions of SIMD extensions and lead to demonstrable speed-up factors of up to 1.8 for two-way SIMD extensions and 3.3 for four-way extensions as compared with the best-performing scalar FFT codes. In addition, our codes are twice as fast as the fastest codes obtained with vectorizing compilers. This impressive performance improvement has been obtained despite the fact that the structure of FFT computation does not fit well to the architectural features of SIMD extensions.

B. Vectorizing Codes for Short Vector SIMD Extensions

Currently available methods for producing programs that are able to utilize short vector SIMD instructions can be categorized as follows:

1) *Interface Used.* Currently, there are three commonly-used interfaces available to programmers who want to utilize SIMD instructions: (i) Portable high-level language in tandem with vectorizing compilers, possibly requiring hints (pragmas) on how to vectorize, (ii) proprietary C language extensions that provide explicit access to short vector SIMD extensions on source level, and (iii) assembly language.
2) *Who Vectorizes.* The actual vectorization process can be done explicitly by programmers. Contrary to this approach is the use of vectorizing compilers (which may or may not be hinted by programmers) to extract parallelism from portable programs. As a third option, program generators may generate innately vectorized codes.
3) *Structures Vectorized.* Vectorization methods either extract parallelism from independent loop iterations or extract instruction-level parallelism from basic blocks.
4) *Generality of Approach.* Some vectorization approaches are general purpose techniques (for instance, when they are applicable to any program that features loops or to any basic block) while other methods depend on the special structure of given algorithms like complex FFTs.

All these approaches involve tradeoffs between portability and generality on the one hand and achievable performance on the other.

Hand-coding. For FFTs, there exist well-known hand-coded vector algorithms like Stockham's FFT algorithm [29], [48] and vector computer libraries like the SCIPORT library [34]. However, without further adaptation to the memory hierarchy, these algorithms lead to disappointingly low performance on current SIMD architectures.

A general purpose hand-coding method utilizing instruction-level parallelism is to implement a program using the complex arithmetic of C99 [5] and let an appropriate compiler map the complex operations to sequences of two-way vector instructions.

To exploit an algorithm's intrinsic parallelism, ad-hoc utilization of instruction-level parallelism within a program and the hand-vectorization of loops is often performed. In this case the programmer formulates the parallelism within the algorithm using proprietary language extensions, which inhibits portability.

In the field of DSP transforms, these hand-coding approaches are used in SIMD-enabled vendor libraries (examples include Intel's MKL and IPP [28], Apple's vDSP [6] and vBigDSP [12], as well as AMD's core math library ACML [4]), application notes (Intel's split radix FFT [26]), and free implementations like the NEC V80R FFT [41] or the Linux SIMD library libSIMD [42]. SIMD-vectorized wavelet transforms are presented in [11] and a SIMD-vectorized FFT library is presented in [45].

Vectorizing Compilers. There exist many research and production-quality compilers for SIMD extensions, including Intel's C++ compiler [27], IBM's XL C compiler for BlueGene/L supercomputers [3], a vectorizing extension to the SUIF compiler [46], Codeplay's VECTOR C compiler [10], and the SWAR compiler `scc` [14], [15].

Automatic general-purpose loop-vectorizing compiler technology originates from vector computer research and is included in most vectorizing compilers [54]. These algorithms were designed for long vector lengths and other characteristics of conventional vector computers, like constant non-unit stride memory access. Some of these implicit assumptions are no longer valid for short vector SIMD extensions. In addition, vectorization- and locality-enhancing loop transformations often conflict. Compilers therefore often require user hints (for instance, by pragmas) in order to successfully vectorize loops [21].

Automatic general purpose methods based on the extraction of instruction-level parallelism in basic blocks are used in many compilers (e. g., Intel, VectorC, and IBM compilers). They originate from VLIW research [15], [35]. These algorithms search for code sub-blocks that feature parallelism. In order to map the full computation, these parallel blocks must be connected, either by scalar operations or by data shuffling operations, which can introduce considerable vectorization overhead. Due to an exploding search space, these algorithms tend to fail on large basic blocks having complicated structure. Experiments show that in this case, these algorithms may produce negligible speed-up or may even slow down the code.

A graph-based *code selection* technique for DSPs with SIMD support has been introduced in [36]. Techniques for SIMD utilization in the context of energy-aware compilation for DSPs are presented in [38].

Pre-Existing Methods Used in Code Generators. ATLAS allows for the insertion of hand-coded kernels featuring SIMD instructions into its optimization cycle. ATLAS depends on programmers contributing such hand-coded kernels for new architectures [52]. These kernels are typically coded in assembly language.

The code generator of FFTW 3 includes instruction-level vectorization for two-way vector extensions that is based on properties of complex FFTs and utilizes C language extensions. For four-way vector extensions a combination of this method and loop vectorization is applied.

C. Our Approach

This paper introduces two approaches to domain-specific vectorization. Section IV introduces a loop vectorization method for DSP algorithms. It provides a DSP-specific approach to optimizing memory access operations (in the presence of a deep memory hierarchy) and SIMD vectorization simultaneously. This method was included in the code generators and the runtime systems of experimental SIMD versions of FFTW and SPIRAL.

Section V provides a vectorizing compiler for numerical kernels. It introduces a domain specific method to extract instruction level parallelism. Domain knowledge is utilized to search for a low vectorization overhead and avoid combinatorial explosion. It is applied to the output of the automatic performance tuning systems ATLAS, FFTW, and SPIRAL.

IV. SYMBOLIC VECTORIZATION OF SIGNAL TRANSFORMS

This section introduces domain-specific vectorization techniques for DSP transforms [16]–[21]. The presented approach originates from the observation that neither original vector computer DSP transform algorithms [34] nor vectorizing compilers [27], [35] are generally capable of producing high-performance DSP transform implementations for short vector SIMD architectures, even in tandem with automatic performance tuning [21].

The intrinsic structure of DSP transforms prevents the successful application of these well-known methods. The main obstacle is that the structure of memory access (given by permutations and loop carried array references) occurring in DSP transform algorithms is incompatible with the features offered by currently available short vector SIMD target architectures.

To overcome these problems, we introduce new SIMD vectorization techniques that are designed to be used in code generators and the runtime environment of automatic performance tuning systems, specifically targeting FFTW and SPIRAL:

(i) Vectorization of general DSP transforms like Walsh-Hadamard transforms, two-dimensional discrete cosine transforms, as well as other transforms, in their recursive formulation. This method provides for vectorizing a larger class of DSP transforms by allowing some less efficient instructions if necessary to enable successful vectorization.

(ii) Vectorization of recursive Cooley-Tukey FFT algorithms of size $k\nu^2$ where ν is the vector length in the targeted SIMD extension and k may be chosen arbitrarily. This method ensures that FFTs are implemented using the most efficient vector instructions solely, across all current short vector SIMD architectures.

These vectorization methods are at the border of loop vectorization and the utilization of instruction-level parallelism. All vectorized loops feature precisely the same number of ν iterations. This approach allows the support of both the recursive nature of DSP algorithms as well as the generic SIMD extensions (with arbitrary vector lengths ν) at the same time. Thus, it overcomes the limitations of classical loop vectorization and extraction of ILP in this particular field.

The most challenging problem in our approach is the recursive break-down of DSP algorithms or the manipulation of given recursive algorithms so that all arithmetic operations occur in vectorizable loops with ν iterations, and the resulting data access patterns can be handled efficiently across all current SIMD architectures.

A. *Mathematical Framework*

We use the representation of DSP algorithms as matrix factorizations applying the Kronecker product formalism [29], [43] as language to express our vectorization algorithms. The approach is based on the fact that many Kronecker product formulas have an intuitive interpretation as programs [29], [53].

The *tensor* or *Kronecker product* defined as

$$A \otimes B = \left(a_{i,j}\, B\right)_{\substack{i=0,\dots,m-1\\ j=0,\dots,n-1}} \quad \text{for} \quad A = \left(a_{i,j}\right) \in \mathbb{C}^{m,n}$$

is the most important construct to describe structure in DSP algorithms. Tensor products with identity matrices—as in (1)—describe the multiple application of the other factor to different data sets, thus expressing data parallelism.

