SPIRAL: Code Generation for DSP Transforms

MARKUS PÜSCHEL, MEMBER, IEEE, JOSÉ M. F. MOURA, FELLOW, IEEE,
JEREMY R. JOHNSON, MEMBER, IEEE, DAVID PADUA, FELLOW, IEEE,
MANUELA M. VELOSO, BRYAN W. SINGER, JIANXIN XIONG, FRANZ FRANCHETTI,
ACA GAČIĆ, STUDENT MEMBER, IEEE, YEVGEN VORONENKO, KANG CHEN,
ROBERT W. JOHNSON, AND NICHOLAS RIZZOLO

Invited Paper

Fast changing, increasingly complex, and diverse computing platforms pose central problems in scientific computing: How to achieve, with reasonable effort, portable optimal performance? We present SPIRAL, which considers this problem for the performance-critical domain of linear digital signal processing (DSP) transforms. For a specified transform, SPIRAL automatically generates high-performance code that is tuned to the given platform. SPIRAL formulates the tuning as an optimization problem and exploits the domain-specific mathematical structure of transform algorithms to implement a feedback-driven optimizer. Similar to a human expert, for a specified transform, SPIRAL "intelligently" generates and explores algorithmic and implementation choices to find the best match to the computer's microarchitecture. The "intelligence" is provided by search and learning techniques that exploit the structure of the algorithm and implementation space to guide the exploration and optimization. SPIRAL generates high-performance code for a broad set of DSP transforms, including the discrete Fourier transform, other trigonometric transforms, filter transforms, and discrete wavelet transforms. Experimental results show that the code generated by SPIRAL competes with, and sometimes outperforms, the best available human tuned transform library code.

Keywords*—Adaptation, automatic performance tuning, code optimization, discrete cosine transform (DCT), discrete Fourier transform (DFT), fast Fourier transform (FFT), filter, genetic and evolutionary algorithm, high-performance computing, learning, library generation, linear signal transform, Markov decision process, search, wavelet.*

Manuscript received April 29, 2004; revised October 15, 2004. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant DABT63-98-1-0004 administered by the Army Directorate of Contracting and in part by the National Science Foundation under Awards ACR-0234293, ITR/NGS-0325687, and SYS-310941.

M. Püschel, J. M. F. Moura, F. Franchetti, A. Gačić, and Y. Voronenko are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213-3890 USA (e-mail: pueschel@ece.cmu.edu; moura@ece.cmu.edu; franzf@ece.cmu.edu; agacic@ece.cmu.edu; yvoronen@ece.cmu.edu).

J. R. Johnson is with the Department of Computer Science, Drexel University, Philadelphia, PA 19104-2875 USA (e-mail: jjohnson@cs.drexel.edu).

D. Padua is with the 3318 Digital Computer Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA (e-mail: padua@uiuc.edu).

M. M. Veloso is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890 USA (e-mail: veloso@cs.cmu.edu).

B. W. Singer is at 716 Quiet Pond Ct., Odenton, MD 21113 USA (e-mail: bsinger@cs.cmu.edu).

J. Xiong is with the 3315 Digital Computer Laboratory, Urbana, IL 61801 USA (e-mail: jxiong@cs.uiuc.edu).

K. Chen is with STMicroelectronics, Inc., Malvern, PA 19355 USA (e-mail: chenka88@hotmail.com).

R. W. Johnson is at 3324 21st Ave. South, St. Cloud, MN 56301 USA.

N. Rizzolo is with the Siebel Center for Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA (e-mail: rizzolo@cs.uiuc.edu)

Digital Object Identifier 10.1109/JPROC.2004.840306

I. INTRODUCTION

At the heart of the computer revolution is Moore's law, which has accurately predicted, for more than three decades, that the number of transistors per chip doubles roughly every 18 months. The consequences are dramatic. The current generation of off-the-shelf single processor workstation computers has a theoretical peak performance of more than 10 Gflops,[1] rivaling the most advanced supercomputers from only a decade ago. Unfortunately, at the same time, it is increasingly harder to harness this peak performance, except for the most simple computational tasks. To understand this problem, one has to realize that modern computers are not just faster counterparts of their ancestors but vastly more complex and, thus, with different characteristics. For example, about 15 years ago, the performance for most numerical kernels was determined by the number of operations they require; nowadays, in contrast, a cache miss may be 10–100 times more expensive than a multiplication. More generally, the performance of numerical code now depends crucially on the use of the platform's

[1]Gigaflops (Gflops) = 10^9 floating-point operations per second.

memory hierarchy, register sets, available special instruction sets (in particular vector instructions), and other, often undocumented, microarchitectural features. The problem is aggravated by the fact that these features differ from platform to platform, which makes optimal code platform dependent. As a consequence, the performance gap between a "reasonable" implementation and the best possible implementation is increasing. For example, for the discrete Fourier transform (DFT) on a Pentium 4, there is a gap in runtime performance of one order of magnitude between the code of Numerical Recipes or the GNU scientific library and the Intel vendor library Intel Performance Primitives (IPP) (see Section VII). The latter is most likely handwritten and hand-tuned assembly code, an approach still employed if highest performance is required—a reminder of the days before the invention of the first compiler 50 years ago. However, keeping handwritten code current requires reimplementation and retuning whenever new platforms are released—a major undertaking that is not economically viable in the long run.

In concept, compilers are an ideal solution to performance tuning, since the source code does not need to be rewritten. However, high-performance library routines are carefully hand-tuned, frequently directly in assembly, because today's compilers often generate inefficient code even for simple problems. For example, the code generated by compilers for dense matrix–matrix multiplication is several times slower than the best handwritten code [1] despite the fact that the memory access pattern of dense matrix–matrix multiplication is regular and can be accurately analyzed by a compiler. There are two main reasons for this situation.

The first reason is the lack of reliable program optimization techniques, a problem exacerbated by the increasing complexity of machines. In fact, although compilers can usually transform code segments in many different ways, there is no methodology that guarantees successful optimization. Empirical search [2], which measures or estimates the execution time of several versions of a code segment and selects the fastest, is a simple and general method that is guaranteed to succeed. However, although empirical search has proven extraordinarily successful for library generators, compilers can make only limited use of it. The best known example of the actual use of empirical search by commercial compilers is the decision of how many times loops should be unrolled. This is accomplished by first unrolling the loop and then estimating the execution time in each case. Although empirical search is adequate in this case, compilers do not use empirical search to guide the overall optimization process because the number of versions of a program can become astronomically large, even when only a few transformations are considered.

The second reason why compilers do not perform better is that often important performance improvements can only be attained by transformations that are beyond the capability of today's compilers or that rely on algorithm information that is difficult to extract from a high-level language. Although much can be accomplished with program transformation techniques [3]–[8] and with algorithm recognition [9], [10], starting the transformation process from a high-level language version does not always lead to the desired results. This limitation of compilers can be overcome by library generators that make use of domain-specific, algorithmic information. An important example of the use of empirical search is ATLAS, a linear algebra library generator [11], [12]. The idea behind ATLAS is to generate platform-optimized Basic Linear Algebra Subroutines (BLAS) by searching over different blocking strategies, operation schedules, and degrees of unrolling. ATLAS relies on the fact that LAPACK [13], a linear algebra library, is implemented on top of the BLAS routines, which enables porting by regenerating BLAS kernels. A model-based and, thus, deterministic version of ATLAS is presented in [14]. The specific problem of sparse matrix vector multiplications is addressed in SPARSITY [12], [15], again by applying empirical search to determine the best blocking strategy for a given sparse matrix. References [16] and [17] provide a program generator for parallel programs of tensor contractions, which arise in electronic structure modeling. The tensor contraction algorithm is described in a high-level mathematical language, which is first optimized and then compiled into code.

In the signal processing domain, FFTW [18]–[20] uses a slightly different approach to automatically tune the implementation code for the DFT. For small DFT sizes, FFTW uses a library of automatically generated source code. This code is optimized to perform well with most current compilers and platforms, but is not tuned to any particular platform. For large DFT sizes, the library has a built-in degree of freedom in choosing the recursive computation and uses search to tune the code to the computing platform's memory hierarchy. A similar approach is taken in the UHFFT library [21] and in [22]. The idea of platform adaptive loop body interleaving is introduced in [23] as an extension to FFTW and as an example of a general adaptation idea for divide-and-conquer algorithms [24]. Another variant of computing the DFT studies adaptation through runtime permutations versus readdressing [25], [26]. Adaptive libraries for the related Walsh–Hadamard transform (WHT), based on similar ideas, have been developed in [27]. Reference [28] proposes an object-oriented library standard for parallel signal processing to facilitate porting of both signal processing applications and their performance across parallel platforms.

SPIRAL. In this paper, we present SPIRAL,[2] our research on automatic code generation, code optimization, and platform adaptation. We consider a restricted, but important, domain of numerical problems: namely, digital signal processing (DSP) algorithms, or more specifically, linear signal transforms. SPIRAL addresses the general problem: *How do we enable machines to automatically produce high-quality code for a given platform?* In other words, how can the processes that human experts use to produce highly optimized

[2]The acronym SPIRAL means Signal Processing Implementation Research for Adaptable Libraries. As a tribute to Lou Auslander, the SPIRAL team decided early on that SPIRAL should likewise stand for (in reverse) Lou Auslander's Remarkable Ideas for Processing Signals.

code be automated and possibly improved through the use of automated tools?

Our solution formulates the problem of automatically generating optimal code as an optimization problem over the space of alternative algorithms and implementations of the same transform. To solve this optimization problem using an automated system, we exploit the mathematical structure of the algorithm domain. Specifically, SPIRAL uses a formal framework to efficiently generate many alternative algorithms for a given transform and to translate them into code. Then, SPIRAL uses search and learning techniques to traverse the set of these alternative implementations for the same given transform to find the one that is best tuned to the desired platform while visiting only a small number of alternatives.

We believe that SPIRAL is unique in a variety of respects.

1) SPIRAL is applicable to the entire domain of linear DSP algorithms, and this domain encompasses a large class of mathematically complex algorithms.
2) SPIRAL encapsulates the mathematical algorithmic knowledge of this domain in a concise declarative framework suitable for computer representation, exploration, and optimization—this algorithmic knowledge is far less bound to become obsolete as time goes on than coding knowledge such as compiler optimizations.
3) SPIRAL can be expanded in several directions to include new transforms, new optimization techniques, different target performance metrics, and a wide variety of implementation platforms including embedded processors and hardware generation.
4) We believe that SPIRAL is first in demonstrating the power of machine learning techniques in automatic algorithm selection and optimization.
5) Finally, SPIRAL shows that, even for mathematically complex algorithms, machine generated code can be as good as, or sometimes even better than, any available expert handwritten code.