A (linear) DSP transform is a multiplication of the sampled signal vector $x \in \mathbb{C}^n$ by a transform matrix $M \in \mathbb{C}^{n\times n}$. A particularly important example is the discrete Fourier transform (DFT), whose transform matrix, for size n, is given by

$$\mathrm{DFT}_n = \left(\omega_n^{k\ell}\right)_{k,\ell=0,1,\dots,n-1}, \quad \omega_n = e^{2\pi i/n}.$$

Other important examples of DSP transforms are the discrete sine/cosine transforms (DSTs and DCTs) and the Walsh-Hadamard transform (WHT), as well as their multidimensional analogs. Fast algorithms for DSP transforms can be represented as structured sparse factorizations of the transform matrix. The Cooley-Tukey fast Fourier transform (FFT) is a recursion that computes DFT_{mn} from the smaller DFT_m and DFT_n [49]

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

where I_n is the $n \times n$ identity matrix, T_n^{mn} the complex *twiddle* diagonal matrix, and L_n^{mn} the *stride permutation* matrix. Recursive application of rules like (1) yields a fast algorithm. The degree of freedom in choosing a factorization of the transform size in each step leads to a large number of mathematically equivalent formulas with similar arithmetic cost, but different structure (data flow).

To describe algorithms for short vector SIMD extensions, a formal translation from complex matrices into real matrices is required [17]. The complex multiplication $(u+iv)\times(a+ib)$ is equivalent to a real matrix-vector product,

$$(u+iv)\times(a+ib) \cong \begin{pmatrix} u & -v \\ v & u \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}, \qquad (2)$$

inducing the definition of the operator $\overline{(\;)}$. Formally, the complex matrix-vector multiplication $M\,x \in \mathbb{C}^n$ is translated into $\overline{M}\,\overline{x} \in \mathbb{R}^{2n}$, where $\overline{M} \in \mathbb{R}^{2n\times 2n}$ arises from M by replacing every entry $u+iv$ by the 2×2 matrix in (2) and $\overline{x} \in \mathbb{R}^{2n}$ is obtained by replacing each entry $a+ib$ of x by the two-dimensional real vector in (2).

TABLE III
MAPPING BLOCK MATRICES TO VECTOR CODE

Matrix	SSE ($\nu=4$)	SSE 2 ($\nu=2$)	AltiVec ($\nu=4$)
$(\mathrm{I}_\nu \,\vert\, \mathrm{I}_\nu)$	`addps`	`addpd`	`vaddfp`
$(\mathrm{I}_\nu \,\vert\, -\mathrm{I}_\nu)$	`subps`	`subpd`	`vsubfp`
$\mathrm{diag}(a_0,\dots,a_{\nu-1})$	`mulps`	`mulpd`	`vmaddfp`
$\mathrm{L}_\nu^{2\nu}$, $\mathrm{L}_\nu^{\nu^2}$	`unpcklps` `unpckhps`	`unpcklpd` `unpckhpd`	`vmrghw` `vmrglw`
$\mathrm{L}_2^{2\nu}$	`shufps`	`shufpd`	`vperm`

Algebraic manipulation of formulas by applying matrix identities allows to change the data flow of a fast algorithm [29]. Important examples in the context of this paper include stride permutation matrix factorizations like

$$\mathrm{L}_n^{kmn} = (\mathrm{L}_n^{kn} \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}_n^{mn}) \quad \text{and} \quad \mathrm{L}_{mn}^{kmn} = \mathrm{L}_m^{kmn}\, \mathrm{L}_n^{kmn},$$

as well as the *conjugation* $M^P = P^{-1}MP$ of a matrix M by a permutation P as applied to a tensor product by

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

and manipulation rules for the bar operator,

$$\overline{A\,B} = \overline{A}\,\overline{B}, \quad \overline{A_m \otimes \mathrm{I}_{n\nu}} = \left(\overline{A_m \otimes \mathrm{I}_n} \otimes \mathrm{I}_\nu\right)^{(\mathrm{I}_{mn} \otimes \mathrm{L}_2^{2\nu})}.$$

Different data layouts for complex numbers can be expressed in terms of the bar operator and conjugation. The full set of identities required to derive the formal vectorization algorithm for DSP transforms as well as the full derivation can be found in [17].

B. *Vectorizable Formulas*

Formulas describing DSP transforms are viewed as being built from certain block matrices with block size $\nu \times \nu$ and matrices that are not such block matrices. The non-block matrices are either mapped to less efficient extension specific vector code or to scalar code.

Block matrices are either built from diagonal matrices of size $\nu \times \nu$ or they are special permutation matrices. Later on, the block matrices are mapped to highly efficient vector code independently of the target machine's short vector architecture. Their $\nu \times \nu$ block structure provides that all memory access operations are properly aligned vector loads and stores. All data reordering is done in the vector registers and all arithmetic operations are pure vector operations. See Table III for vector instructions and examples of corresponding block matrices. The remainder of this section gives the particulars of the two types of block matrices.

Compute Matrices. All arithmetic operations are done in matrix-vector operations on vectors of length ν. Their building blocks are the real $\nu \times \nu$ diagonal matrices

$$0_\nu, \quad \mathrm{I}_\nu, \quad -\mathrm{I}_\nu, \quad \text{and} \quad \mathrm{diag}(a_0,\dots,a_{\nu-1}),$$

where 0_ν denotes the $\nu \times \nu$ zero matrix.

One of the most important constructs in this class of block matrices is the tensor product

$$A \otimes \mathrm{I}_\nu, \quad A \in \mathbb{R}^{m\times n}. \qquad (3)$$

```
__m128 t0, t1, x[2], y[2];
t0 = _mm_unpacklo_ps(x[0], x[1]);
t1 = _mm_unpackhi_ps(x[0], x[1]);
y[0] = _mm_add_ps(t0,t1);
y[1] = _mm_sub_ps(t0,t1);
```

Fig. 2. **Implementation of $y = (\mathrm{DFT}_2 \otimes \mathrm{I}_4)\,\mathrm{L}_4^8\,x$ for the 4-way vector extension SSE using Intel's C++ compiler intrinsics.**

To obtain $A \otimes \mathrm{I}_\nu$, all entries $a_{i,j}$ in A are replaced by

$$\mathrm{diag}(a_{i,j}, \ldots, a_{i,j}) \quad \in \quad \mathbb{R}^{\nu\times\nu}.$$

It is crucial to our approach that the vectorization of constructs of the form $A \otimes \mathrm{I}_\nu$ is done independently of the actual Kronecker formula describing the structure of A.

To describe block matrices arising from complex diagonal matrices with entries $c_i \in \mathbb{C}$, the operator $\overline{(\ )}'$ defined as

$$\overline{D}' := \overline{D}^{\left(\mathrm{I}_k \otimes \mathrm{L}_\nu^{2\nu}\right)} \quad \text{for} \quad D = \mathrm{diag}(c_0, \ldots, c_{k\nu-1}) \tag{4}$$

is used. The conjugation of the real matrix $\overline{D}$ by $\mathrm{I}_k \otimes \mathrm{L}_\nu^{2\nu}$ produces a matrix built from real diagonal matrices of size $\nu \times \nu$. As an example, consider the matrix derived from $D_0 = \mathrm{diag}(a_0 + ib_0, \ldots, a_{\nu-1} + ib_{\nu-1})$:

$$\overline{D_0}' = \begin{pmatrix} \mathrm{diag}(a_0, \ldots, a_{\nu-1}) & \mathrm{diag}(-b_0, \ldots, -b_{\nu-1}) \\ \mathrm{diag}(b_0, \ldots, b_{\nu-1}) & \mathrm{diag}(a_0, \ldots, a_{\nu-1}) \end{pmatrix}. \tag{5}$$

An important block matrix is $\overline{\mathrm{T}}'^{mn}_{n}$, which is built from diagonals of size $\nu \times \nu$ and does *not* originate from a tensor product of the twiddle diagonal T_n^{mn}.

Permutation Matrices. To be considered a block matrix, a permutation matrix P must be the product

$$(U \otimes \mathrm{I}_\nu)(\mathrm{I}_k \otimes W)(V \otimes \mathrm{I}_\nu), \quad \nu \text{ divides the size of } W, \tag{6}$$

built from the permutation matrices U, V, and W.