Organization of this paper. The paper begins, in Section II, with an explanation of our approach to code generation and optimization and an overview of the high-level architecture of SPIRAL. Section III explains the theoretical core of SPIRAL that enables optimization in code design for a large class of DSP transforms: a mathematical framework to structure the algorithm domain and the signal processing language (SPL) to make possible efficient algorithm representation, generation, and manipulation. The mapping of algorithms into efficient code is the subject of Section IV. Section V describes the evaluation of the code generated by SPIRAL—by adapting the performance metric, SPIRAL can solve various code optimization problems. The search and learning strategies that guide the automatic feedback-loop optimization in SPIRAL are considered in Section VI. We benchmark the quality of SPIRAL's automatically generated code in Section VII, showing a variety of experimental results. Section VIII discusses current limitations of SPIRAL and ongoing and future work. Finally, we offer conclusions in Section IX.

II. SPIRAL: Optimization Approach to Tuning Implementations to Platforms

In this section, we provide a high-level overview of the SPIRAL code generation and optimization system. First, we explain the high-level approach taken by SPIRAL, which restates the problem of finding fast code as an optimization problem over the space of possible alternatives. Second, we explain the architecture of SPIRAL, which implements a flexible solver for this optimization problem and which resembles the human approach for code creation and optimization. Finally, we discuss how SPIRAL's architecture is general enough to solve a large number of different implementation/optimization problems for the DSP transform domain. More details are provided in later sections.

A. Optimization: Problem Statement

We restate the problem of automatically generating software (SW) implementations for linear DSP transforms that are tuned to a target hardware (HW) platform as the following optimization problem. Let $\mathbf{P}$ be a target platform, $\mathbf{T}_n$ a DSP transform parameterized at least by its size n, $\mathbf{I} \in \mathcal{I}$ a SW implementation of $\mathbf{T}_n$, where $\mathcal{I}$ is the set of SW implementations for the platform $\mathbf{P}$ and transform $\mathbf{T}_n$, and $\mathbf{C}(\mathbf{T}_n, \mathbf{P}, \mathbf{I})$ the cost of the implementation $\mathbf{I}$ of the transform $\mathbf{T}_n$ on the platform $\mathbf{P}$.

The implementation $\widehat{\mathbf{I}}$ of $\mathbf{T}_n$ that is tuned to the platform $\mathbf{P}$ with respect to the performance cost $\mathbf{C}$ is

$$\widehat{\mathbf{I}} = \widehat{\mathbf{I}}(\mathbf{P}) = \arg \min_{\mathbf{I} \in \mathcal{I}(\mathbf{P})} \mathbf{C}(\mathbf{T}_n, \mathbf{P}, \mathbf{I}). \tag{1}$$

For example, we can have the following: as target platform $\mathbf{P}$ a particular Intel Pentium 4 workstation; as transform $\mathbf{T}_n$ the DFT of size $n = 1024$, which we will refer to as $\mathbf{DFT}_{1024}$, or the discrete cosine transform (DCT) of type 2 and size 32 $\mathbf{DCT\text{-}2}_{32}$; as SW implementation $\mathbf{I}$ a C-program for computing $\mathbf{T}_n$; and as cost measure $\mathbf{C}$ the runtime of $\mathbf{I}$ on $\mathbf{P}$. In this case, the cost depends on the chosen compiler and flags; thus, this information has to be included in $\mathbf{P}$. Note that with the proliferation of special vendor instruction sets, such as vector instructions that exceed the standard C programming language, the set of all implementations becomes in general platform dependent, i.e., $\mathcal{I} = \mathcal{I}(\mathbf{P})$ with elements $\mathbf{I} = \mathbf{I}(\mathbf{P})$.

To carry out the optimization in (1) and to automatically generate the tuned SW implementation $\widehat{\mathbf{I}}$ poses several challenges.

- **Set of implementations** $\mathcal{I}$. How to characterize and generate the set $\mathcal{I}$ of SW implementations $\mathbf{I}$ of $\mathbf{T}_n$?
- **Minimization of** $\mathbf{C}$. How to automatically minimize the cost $\mathbf{C}$ in (1)?

In principle, the set of implementations $\mathcal{I}$ for $\mathbf{T}_n$ should be unconstrained, i.e., include *all* possible implementations. Since this is unrealistic, we aim at a broad enough set of implementations. We solve both challenges of characterizing $\mathcal{I}$

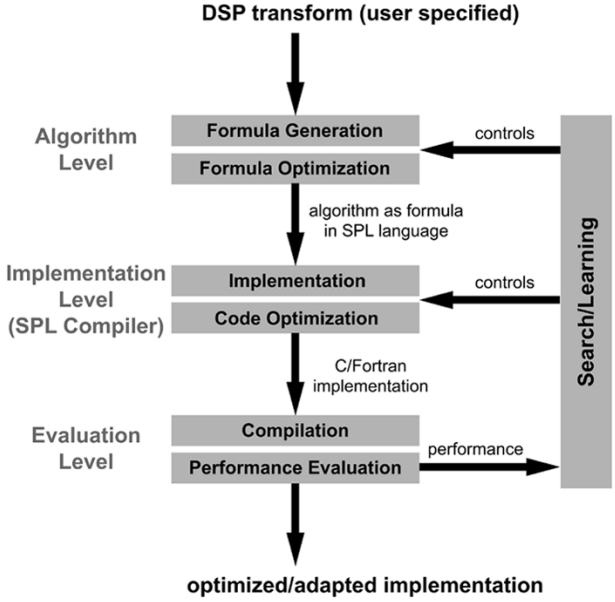


Fig. 1. The architecture of SPIRAL.

and minimizing $\mathbf{C}$ by recognizing and exploiting the specific structure of the domain of linear DSP transforms. This structure enables us to represent algorithms for $\mathbf{T}_n$ as *formulas* in a concise mathematical language called "signal processing language" (SPL), which utilizes only a few constructs. Further, it is possible to generate these SPL formulas (or algorithms) *recursively* using a small set of *rules* to obtain a large formula space $\mathcal{F}$. These formulas, in turn, can be translated into code. The SPIRAL system implements this framework and we define $\mathcal{I}$ as the set of implementations that SPIRAL can generate. The degrees of freedom in translating from $\mathcal{F}$ to $\mathcal{I}$ reflect the implementation choices that SPIRAL can consider for the given algorithms. Finally, the recursive structure of $\mathcal{F}$ and, thus, $\mathcal{I}$ enables the use of various, transform independent, search and learning techniques that successfully produce very good solutions for (1), while generating only a small subset of $\mathcal{I}$.

SPIRAL's architecture, shown in Fig. 1, is a consequence of these observations and, for the class of DSP transforms included in SPIRAL, can be viewed as a solver for the optimization problem (1). To benchmark the performance of the transform implementations generated by SPIRAL, we compare them against the best available implementations *whenever possible*. For example, for the DFT, we benchmark SPIRAL against the DFT codes provided by FFTW, [18], [19], and against vendor libraries like Intel's IPP and Math Kernel Library (MKL); the latter are coded by human experts. However, because of SPIRAL's breadth, there are no readily available high-quality implementations for many of SPIRAL's transforms. In these cases, we explore different alternatives generated by SPIRAL itself.

In the following paragraphs, we briefly address the above two challenges of generating the set of implementations $\mathcal{I}$ and of minimizing $\mathbf{C}$. The discussion proceeds with reference to Fig. 1, which shows the architecture of SPIRAL as a block diagram.

B. Set of Implementations $\mathcal{I}$

To characterize the set of implementations $\mathcal{I}$, we first outline the two basic steps that SPIRAL takes to go from the high-level specification of the transform $\mathbf{T}_n$ to an actual implementation $\mathbf{I} \in \mathcal{I}$ of $\mathbf{T}_n$. The two steps correspond to the "Algorithm Level" and the "Implementation Level" in Fig. 1. The first derives an algorithm for the given transform $\mathbf{T}_n$, represented as a *formula* $\mathbf{F} \in \mathcal{F}$ where $\mathcal{F}$ is the formula or algorithm space for $\mathbf{T}_n$. The second translates the formula $\mathbf{F}$ into a program $\mathbf{I} \in \mathcal{I}$ in a high-level programming language such as Fortran or C, which is then compiled by an existing commercial compiler.

Algorithm level. In SPIRAL, an algorithm for a transform $\mathbf{T}_n$ is generated recursively using *breakdown rules* and *manipulation rules*. Breakdown rules are recursions for transforms, i.e., they specify how to compute a transform from other transforms of the same or a different type and of the same or a smaller size. The "Formula Generation" block in Fig. 1 uses a database of breakdown rules to recursively expand a transform $\mathbf{T}_n$, until no further expansion is possible to obtain a completely expanded formula $\mathbf{F} \in \mathcal{F}$. This formula specifies one algorithm for $\mathbf{T}_n$. The "Formula Optimization" block then applies manipulation rules to translate the formula into a different formula that may better exploit the computing platform's HW characteristics. These optimizations at the mathematical level can be used to overcome inherent shortcomings of compiler optimizations, which are performed at the code level where much of the structural information is lost.

SPIRAL expresses rules and formulas in a special language—the *SPL*, which is introduced and explained in detail in Section III; here, we only provide a brief glimpse. SPL uses a small set of constructs including symbols and matrix operators. Symbols are, for example, certain patterned matrices like the identity matrix I_m of size m. Operators are matrix operations such as matrix multiplication or the tensor product $\otimes$ of matrices. For example, the following is a breakdown rule for the transform $\mathbf{DCT\text{-}2}_n$ written in SPL:

$$\begin{aligned}\mathbf{DCT\text{-}2}_n \rightarrow L^n_m(\mathbf{DCT\text{-}2}_m \oplus \mathbf{DCT\text{-}4}_m)\\ \cdot (F_2 \otimes I_m)(I_m \oplus J_m)\end{aligned} \tag{2}$$

where $n = 2m$. This rule expands the $\mathbf{DCT\text{-}2}$ of size $n = 2m$ into transforms $\mathbf{DCT\text{-}2}$ and $\mathbf{DCT\text{-}4}$ of half the size m, and additional operations (the part that is not boldfaced).

An example of a manipulation rule expressed in SPL is

$$I_n \otimes A_m \rightarrow L^{mn}_n (A_m \otimes I_n) L^{mn}_m.$$

We will see later that the left-hand side $I_n \otimes A_m$ is a parallelizable construct, while the right-hand side $A_m \otimes I_n$ is a vectorizable construct.

Implementation level. The output of the "Algorithm Level" block is an SPL formula $\mathbf{F} \in \mathcal{F}$, which is fed into the second level in Fig. 1, the "Implementation Level," also called the SPL Compiler.

The SPL Compiler is divided into two blocks: the Implementation and Code Optimization blocks. The Implementa-

tion block translates the SPL formula into C or Fortran code using a particular set of implementation options, such as the degree of unrolling. Next, the Code Optimization block performs various standard and less standard optimizations at the C (or Fortran) code level, e.g., common subexpression elimination (CSE) and code reordering for locality. These optimizations are necessary, as standard compilers are often not efficient when used for automatically generated code, in particular, for large blocks of straightline code (i.e., code without loops and control structures).