Permutations P that are of type (6) only require vector memory access (addressing encoded in U and V) and a moderate number (depending on W) of register-to-register data reordering operations, independently of the target architecture.

All other permutations must be implemented using scalar code or less efficient vector code.

C. Symbolic Vectorization Algorithm

Symbolic vectorization of DSP transforms translates the problem of vectorizing DSP transform algorithms into the problem of generating efficient scalar code for DSP transforms. First, formula manipulation is used to transform a DSP algorithm (given in Kronecker product notation) into a vectorizable algorithm with similar characteristics. This new transform algorithm is then implemented using vector instructions by utilizing existing code generators (FFTW's `genfft` [22] and SPIRAL's SPL compiler [53]) and newly developed vector-specific extensions to these code generators.

Our formal vectorization approach uses tensor products as core constructs. Diagonal matrices and permutation matrices are vectorized with respect to tensor products. All vectorized constructs are transformed into *symbols* S, defined by

$$\mathrm{S} = PD(A \otimes \mathrm{I}_\nu)EQ \tag{7}$$

with permutation matrices P and Q. D and E are block matrices originating from real or complex diagonals. A is an arbitrary formula in Kronecker product notation.

The implementation of a symbol S is centered around the implementation of A. First, existing code generators are utilized to generate efficient vector code for $A \otimes \mathrm{I}_\nu$ by generating efficient scalar code for A and then replacing each scalar instruction by the respective vector instruction (for instance, `c = a+b` is replaced by `c = vec_add(a,b)`). In the respective code for $A \otimes \mathrm{I}_\nu$ all vector load and store operations are then replaced by the arithmetic operations required by D and E and the data reorganization operations required by P and Q. Provided P and Q match (6), this approach leads to an efficient vector implementation of S. For example, Fig. 2 shows the generated code for the symbol $\mathrm{S} = (\mathrm{DFT}_2 \otimes \mathrm{I}_4)\,\mathrm{L}_4^8$.

D. Short Vector Cooley-Tukey Vectorization

The most crucial part in achieving good performance by symbolic vectorization is to make the permutations P and Q of a symbol S match (6) while keeping the changes to the original formula minimal. For many transform algorithms like Walsh-Hadamard transforms or two-dimensional transforms this is an easy task. In case of the vectorization of Cooley-Tukey FFTs, however, a more sophisticated approach is required.

The short vector Cooley-Tukey rules given by (8) to (10) solve this problem for FFTs of size $N = N_1\nu^2$ with arbitrary N_1. All matrices occurring in FFT algorithms vectorized by (8) to (10) are block matrices with diagonal matrices of size $\nu \times \nu$ as blocks. All permutations match (6). When applying the short vector Cooley-Tukey rules, the SIMD enabled versions of FFTW and SPIRAL optimize efficient SIMD implementations for the memory hierarchy by simultaneously searching for the best factorization of N_1 and for the best implementation of constructs $\overline{\mathrm{DFT}_r \otimes \mathrm{I}_s} \otimes \mathrm{I}_\nu$. We also derived a transposed version and versions for different complex data formats. (9) originates from the vector recursion of FFTW 3 [24].

The short vector Cooley-Tukey "entry rule" is given by

$$\overline{\mathrm{DFT}}_{\nu^2 m k_1 \cdots k_s n} \quad = \quad \mathrm{S}_0 \; R_1 \tag{8}$$

translating $\overline{\mathrm{DFT}}_N$ into the symbol

$$\mathrm{S}_0 := P_0\big(\overline{\mathrm{DFT}_{\nu m} \otimes \mathrm{I}_{k_1 \cdots k_s n}} \otimes \mathrm{I}_\nu\big)\overline{\mathrm{T}}'^{\nu^2 m k_1 \cdots k_s n}_{\nu k_1 \cdots k_s n}$$

with

$$P_0 := \mathrm{I}_{\nu m k_1 \cdots k_s n} \otimes \mathrm{L}_\nu^{2\nu}$$

and the recursive part R_1. Any recursive part

$$R_i := \big(\mathrm{I}_{\nu m k_i \cdots k_s n} \otimes \mathrm{L}_2^{2\nu}\big)\,\overline{(\mathrm{I}_{\nu m} \otimes \mathrm{DFT}_{\nu k_i \cdots k_s n})\,\mathrm{L}_{\nu m}^{\nu^2 m k_i \cdots k_s n}}$$

is further factorized using the recursive rule

$$R_i = (\mathrm{I}_{\nu m} \otimes \mathrm{S}_i)\,P_i\,(\mathrm{I}_{k_i} \otimes R_{i+1})\,Q_i \tag{9}$$

with

$$P_i := \mathrm{L}_{\nu m}^{\nu k_i m} \otimes \mathrm{I}_{2\nu k_{i+1} \cdots k_s n}, \quad Q_i := \mathrm{L}_{k_i}^{\nu k_i \cdots k_s n} \otimes \mathrm{I}_{2\nu m},$$

leading to a symbol

$$\mathrm{S}_i := \left(\overline{\mathrm{DFT}_{k_i} \otimes \mathrm{I}_{k_{i+1}\cdots k_s n}} \otimes \mathrm{I}_\nu\right) \overline{\mathrm{T}}'^{\nu k_i \cdots k_s n}_{\nu k_{i+1} \cdots k_s n}$$

and a new recursive part R_{i+1}. Considering the factorization $N_1 = mk_1 \cdots k_s n$, the i^{th} application of rule (9) "consumes" the factor k_i by breaking the factor $mk_i \cdots k_s n$ recursively into k_i and $mk_{i+1} \cdots k_s n$. The recursion ends when all factors k_i are "consumed" by the application of rule (9), finally leading to the last recursive part

$$R_{s+1} := \left(\mathrm{I}_{\nu mn} \otimes \mathrm{L}_2^{2\nu}\right) \overline{\left(\mathrm{I}_{\nu m} \otimes \mathrm{DFT}_{\nu n}\right) \mathrm{L}_{\nu m}^{\nu^2 mn}}.$$

The application of the leaf rule,

$$R_{s+1} = \Big(\mathrm{I}_m \otimes P_{s+1}\big(\overline{\mathrm{DFT}}_{\nu n} \otimes \mathrm{I}_\nu\big)\Big) Q_{s+1} \tag{10}$$

with

$$P_{s+1} := \left(\mathrm{L}_\nu^{2\nu n} \otimes \mathrm{I}_\nu\right)\left(\mathrm{I}_{2n} \otimes \mathrm{L}_\nu^{\nu^2}\right), \quad Q_{s+1} := \mathrm{L}_m^{\nu mn} \otimes \mathrm{L}_2^{2\nu}$$

transforms R_{s+1} into the required form to make the permutations P_{s+1} and Q_{s+1} match (6).

V. Vectorization Techniques for Straight-Line Code

This section introduces the Vienna MAP vectorizer [32], [33], [37] that automatically extracts 2-way SIMD parallelism out of given numerical straight-line code. The MAP vectorizer additionally supports the extraction of SIMD fused multiply-add instructions.

MAP has been applied successfully to straight-line code produced by FFTW, SPIRAL, and ATLAS. Thus, a large variety of numerical computations ranging from FFTs and other DSP transforms to BLAS kernels can be vectorized automatically.

A. Fundamentals of Vectorization

Existing approaches to vectorizing basic blocks [15], [35], [36] try to find an efficient mix of SIMD and scalar instructions to do the required computation, MAP's vectorization mandates that *all* computation is performed by SIMD instructions, while attempting to keep the SIMD reordering overhead reasonably small.

MAP's vectorizer uses depth-first search with chronological backtracking to discover SIMD style parallelism in a scalar code block, aiming at a reduction of the overall instruction count. Obtaining a satisfactory SIMD utilization is tantamount to minimizing the amount of SIMD data reorganization while maximizing the coverage of the scalar DAG by natively supported SIMD instructions.

Fig. 3 gives an example of short vector SIMD code obtained by vectorizing straight-line complex FFT code.