Both blocks, Algorithm Level and Implementation Level, are used to generate the elements of the implementation space $\mathcal{I}$. We now address the second challenge, the optimization in (1).

C. Minimization of $\mathbf{C}$

Solving the minimization (1) requires SPIRAL to evaluate the cost $\mathbf{C}$ for a given implementation $\mathbf{I}$ and to autonomously explore the implementation space $\mathcal{I}$. Cost evaluation is accomplished by the third level in SPIRAL, the Evaluation Level block in Fig. 1. The computed value $\mathbf{C}(\mathbf{T}_n, \mathbf{P}, \mathbf{I})$ is then input to the Search/Learning block in the feedback loop in Fig. 1, which performs the optimization.

Evaluation level. The Evaluation Level is decomposed into two blocks: the Compilation and Performance Evaluation. The Compilation block uses a standard compiler to produce an executable and the Performance Evaluation block evaluates the performance of this executable, for example, the actual runtime of the implementation $\mathbf{I}$ on the given platform $\mathbf{P}$. By keeping the evaluation separated from implementation and optimization, the cost measure $\mathbf{C}$ can easily be changed to make SPIRAL solve various implementation optimization problems (see Section II-E).

Search/Learning. We now consider the need for intelligent navigation in the implementation space $\mathcal{I}$ to minimize (or approximate the minimization of) $\mathbf{C}$. Clearly, at both the Algorithm Level and the Implementation Level, there are choices to be made. At each stage of the Formula Generation, there is freedom regarding which rule to apply. Different choices of rules lead to different formulas (or algorithms) $\mathbf{F} \in \mathcal{F}$. Similarly, the translation of the formula $\mathbf{F}$ to an actual program $\mathbf{I} \in \mathcal{I}$ implies additional choices, e.g., the degree of loop unrolling or code reordering. Since the number of these choices is finite, the sets of alternatives $\mathcal{F}$ and $\mathcal{I}$ are also finite. Hence, an exhaustive enumeration of all implementations $\mathbf{I} \in \mathcal{I}$ would lead to the optimal implementation $\widehat{\mathbf{I}}$. However, this is not feasible, even for small transform sizes, since the number of available algorithms and implementations usually grows exponentially with the transform size. For example, the current version of SPIRAL reports that the size of the set of implementations $\mathcal{I}$ for the $\mathbf{DCT\text{-}2}_{64}$ exceeds $1.47 \cdot 10^{19}$. This motivates the feedback loop in Fig. 1, which provides an efficient alternative to exhaustive search and an engine to determine an approximate solution to the minimization problem in (1).

The three main blocks on the left in Fig. 1, and their underlying framework, provide the machinery to enumerate, for the same transform, different formulas and different implementations. We solve the optimization problem in (1) through an empirical exploration of the space of alternatives. This is the task of the Search/Learning block, which, in a feedback loop, drives the algorithm generation and controls the choice of algorithmic and coding implementation options. SPIRAL uses search methods such as dynamic programming and evolutionary search (see Section VI-A). An alternative approach, also available in SPIRAL, uses techniques from artificial intelligence to *learn* which choice of algorithm is best. The learning is accomplished by reformulating the optimization problem (1) in terms of a Markov decision process and reinforcement learning. Once learning is completed, the degrees of freedom in the implementation are fixed. The implementation is *designed* with no need for additional search (see Section VI-B).

An important question arises: Why is there is a need to explore the formula space $\mathcal{F}$ at all? Traditionally, the analysis of algorithmic cost focuses on the number of arithmetic operations of an algorithm. Algorithms with a similar number of additions and multiplications are considered to have similar cost. The rules in SPIRAL lead to "fast" algorithms, i.e., the formulas $\mathbf{F} \in \mathcal{F}$ that SPIRAL explores are essentially equal in terms of the operation count. By "essentially equal" we mean that for a transform of size n, which typically has a complexity of $\Theta(n \log(n))$, the costs of the formulas differ only by $O(n)$ operations and are often even equal. So the formulas' differences in performance are in general not a result of different arithmetic costs, but are due to differences in locality, block sizes, and data access patterns. Since computers have an hierarchical memory architecture, from registers—the fastest level—to different types of caches and memory, different formulas will exhibit very different access times. These differences cause significant disparities in performance across the formulas in $\mathcal{F}$. The Search/Learning block searches for or learns those formulas that best match the target platforms memory architecture and other microarchitectural features.

D. General Comments

The following main points about SPIRAL's architecture are worth noting.

- SPIRAL is autonomous, optimizing at both the algorithmic level and the implementation level. SPIRAL incorporates domain specific expertise through both its mathematical framework for describing and generating algorithms and implementations and through its effective algorithm and implementation selection through the Search/Learning block.
- The SPL language is a key element in SPIRAL: SPL expresses recursions and formulas in a mathematical form accessible to the transform expert, while retaining all the structural information that is needed to generate efficient code. Thus, SPL provides the link between the "high" mathematical level of transform algorithms and the "low" level of their code implementations.

- SPIRAL's architecture is modular: it clearly separates algorithmic and implementation issues. In particular, the code optimization is decomposed as follows.
  a) *Deterministic optimizations* are always performed without the need for runtime information. These optimization are further divided into algorithm level optimizations (Formula Optimization block) such as formula manipulations for vector code, and into implementation level optimizations (Code Optimization block) such as CSE.
  b) *Nondeterministic optimizations* arise from choices whose effect cannot easily be statically determined. The generation and selection of these choices is driven by the Search/Learning block. These optimizations are also divided into algorithmic choices and implementation choices.