B. The Vectorization Engine

The central goal of vectorization is to replace all scalar instructions by vector instructions. The description of the vectorization technique relies on the following definitions.

Operation Pairing. Pairing rules specify ways of transforming pairs of scalar instructions into a sequence of SIMD instructions. A pairing rule often provides several alternatives to

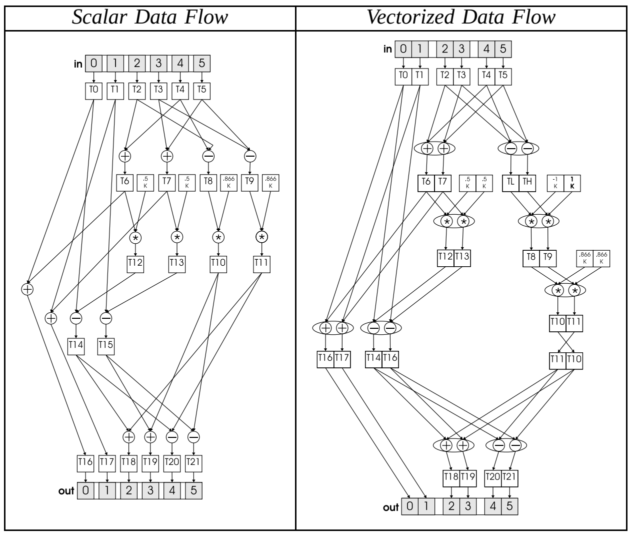


Fig. 3. **Vectorization of a Scalar FFT of Size 3.** The scalar data fbw in the left part of the illustration is computationally equivalent to the vectorized data fbw depicted in the right part.

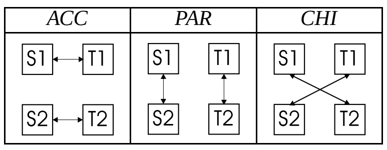


Fig. 4. **Fusion Layouts.** Three layouts for fusing the source variables of the scalar instructions `(op1,S1,T1,D1)` and `(op2,S2,T2,D2)` are supported by the MAP vectorization engine. Intra-operand SIMD instructions are supported by the *ACC* layout, whereas the layouts *PAR* and *CHI* are used for inter-operand SIMD instructions.

do so. The vectorizer supports pairing rules of the following instruction combinations: (i) load/load, (ii) store/store, (iii) unary/unary, (iv) binary/binary, (v) unary/binary, (vi) unary/load, and (vii) load/binary.

Variable Fusion. Two scalar variables `s`, `t` can be fused to a SIMD variable of the form `st = (s,t)` or `ts = (t,s)`. The vectorization process ensures that no scalar variable is part of two different SIMD variables.

C. Vectorization Quality

To produce vector code of the highest quality, the vectorization engine starts out by constraining all SIMD memory operations to access consecutive locations and by disabling sub-optimal operation pairing rules of type (v), (vi), and (vii).

If these restrictions cause the vectorization process to fail, it is restarted after enabling operation pairing rules of type (v), (vi), and (vii), and the support for less efficient, i. e., non-consecutive, memory accesses. This step substantially extends the class of vectorizable codes by allowing the extraction of some less efficient instruction combinations.

In the worst case, a fallback to the vector implementation of scalar code is made by leaving half of each SIMD instruction's capacity unused. On all surveyed x86 machines, the resulting codes are faster than the scalar legacy x87 codes generated by standard general purpose compilers.

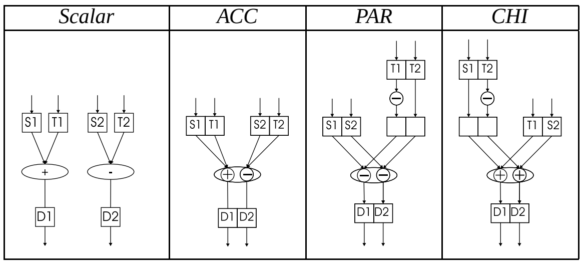


Fig. 5. **Vectorization Alternatives.** Two scalar instructions, one addition and one subtraction, are transformed into a sequence of SIMD instructions in three different ways.

D. The Vectorization Algorithm

Before the actual vectorization process is carried out, the following preparatory steps are taken. First, dependencies of the scalar DAG are analyzed and instruction statistics are assembled. This data is used to speed up the vectorization process by avoiding useless vectorization. Then, store instructions are combined non-deterministically by fusing their respective source operands.

The actual vectorization algorithm consists of two steps:

(i) Pick two scalar instructions `I1=(op1,s1,t1,d1)` and `I2=(op2,s2,t2,d2)` that have not been vectorized, with `(d1,d2)` or `(d2,d1)` being an existing fusion.

(ii) Non-deterministically pair the two scalar operations into one SIMD operation. This step may produce new fusions for the respective source operands.

The vectorizer alternatingly applies these two steps until either the vectorization succeeds, i. e., thereafter all scalar variables are part of at most one fusion and all scalar operations have been paired, or the vectorization fails. If the vectorizer succeeds, it commits to the first solution of the search process.

Non-determinism in vectorization arises due to different vectorization choices. For a fusion `(d1,d2)` there may be several ways (see Fig. 4) of fusing the source operands `s1,t1,s2,t2`, depending on the pairing `(op1,op2)`, as depicted in Fig. 5.

The *rule ranking*, i. e., the order in which vectorization alternatives are tried, influences the order of the solutions of the vectorization process. As the vectorizer always commits to the first solution, the rule ranking is adapted in such a way that the first solution favors instruction sequences which are particularly well-suited for a given target machine. For instance, on an AMD K7 processor the vectorizer prefers extracting intra-operand (*ACC*) over inter-operand (*PAR*, *CHI*) style SIMD instructions (see Fig. 5).

E. Peephole Based Optimization

After vectorization, a local rewriting system is used to implement peephole optimization on the vector DAG.

The first group of rewriting rules aims at (i) minimizing the number of instructions, (ii) eliminating redundancies and dead code, (iii) reducing the number of source operands, (iv) copy propagation, and (v) constant folding.

The second group of rules can be used to extract SIMD-FMA instructions.

The third group of rules rewrites unsupported SIMD instructions into sequences of SIMD instructions actually supported by the target architecture.

Finally, the optimizer topologically sorts the instructions of the vector DAG. The scheduling algorithm minimizes the lifespan of variables by improving the locality of variable accesses. It is based on the scheduler of `genfft` [22].

VI. Backend Techniques for Straight-Line Code

The Vienna MAP backend [32], [33] introduced in this section generates assembly code optimized for short vector SIMD hardware. It is included in an experimental version of FFTW, FFTW-GEL [31], and has been connected to SPIRAL and ATLAS. Currently supported targets include x86/3DNow! and x86/SSE2. A PowerPC version is currently being developed.

Like in [25], the MAP backend uses the farthest first algorithm as its spilling policy in the register allocator. Additionally, as the MAP backend does not target a broad range of structurally different hand-written code, it makes use of domain specific meta information by exploiting the following properties of array access operations occurring in automatically generated straight-line code. (i) Any memory access is indexed, possibly with a stride as runtime parameter, and (ii) each memory location is read/written at most once.

A. Main Parts of the Backend

The MAP backend performs *ISA specific optimization* in (i) register allocation, (ii) computation of effective addresses, (iii) usage of in-memory operands, and (iv) register reallocation. ISA specific optimization addresses general properties such as the number (and type) of available logical registers, available instruction forms, and the existence of special instructions like `lea` (on x86).

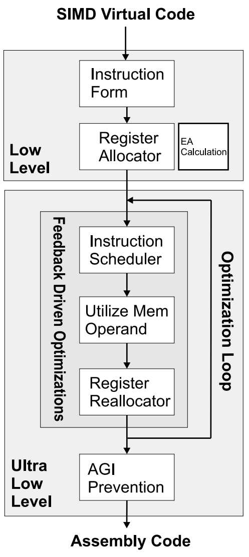


Fig. 6. **The basic structure of the MAP backend.** A SIMD DAG generated by MAP's vectorization frontend is optimized and compiled to assembly code.

The instruction scheduler performs *processor specific optimization* by taking into account execution properties provided by a processor-specific execution model. This model specifies required execution resources, available resources, the maximum number of instructions issued at each clock cycle, and instruction latencies.