  Because of its modularity, SPIRAL can be extended in different directions without the need for understanding all domains involved.
- SPIRAL abstracts into its high-level mathematical framework many common optimizations that are usually performed at the low-level compilation step. For example, as we will explain in Section IV-E, when platform specific vector instructions are available, they can be matched to certain patterns in the formulas and, using mathematical manipulations, a formula's structure can be improved for mapping into vector code. Rules that favor the occurrence of these patterns in the produced formula are then naturally selected by the search engine in SPIRAL to produce better tuned code.
- SPIRAL makes use of runtime information in the optimization process. In a sense, it could be said that SPIRAL carries out profile-driven optimization although compiler techniques reported in the literature require profiling to be done only once [29], [30]. Compiler writers do not include profiling in a feedback loop to avoid long compilation times, but for the developers of library generators like SPIRAL, the cost of installation is less of a concern since installation must be done only once for each class of machines.
- With slight modifications, SPIRAL can be used to automatically solve various implementation or algorithm optimization problems for the domain of linear DSP transforms; see Section II-E.

Next, we provide several examples to show the breadth of SPIRAL.

E. Applications of SPIRAL

SPIRAL's current main application is the generation of very fast, platform-tuned implementations of linear DSP transforms for desktop or workstation computers. However, SPIRAL's approach is quite versatile and the SPIRAL system can be used for a much larger scope of signal processing implementation problems and platforms.

1) It goes beyond trigonometric transforms such as the DFT and the DCT, to other DSP transforms such as the wavelet transform and DSP kernels like filters.
2) It goes beyond desktop computers and beyond C and Fortran to implementations for multiprocessor machines and to generating code using vendor specific instructions like SSE for the Pentium family, or AltiVec for the Power PC.
3) It goes beyond runtime to other performance metrics including accuracy and operation count.

We briefly expand here on two important examples to illustrate SPIRAL's flexibility. More details are provided later in Sections V and VII.

Special instructions and parallel platforms. Most modern platforms feature special instructions, such as vector instructions, which offer a large potential speedup. Compilers are restricted to code level transformations and cannot take full advantage of these instructions, except for simple numerical algorithms. SPIRAL automatically generates code that uses these special instructions. This is achieved in three steps: 1) by identifying structures in SPL formulas that can be naturally mapped into code using these special instructions; 2) by identifying SPL manipulation rules whose application produces these structures; these rules are included into the Formula Optimization block in Fig. 1; and (3) by extending the Implementation block in Fig. 1 to produce code that uses those special instructions. We provide details for vector instructions in Section IV-E. We also have results demonstrating that the same approach can be used to generate code for SMP platforms (see Section IV-F).

Expanding SPIRAL: new transforms and rules. SPIRAL is easily expanded with new transforms and/or new rules by including them in the rule database of the Formula Generation block. This is achieved without affecting the remaining components of SPIRAL, provided that the new rules can be expressed using the SPL constructs currently available in SPIRAL. If this is not the case, SPL can be extended to include new constructs. Once this is accomplished, the entire functionality of SPIRAL, including the code generation, the Search/Learning block, and the automatic tuning of implementations becomes immediately available to the new transform or rule.

Other applications. There are various other implementation/algorithm optimization problems that can be addressed by the SPIRAL system. Examples include the generation of numerically accurate code, multiplierless implementations, or algorithms with minimal operation counts. We will briefly discuss these extensions in Section V.

In summary, the discussion in this overview outlined how SPIRAL integrates algorithmic knowledge with code mapping and feedback optimization and pointed out the capabilities of the resulting system. The SPIRAL system can be adapted to new platforms, extended with new linear transforms and their algorithms, and extended with new performance measures. Extensions of the system, once completed, apply to the entire collection of DSP transforms and kernels as well as to the full set of problems included in its current

domain rather than just a single transform or a single problem type.

Now we begin the detailed description of SPIRAL.

III. SPIRAL's Mathematical Framework and Formula Generation

This section details SPIRAL's mathematical framework to represent and generate fast algorithms for linear DSP transforms. The framework is declarative in nature, i.e., the knowledge about transforms and algorithms is represented in the form of equations and rules. The framework enables: 1) the automatic generation of transform algorithms; 2) the concise symbolic representation of transform algorithms as *formulas* in the language SPL that we introduce; 3) the structural optimization of algorithms in their formula representation; and 4) the automated mapping into various code types, which is the subject of Section IV.

We divide the framework into the following parts: transforms, the language SPL, breakdown and manipulation rules, and ruletrees and formulas. Finally, we explain how the framework is implemented in the Formula Generation and Formula Optimization blocks in Fig. 1.

A. Transforms

SPIRAL generates fast implementations for linear DSP transforms. Although in the DSP literature transforms are usually presented in the form of summations, we express them equivalently as a matrix–vector multiplication $y = Mx$. In this equation, x and y are, respectively, the input and the output n-dimensional vectors (or signals) that collect the n signal samples, and M is the $n \times n$ transform matrix. Usually, the transform M exists for every input size n. An example is the DFT, which is defined, for input size n, by the $n \times n$ DFT matrix

$$\mathbf{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi i/n}, \quad i = \sqrt{-1}. \tag{3}$$

In SPIRAL, a transform is a parameterized class of matrices. It is represented symbolically by a mnemonic name such as "DFT" and by a list of parameters, such as the size n. By specifying the parameter(s), we obtain an instance of a transform, which we will also refer to as a transform. An example is $\mathbf{DFT}_8$. Transforms are written using boldfaced type. Transform matrices, as well as the input and output vectors, can be real or complex valued.

At the time of this writing, SPIRAL contains 36 transforms (some of which are variants of each other).

Trigonometric transforms. We provide some important examples of DSP transforms for which SPIRAL can generate tuned code. We first consider the class of trigonometric transforms that, besides the DFT in (3), includes the following transforms: all the 16 types of DCTs and discrete sine transforms (DSTs), of which the most commonly used (e.g., in the JPEG and MPEG multimedia standards) are the DCTs of types 2, 3, and 4; the inverse modulated DCT (IMDCT), which is used in MPEG audio compression standards and is a rectangular transform; the real DFT (RDFT) that computes the DFT on a real input data set; the WHT; and the discrete Hartley transform (DHT). Some of these transforms are defined as follows:

$$\mathbf{DCT\text{-}2}_n = \left[\cos\frac{k(2\ell+1)\pi}{2n}\right]_{0\le k,\,\ell<n} \tag{4}$$

$$\mathbf{DCT\text{-}3}_n = \mathbf{DCT\text{-}2}_n^T \text{ (transpose)} \tag{5}$$

$$\mathbf{DCT\text{-}4}_n = \left[\frac{\cos(2k+1)(2\ell+1)\pi}{4n}\right]_{0\le k,\,\ell<n} \tag{6}$$

$$\mathbf{IMDCT}_n = \left[\cos\frac{(2k+1)(2\ell+1+n)\pi}{4n}\right]_{0\le k<2n,\,0\le\ell<n} \tag{7}$$

$$\mathbf{RDFT}_n = [r_{k\ell}]_{0\le k,\,\ell<n},$$
$$r_{k\ell} = \begin{cases} \cos\frac{2\pi k\ell}{n}, & k \le \lfloor\frac{n}{2}\rfloor, \\ -\sin\frac{2\pi k\ell}{n}, & k > \lfloor\frac{n}{2}\rfloor \end{cases} \tag{8}$$

$$\mathbf{WHT}_n = \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix},$$
$$\mathbf{WHT}_2 = \mathbf{DFT}_2 \tag{9}$$

$$\mathbf{DHT} = \left[\cos\frac{2k\ell\pi}{n} + \sin\frac{2k\ell\pi}{n}\right]_{0\le k,\,\ell<n}. \tag{10}$$

Note that the WHT in (9) is defined recursively.

Besides these trigonometric transforms, SPIRAL includes other transforms and DSP kernels. In fact, in SPIRAL, any linear operation on finite discrete sequences, i.e., matrix–vector multiplication, qualifies as a transform. In particular, this includes linear filters and filter banks.

Filters. We recall that a filter in DSP computes the convolution of two sequences: one is the signal being filtered, the input signal; the other is the sequence that characterizes the filter, its impulse response. As important examples, we consider finite-impulse response (FIR) filters and the discrete wavelet transforms (DWTs).

Although we can represent FIR filters and the DWT as matrices, it is more convenient, and more common, to define them iteratively or recursively, as was the case with the WHT above. We start with the basic building block, the FIR filter transform. The filter's output is the convolution of its impulse response and an infinite support input signal. The filter impulse response can be viewed as the column vector $h = [h_l, \ldots, h_0, \ldots, h_{-r}]^T$ of length $l + r + 1$ or as the z-transform polynomial $h[z]$ (e.g., [31])

$$h[z] = \sum_{k=-r} h_k z^{-k}.$$

The output of the FIR filter for n output points is computed by multiplying the relevant (i.e., contributing to these outputs) finite subvector of length $n+l+r$ of x by the FIR transform matrix $\mathbf{Filt}_n(h[z])$ given by

$$\mathbf{Filt}_n(h[z]) = \begin{bmatrix} h_1 & \ldots & h_{-r} & & & \\ & h_1 & \ldots & h_{-r} & & \\ & & \ddots & & \ddots & \\ & & & h_l & \ldots & h_{-r} \end{bmatrix}. \tag{11}$$

In practice, signals are of finite duration n. To account for boundary effects and to enable filtering, i.e., multiplying with

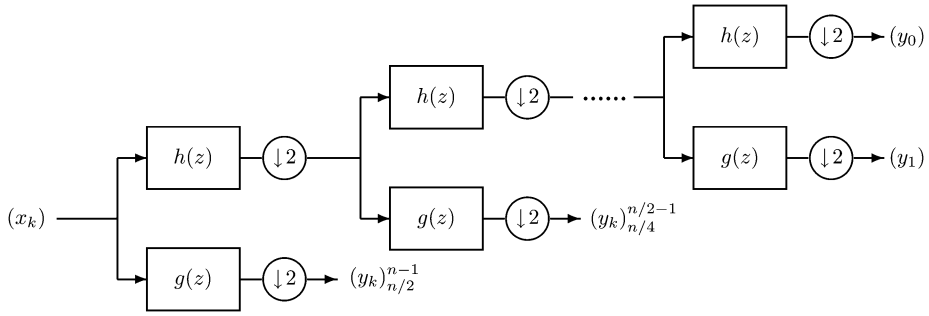


Fig. 2. Filter bank interpretation of the DWT.