The last optimization step of MAP's backend is responsible for the avoidance of address generation interlocks (AGIs).

B. Nonrecurring Optimizations

Register allocation, computation of effective addresses, and AGI prevention are optimization techniques performed only once. Register allocation is followed by instruction scheduling, which tries to maximize the usage of the functional units and the pipeline of the processor on hand.

Register Allocation for Straight-Line Code. While there is no single best phase ordering in the context of general purpose compilers targeting hand-written code, MAP's backend assumes that the input code has been reasonably scheduled on a higher level using domain-specific meta information. This kind of preprocessing can be done, for example, like `genfft` [22] does, by performing an FFT-specific topological sort of the computation DAG in an attempt to minimize the asymptotical number of register spills.

As any future access to temporary variables is known in advance, the MAP backend uses the farthest first algorithm [8] as its spilling heuristic. This means that whenever a temporary variable needs to be mapped to a logical register, the following strategy is used: (i) Whenever possible, choose a fresh logical register. (ii) If there is at least one dead logical register, i. e., a logical register holding a value that is not referenced in the future, choose the one logical register that has been dead for the longest time. Notice that reusing any dead logical register introduces a false dependency. (iii) If there are no dead logical registers, choose that logical register R, which holds a value V whose reference lies farther in the future than all other references to logical registers.

Optimized Index Computation. Index computation is not normally a relevant issue in performance optimization of non-DSP code. However, straight-line codes produced by DSP program generators feature an unusually large proportion of memory access operations in relation to arithmetic operations. Thus, optimized index computation is crucial to achieving high performance in this case.

MAP targets code that operates on arrays of input and output data not necessarily stored contiguously in memory. Thus, access to an array element `a[i]` may result in a memory access operation either at address `a+i*sizeof(float)` or `a+i*sizeof(float)*stride`, where `a` and `stride` are parameters passed from the calling function. Memory access operations with constant stride do not need explicit effective address computation, whereas those with variable stride do.

To achieve an efficient effective address computation, the usage of general integer multiplication instructions is avoided by performing a combination of strength reduction and common subexpression elimination in the register allocator.

Whenever the register allocator needs to emit code for the calculation of an effective address, the (locally) shortest possible sequence of simple instructions (add, sub, shift, lea) is determined by forward chaining using depth-first iterative deepening with a depth limit that depends on the latency of the integer multiplication instruction on the given target architecture. In case the depth limit is exceeded, strength reduction is not applied.

As the shortest sequences of code doing effective address calculation tend to eagerly reuse already calculated contents of the integer register file as factors, a replacement policy based on the LRU heuristic is employed in the allocation of integer registers.

Prevention of AGIs. Although modern x86 compatible processors allow the out-of-order execution of instructions, the definition of Intel's Pentium architecture mandates that instructions accessing memory, i. e., loads and stores, must be executed *in-order*. This requirement is the reason why address generation interlocks (AGIs) occur. Whenever a memory-operation in need of additional effort for effective address calculation (in scaled index-addressing mode) directly precedes a memory operation not requiring additional effort, both memory operations are delayed for the duration of the more expensive address calculation—an AGI occurs. The MAP backend tries to avoid such AGIs by reordering the affected instructions after leaving the feedback-driven optimization loop.

C. Feedback Driven Optimizations

The instruction scheduler and the register reallocator form a feedback driven optimization loop. The instruction scheduler serves as a basis for estimating the runtime of the entire basic block. As long as the code's estimated execution time can be improved, feedback-driven optimization is carried out.

Basic Block Instruction Scheduling. To maximize the overall performance, the respective code has to be scheduled in a way to take maximum advantage of the pipelines provided by the architecture [40], [47]. Instruction scheduling is an optimization technique that rearranges the micro-operations executed in a processor's pipeline, attempting to maximize the number of functional units operating in parallel and to minimize the time they spend waiting for each other [30].

MAP's instruction scheduler deals with the instructions of a single basic block by using local list scheduling [40], [47]. The scheduling algorithm utilizes information about the critical-path lengths of the underlying data dependency graph as a heuristic when selecting an instruction to be issued. The list scheduling algorithm implemented in MAP interacts with an execution model of the target processor to simulate the effects of super-scalar in-order execution of an instruction sequence.

Register Reallocation. Register allocation and instruction scheduling have conflicting goals. As the register allocator tries to minimize the number of register spills, it prefers introducing a false dependency (by reusing a dead logical register) over spilling a live logical register. The instruction scheduler, on the other hand, tries to maximize the pipeline usage of the processor by spreading out the dependent parts of code sequences according to the latencies of the respective instructions.

As the MAP backend performs register allocation before instruction scheduling, false dependencies introduced by the register allocator severely reduce the degree of freedom of the instruction scheduler.

To address this problem, the register reallocator tries to lift some of the restrictive data dependencies introduced by register allocation, enabling the instruction scheduler to do a better job in a subsequent pass. Additionally, the register reallocator uses information about the spills introduced by the register allocator to minimize the code size by appropriately utilizing CISC-style instructions (with in-memory operands) and by optimizing x86 copy instructions.

VII. Experimental Results

This section provides experimental evidence for the impressive performance gain unleashed by the methods introduced in this paper. In particular, high-performance implementations of DSP algorithms are obtained by including support for short vector SIMD extensions to SPIRAL and FFTW.

Symbolic vectorization and the Vienna MAP vectorizer provide the fastest FFTs currently available on x86 architectures featuring 3DNow!, SSE, or SSE 2, for certain problem sizes. In addition, these methods produce the only currently available FFT software for IBM's BlueGene/L PowerPC 440 FP2 processors which takes full advantage of their double FPU. Formal vectorization and the Vienna MAP vectorizer also provide the only current vectorized implementations of general 1D and multidimensional DSP transforms that are automatically tuned, as well as supplying the only fully automatically vectorized ATLAS kernels.

A. Experimental Setup

Numerical experiments were conducted on the following machines featuring different short vector extensions: (i) a prototype of BlueGene/L's PowerPC 440 FP2 running at 500 MHz featuring a *double FPU*, (ii) various Pentium 4 machines featuring SSE and SSE 2 running at 1.8, 2, 2.6, and 3 GHz, (iii) a 1.53 GHz Athlon XP 1800+ featuring 3DNow! professional, and (iv) a 933 MHz MPC7450 G4 featuring AltiVec.

SSE and AltiVec are four-way vector extensions, while 3DNow! and IBM's double FPU are two-way vector extensions (see Table I). SSE (on Pentium 4 processors) and Altivec (on the PowerPC) provide a theoretical speed-up of four, while 3DNow! (on the AMD Athlon and Opteron) and IBM's double FPU (on the IBM PowerPC 440 FP2) provide a theoretical speed-up of two (see Table II). In addition, AltiVec and the *double FPU* provide fused multiply-add (FMA) instructions. Note that these speed-up figures do not imply anything about the theoretical performance per cycle: For instance, both Athlon XP processors with 3DNow! and Pentium 4 processors with SSE can retire four single-precision flops per cycle, despite their different SIMD vector lengths of 2 and 4, respectively.

The vectorization techniques introduced in this paper were assessed using ATLAS kernels and general DSP transforms with special focus on FFTs. The new techniques were compared to (i) scalar public-domain non-adaptive FFT implementations (Numerical Recipes and the GNU scientific library GSL 1.4; both without SIMD support), (ii) vendor libraries with SIMD support (Intel's MKL 6.1 and IPP 4.0), (iii) vectorizing compilers performing loop vectorization and extracting instruction level parallelism (IBM's XL C compiler for BlueGene/L and Intel's C++ compiler 8.0), as well as (iv) the scalar version of SPIRAL (latest experimental version) and ATLAS 3.4.1, as well as both the scalar and SIMD version of FFTW 3.0.1. We used the experimentally determined best compiler flags for each machine/library combination.

Due to the variable flop count of different FFT algorithms, FFT performance is given in *pseudo Gflop/s*, i. e., $5N \log_2 N/$runtime (in nanoseconds). For all other transforms actual performance in *Gflop/s* is measured. This keeps the relation of runtime between different implementations for all transforms.

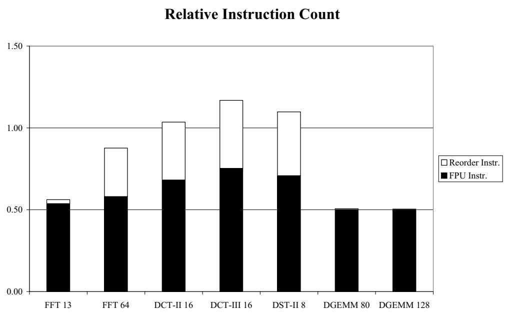


Fig. 7. Instruction counts of codes vectorized by the Vienna MAP vectorizer, relative to scalar code.

B. Main Insights

Figs. 7, 8, and 9 display a selection of instruction statistics and performance data to provide evidence for the most important experimental results.

The main insights provided by the numerical experiments can be summarized as follows:

Speed. Both symbolic vectorization and the Vienna MAP vectorizer considerably speed up computation across all the assessed machines. Fig. 8 (a) and Figs. 9 (a)–(h) show the performance of the generated codes across the test machines.

Using four-way SIMD, speed-ups of up to 3.3 have been achieved leading to FFTs running at 7.3 pseudo Gflop/s on a 3 GHz Pentium 4, displayed in Fig. 9 (a).

Using two-way SIMD, high performance FFTs running at 2.4 pseudo Gflop/s on a 2 GHz Pentium 4 have been achieved, as displayed in Fig. 9 (e).

Versatility. High speed-up values are maintained across a large variety of transforms: real and complex FFTs, both for power of two as well as arbitrary vector lengths displayed in Figs. 8 (a) and Fig. 9 (a), (c)–(f). More general transforms like 2D DCTs have been sped up as well, as shown in Fig. 9 (b).

The Vienna MAP vectorizer is able to vectorize ATLAS kernels (illustrated in Fig. 9 (h)) and FFT codes (Figs. 9 (e)-(g)). The relative instruction counts displayed in Fig. 7 show that by using SIMD instructions the total number of arithmetic instructions is in some cases reduced by 50 %, and in most cases reduced significantly. Slightly increased instruction counts may occur in some cases, which does not introduce severe problems, as the execution units for SIMD data shuffling and SIMD floating-point operations can operate in parallel.

Vectorizing Compilers and Portable Non-adaptive FFT Libraries. Both tested portable *non-adaptive* FFT libraries (Numerical Recipes and GNU GSL) feature optimized algorithms but are much slower than the best scalar *adaptive* code. They are slower than the best SIMD vectorized codes by a factor of five to ten, as can be seen in Figs. 8 (a), 9 (a), and (e).

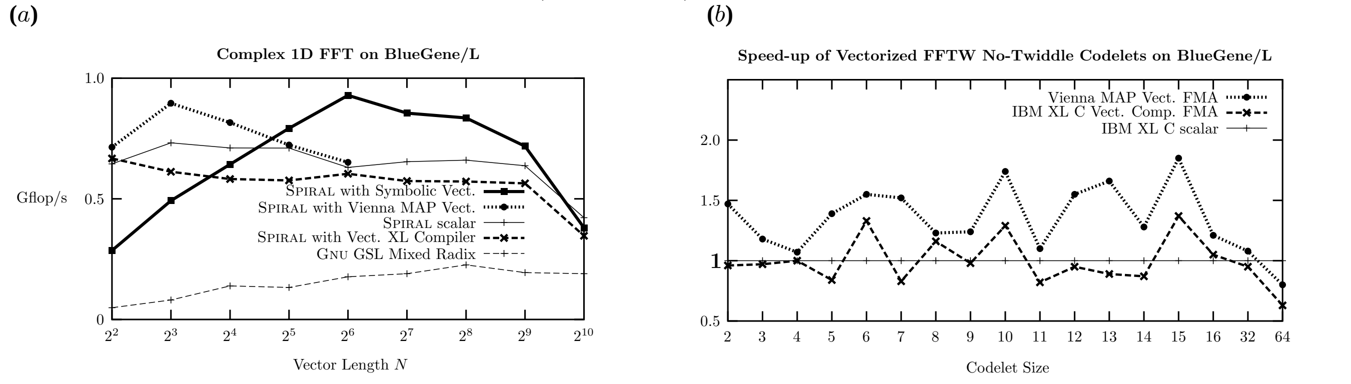


Fig. 8. Floating-point performance and speed-up of vectorized FFT implementations compared to scalar implementations on BlueGene/L's PowerPC 440 FP2 running at 500 MHz. (*a*) Assessment of both symbolic vectorization and the Vienna MAP vectorizer in tandem with SPIRAL. (*b*) Assessment of the vectorization quality of the Vienna MAP vectorizer and IBM's XL C compiler both applied to FFTW no-twiddle codelets.

These experiments provide evidence that modern (vectorizing) compilers are not able to generate fast machine code in conjunction with portable *non-adaptive* libraries.

Vectorizing Compilers and Automatic Performance Tuning. Automatic performance tuning in combination with vectorizing compilers does not, in practice, reach the same level of performance as hand tuned code and code generated using symbolic vectorization or the Vienna MAP vectorizer and backend. In the best case, when using loop vectorization, Intel's C++ compiler reaches 60 % of the speed-up of symbolic vectorization, as demonstrated in Figs. 9 (*a*)–(*b*).

For automatically generated FFT codes, vectorization of basic blocks using instruction level parallelism sometimes achieves a slight acceleration, but in other cases leads to a significant performance degradation. This effect is especially striking when using IBM's XL C compiler for BlueGene/L's PowerPC 440 FP2 processor, as displayed in Figs. 8 (*a*)–(*b*). In all these cases the Vienna MAP vectorizer achieves satisfactory speed-up values.

Hand Tuned Libraries with SIMD Support. Symbolic vectorization and the MAP vectorizer provide about the same performance as vendor-supplied hand tuned libraries that are obtained in a laborious and costly process. In contrast, the presented techniques utilize modern software technology to automatically adapt code to a given machine. The resulting performance behavior is illustrated by Figs. 9 (*a*), (*d*), and (*e*). The highly satisfactory performance level achieved by the MAP vectorizer when applied to linear algebra kernels (as illustrated by Fig. 9 (*h*)) is the same as that achieved by ATLAS.

Effect of the MAP Backend. The Vienna MAP backend accelerates automatically generated DSP code by up to 25 %, compared with standard C compilers like the GNU C and Intel C++ compiler, as displayed in Fig. 9 (*g*).

VIII. CONCLUSION

This paper presents special purpose compilation techniques targeting the SIMD vectorization of DSP transforms and straight-line code, both in the context of automatic performance tuning.

The paper addresses the difficulty of optimizing for SIMD extensions and memory hierarchy simultaneously. The problems of generic loop vectorization and instruction-level parallelism extraction present in state-of-the-art vectorizing compilers are avoided by utilizing knowledge of the properties of the algorithms underlying the programs to be vectorized.

Most approaches described in literature try to find a good SIMD coverage, and have to trade the scope of SIMD application against low vectorization overhead. In contrast, the presented techniques guarantee full SIMD coverage and a very small vectorization overhead for most important BLAS and DSP kernels. This is achieved by utilizing knowledge about the algorithms at three different layers:

(*i*) Symbolic vectorization of DSP transforms handles a large class of DSP transforms, including FFTs, WHTs, and all multidimensional DSP transforms.

(*ii*) Alternatively, automatic vectorization of straight-line code targets the large basic blocks generated by automatic performance tuning systems.

(*iii*) Finally, a special purpose compiler for large basic blocks featuring vector instructions achieves fast object code.

The presented compiler technology obtains unusually high speed-up values—1.85 for two-way and 3.3 for four-way SIMD extensions—on top of the fastest scalar codes available, thus leading to an unprecedentedly high overall performance. In this way the performance level of hand-tuned vendor libraries is attained conjointly with performance portability.

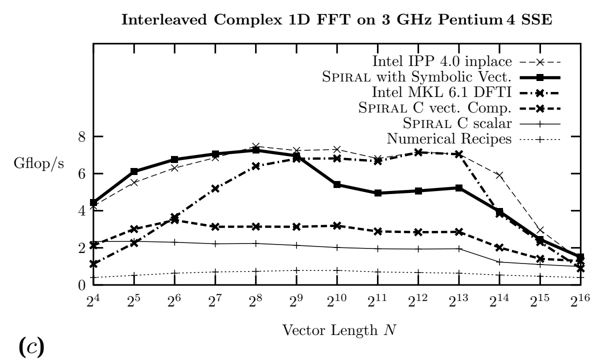
In conjunction with the leading automatic performance tuning systems SPIRAL, FFTW, and ATLAS, high performance short vector SIMD implementations of FFTs, general DSP transforms, and BLAS kernels have been obtained.

Some of the techniques described in this paper have been included in the current release of the industry-standard numerical library FFTW and will become part of IBM's numerical library for BlueGene/L supercomputers.

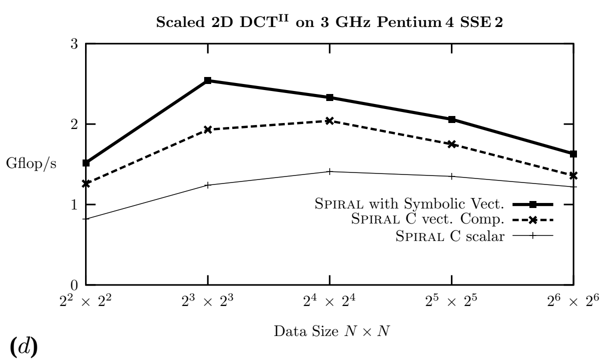
ACKNOWLEDGMENTS

We would like to thank Matteo Frigo and Steven Johnson for many years of prospering cooperation and for making it possible for us to access non-public versions of FFTW. The short vector Cooley-Tukey rule set was inspired by FFTW's vector recursion which Matteo Frigo pointed out to us.

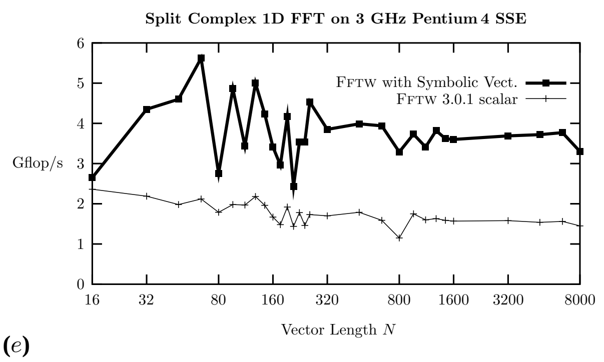
(*a*)



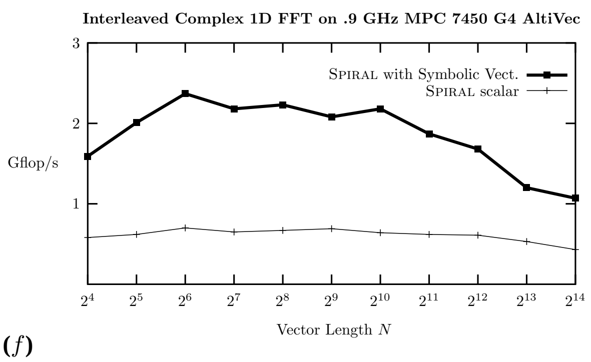
(*b*)



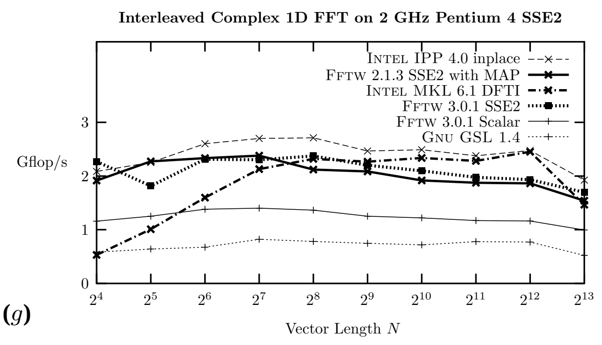
(*c*)



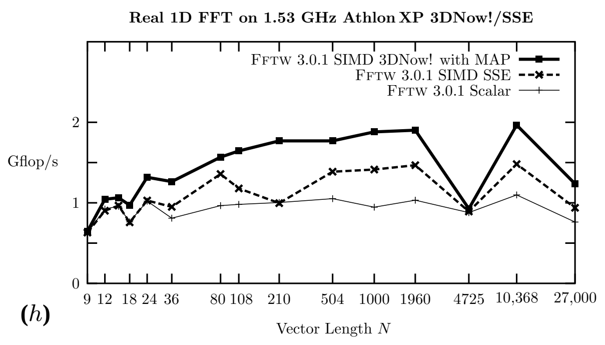
(*d*)



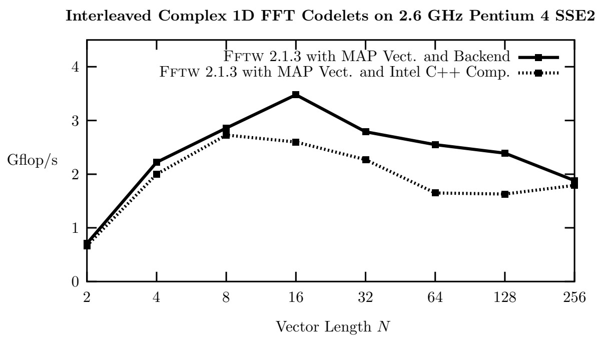
(*e*)



(*f*)



(*g*)



(*h*)

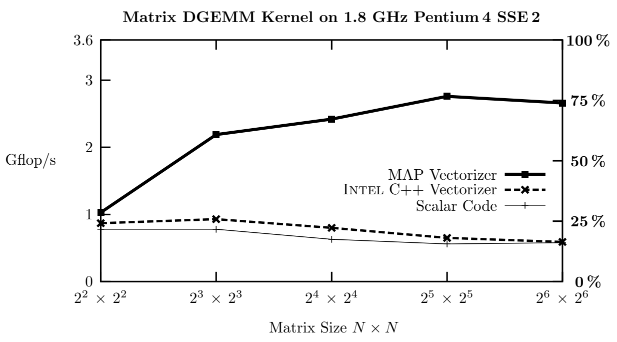


Fig. 9. Floating-point performance of vectorized DSP implementations and linear algebra kernels compared to scalar implementations and third-party SIMD implementations across various machines. (*a*)–(*d*) illustrate the effect of symbolic vectorization, (*e*)–(*g*) assess the Vienna MAP vectorizer and the Vienna MAP backend, and (*h*) shows the performance of MAP vectorized Blas kernels compared to scalar code and code vectorized by the Intel C compiler.

Special thanks go to Markus Pueschel and the SPIRAL team for years of fruitful cooperation. Important parts of the formal vectorization techniques were developed in cooperation with Markus Pueschel.

We kindly appreciate the help we received from Manish Gupta, José Moreira, and their group at IBM T. J. Watson Research Center, making it possible to work on the BlueGene/L prototype and leading to a very pleasant cooperation.

The Center for Applied Scientific Computing at Lawrence Livermore National Laboratory deserves particular appreciation for ongoing support.

Peter Wurzinger and Harald Hofstaetter deserve special appreciation for helpful discussions and for competently carrying out numerical experiments.

We appreciate the very helpful comments and numerous suggestions for improvement made by the anonymous referees.

Additionally, we would like to acknowledge the financial support of the Austrian science fund FWF.

REFERENCES

[1] D. Aberdeen and J. Baxter, "EMMERALD: a fast matrix-matrix multiply using Intel's SSE instructions," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 2, pp. 103–119, 2001.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] G. Almasi et al., "An overview of the BlueGene/L system software organization," Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, pp. 147–159, Springer-Verlag Berlin Heidelberg, 2003.

[4] *AMD Core Math Library (ACML) Manual*, Advanced Micro Devices Corporation, 2000.

[5] ANSI, "ISO/IEC 9899:1999(E), Programming Languages – C," American National Standard Institute (ANSI), New York, 1999.

[6] Apple Computer Inc., "vDSP library," 2001. [Online]. Available: `http://developer.apple.com/tml`

[7] D. F. Bacon and S. L. Graham and O. J. Sharp, *Compiler transformations for high-performance computing*, ACM Comput. Surv., vol. 26, pp. 345–420, 1994.

[8] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, vol. 5, no. 2, 1966.

[9] J. Bilmes, K. Asanovic, C. W. Chin, J. Demmel, *Optimizing matrix multiply using* PHIPAC*: a portable, high-performance, ANSI C coding methodology*, Proceedings of the International Conference on Supercomputing, ACM, Vienna, Austria, pp. 340–347, 1997.

[10] Codeplay Corporation, "VECTOR C," 2002. [Online]. Available: `http://www.codeplay.com`

[11] D. Chaver, C. Tenllado, L. Pinjuel, M. Prieto and F. Tirado, *Wavelet transform for large scale image processing on modern microprocessors*, in J.M.L.M. Palma et al. (Eds.): VECPAR 2002, LNCS 2565, pp. 549–562, Springer-Verlag Berlin Heidelberg, 2003.

[12] R. Crandall and J. Klivington, "Supercomputer-style FFT library for the Apple G4," Advanced Computation Group, Apple Computer Inc., 2002.

[13] J. Demmel, J. Dongarra, V. Eijkhout, and K. Yelick, "Automatic performance tuning for large scale scientifi c applications." *IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2005, *this issue*.

[14] R. J. Fisher and H. G. Dietz, "The SCC Compiler: SWARing at MMX and 3DNow," in *12th Annual Workshop on Languages and Compilers for Parallel Computing*, L. Carter and J. Ferrante (Eds.): LCPC'99, LNCS 1863, pp. 399–414, Springer-Verlag Berlin Heidelberg, 2000.

[15] ——, "Compiling for SIMD within a register," in *11th Annual Workshop on Languages and Compilers for Parallel Computing*, S. Chatterjee et al. (Eds.): LCPC'98, LNCS 1656, pp. 290–304, Springer-Verlag Berlin Heidelberg, 1999.

[16] F. Franchetti, "A portable short vector version of FFTW," in *Proc. Fourth IMACS Symposium on Mathematical Modelling (MATHMOD 2003)*, vol. 2, pp. 1539–1548, 2003.

[17] ——, "Performance portable short vector transforms," Ph. D. Thesis, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2003.

[18] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber, "Architecture independent short vector FFTs," in *Proc. ICASSP*, vol. 2, pp. 1109–1112, 2001.

[19] F. Franchetti and M. Püschel, "A SIMD vectorizing compiler for digital signal processing algorithms," in *Proc. IPDPS*, pp. 20–26, 2002.

[20] ——, "Short vector code generation and adaptation for DSP algorithms." in *Proceedings of the International Conerence on Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP'03)*, vol. 2, pp. 537–540, 2003.

[21] ——, "Short vector code generation for the discrete Fourier transform." in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, pp. 22–26, 2003.

[22] M. Frigo, "A fast Fourier transform compiler," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. New York: ACM Press, pp. 169–180, 1999.

[23] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *ICASSP 98*, vol. 3, pp. 1381–1384, 1998, `http://www.fftw.org`

[24] ——, "The design and implementation of FFTW," *IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2005, *this issue*.

[25] J. Guo, M. Garzarán, and D. Padua, "The power of Belady's algorithm in register allocation for long basic blocks," *Proceedings of the LCPC*, 2003.

[26] Intel Corporation, "AP-808 split radix fast Fourier transform using streaming SIMD extensions," 1999.

[27] ——, "Intel C/C++ compiler user's guide," 2002.

[28] ——, "Math kernel library," 2002. [Online]. Available: `http://www.intel.com/software/products/mkl`

[29] 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*, vol. 9, pp. 449–500, 1990.

[30] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," Digital Western Research Laboratory Palo Alto, California, WRL Research Report 7, 1989.

[31] S. Kral, "The FFTW-GEL web page," 2003. [Online]. Available: `http://www.complang.tuwien.ac.at/skral/fftwgel`

[32] S. Kral, F. Franchetti, J. Lorenz, and C. Ueberhuber, "SIMD vectorization of straight line FFT code," Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, pp. 251–260, 2003.

[33] ——, "FFT compiler techniques," Proceedings of the 13th International Conference on Compiler Construction LNCS 2790, pp. 217–231, 2004.

[34] S. Lamson, "SCIPORT," 1995. [Online]. Available: `http://www.netlib.org/scilib/`

[35] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 145–156, 2000.

[36] R. Leupers and S. Bashford, "Graph-based code selection techniques for embedded processors," *ACM Transactions on Design Automation of Electronic Systems.*, vol. 5, no. 4, pp. 794–814, 2000.

[37] J. Lorenz, "Automatic SIMD vectorization," Ph. D. Thesis, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2004.

[38] M. Lorenz, L. Wehmeyer, and T. Draeger, "Energy aware compilation for DSPs with SIMD instructions," *Proceedings of the 2002 Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02).*, pp. 94–101, 2002.

[39] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL:Code Generation for DSP Transforms." *IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2005, *this issue*.

[40] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[41] K. Nadehara, T. Miyazaki, and I. Kuroda, "Radix-4 FFT implementation using SIMD multi-media instructions," in *Proc. ICASSP 99*, pp. 2131–2135, 1999.

[42] I. Nicholson, "LIBSIMD," 2002. [Online]. Available: `http://libsimd.sourceforge.net`

[43] 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 on High Performance Computing and Applications*, special issue on Automatic Performance Tuning, Vol. 18, pp. 21–45, 2004.

[44] G. Ren, P. Wu and D. A. Padua, "A preliminary study on the vectorization of multimedia applications for multimedia extensions", in *Proc. LCPC 03*, pp. 420–435, 2003.
[45] P. Rodriguez, "A radix-2 FFT algorithm for modern single instruction multiple data (SIMD) architectures", in *Proc. ICASSP 02*, 2002.
[46] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363–400, 2000.
[47] Y. Srikant and P. Shankar, *The Compiler Design Handbook*. Boca Raton London New York Washington D.C.: CRC Press LLC, 2003.
[48] P. N. Swarztrauber, "FFT algorithms for vector computers," *Parallel Comput.*, vol. 1, pp. 45–63, 1984.
[49] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, ser. Frontiers in Applied Mathematics. Philadelphia: Society for Industrial and Applied Mathematics, 1992, vol. 10.
[50] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, CD-ROM Proceedings, 1999.
[51] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Comput.*, vol. 27, pp. 3–35, 2001.
[52] R. C. Whaley, "User contribution to ATLAS," [Online]. Available: `http://www.cs.utk.edu/~rwhaley/papers/atlas_contrib.ps`
[53] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proceedings of the Conference on Programming Languages Design and Implementation* (PLDI), pp. 298–308, 2001.
[54] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. New York: ACM Press, 1991.

Franz Franchetti received the Dipl.-Ing. degree and the PhD degree in technical mathematics from the Vienna University of Technology in 2000 and 2003, respectively. Dr. Franchetti has been with the Vienna University of Technology since 1997. He is currently a research associate with the Dept. of Electrical and Computer Engineering at Carnegie Mellon University. His research interests concentrate on the development of high performance DSP algorithms.

Stefan Kral received the Dipl.-Ing. degree in computer science from the Vienna University of Technology in 2004. He is currently a research associate at the Vienna University of Technology. His research interests include logic programming and compiler backends.

Juergen Lorenz received the Dipl.-Ing. degree and the PhD degree in computer science from the Vienna University of Technology in 2002 and 2004, respectively. He is currently a research associate at the Vienna University of Technology. His research interests include parallel programming and special purpose compilers.

Christoph W. Ueberhuber received the Dipl.-Ing. degree and the PhD degree in technical mathematics from the Vienna University of Technology in 1973 and 1976, respectively, and the *venia docendi habilitation* for numerical mathematics in 1979. Dr. Ueberhuber has been with the Vienna University of Technology since 1973 and is currently a professor of numerical mathematics. His research interests include numerical analysis, high performance numerical computing, and advanced scientific computing. He has published 15 books and more than 100 publications in journals, books, and conference proceedings.