(11), these signals are, thus, extended to the left (up) and to the right (below) to have length $n+l+r$. Linear extensions can be also interpreted as matrix–vector multiplications with an $(n+l+r)\times n$ matrix $E_{n,l,r}^{f_l,f_r}$, where f_l and f_r specify the left and the right signal extension type, and l and r are the number of left and right extension points. Examples of extension types include periodic (per), whole-point and half-point symmetric or antisymmetric (*ws/hs/wa/ha*), and zero-padding $(zero)$. For example, in a per extension, the signal is extended by l points to the left and by r points to the right, by assuming that the signal is periodically repeated beyond its fundamental period, which is given by the actual available data. After extending the signal, we can define the *extended* FIR filter transform as the composition of both the FIR filter transform (11) and the extension transform

$$\mathbf{Filt}_n^{f_l,f_r}(h[z]) = \mathbf{Filt}_n(h[z])\cdot E_{n,l,r}^{f_l,f_r} \tag{12}$$

where the parameters l and r are implicitly given by $h[z]$. For example, the matrix $E_{n,l,r}^{f_l,f_r}$ for periodic signal extension, i.e., $f_l = f_r = per$, is

$$E_{n,l,r}^{\mathrm{per,per}} = \left[\begin{array}{c|c} 0 & I_l \\ \hline \multicolumn{2}{c}{I_n} \\ \hline I_r & 0 \end{array}\right]$$

where I_n denotes the $n\times n$ identity matrix.

DWT. Many applications, such as JPEG2000 [32], make use of a two-channel DWT, which is usually defined as the recursive bank of filters and downsamplers shown in Fig. 2.

The filters in the filter bank are linear, and hence is the DWT. In matrix form, the DWT is given by

$$\mathbf{DWT}_n^{f_l,f_r}(h[z],g[z]) = \begin{bmatrix} (\downarrow n)_n \mathbf{Filt}_n^{f_l,f_r}\left(\prod_{k=0}^{n-1} h[z^{2^k}]\right) \\ \left(\downarrow \tfrac{n}{2}\right)_n \mathbf{Filt}_n^{f_l,f_r}\left(g[z^{2^{n-1}}]\prod_{k=0}^{n-2} h[z^{2^k}]\right) \\ \vdots \\ (\downarrow 8)_n \mathbf{Filt}_n^{f_l,f_r}(g[z^4]h[z^2]h[z]) \\ (\downarrow 4)_n \mathbf{Filt}_n^{f_l,f_r}\left(g[z^2]h[z]\right) \\ (\downarrow 2)_n \mathbf{Filt}_n^{f_l,f_r}(g[z]) \end{bmatrix} \tag{13}$$

where $(\downarrow k)_n$ is the $n/k\times n$ matrix that selects every kth element from its input, starting with the first. The matrix form (13) is obtained from Fig. 2 by observing that $\mathbf{Filt}_{n/k}(h[z])\cdot(\downarrow k)_{n+kl+kr} = (\downarrow k)_n \cdot \mathbf{Filt}_n\left(h[z^k]\right)$. (Note that when stacking filters as in (13), the defining polynomials may need to be zero extended to equalize the sizes of the blocks.)

B. SPL

The significance in DSP of the transforms introduced in Section III-A arises from the existence of *fast algorithms* to compute them. The term "fast" refers to the number of operations required to compute the transform: fast algorithms for transforms of size n typically reduce the number of operations from $O(n^2)$ (as required by direct evaluation) to $O(n\log(n))$. Furthermore, these algorithms are highly structured. To exploit the structure of the DSP transforms, SPIRAL represents these algorithms in a specially designed language—SPL—which is described in this section. For example, an important element in SPL is the tensor or Kronecker product, whose importance for describing and manipulating DFT algorithms was already demonstrated in [33], [34]. After introducing SPL, we develop the framework to efficiently generate and manipulate algorithms for DSP transforms in Sections III-C and D.

We start with a motivating example. Consider the DCT of type 2 defined in (4) and given by the following 4×4 matrix, which is then factored into a product of three sparse matrices. We use the notation $c_k = \cos k\pi/8$

$$\begin{aligned} \mathbf{DCT\text{-}2}_4 &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ c_1 & c_3 & c_5 & c_7 \\ c_2 & c_6 & c_6 & c_2 \\ c_3 & c_7 & c_1 & c_5 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ c_2 & c_6 & 0 & 0 \\ 0 & 0 & c_1 & c_3 \\ 0 & 0 & c_3 & c_7 \end{bmatrix} \\ &\quad \times \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}. \end{aligned} \tag{14}$$

The right-hand side of (14) decomposes the matrix $\mathbf{DCT\text{-}2}_4$ into a product of three sparse matrices. This factorization reduces the cost for computing the transform (the matrix–vector product) from 12 additions and 12 multiplications to eight additions and six multiplications. To avoid a possible confusion, we emphasize again that this cost does not refer to multiplying the three sparse matrices together, but to the computation of the matrix–vector product

Table 1
Definition of the Most Important SPL Constructs in BNF;
n, k Are Positive Integers, α, a_i Real Numbers

$$
\begin{array}{rcll}
\langle \text{spl} \rangle & ::= & \langle \text{generic} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{transform} \rangle \mid & \\
 & & \langle \text{spl} \rangle \cdot \cdots \cdot \langle \text{spl} \rangle \mid & \text{(product)} \\
 & & \langle \text{spl} \rangle \oplus \ldots \oplus \langle \text{spl} \rangle \mid & \text{(direct sum)} \\
 & & \langle \text{spl} \rangle \otimes \cdots \otimes \langle \text{spl} \rangle \mid & \text{(tensor product)} \\
 & & \mathrm{I}_n \otimes_k \langle \text{spl} \rangle \mid \mathrm{I}_n \otimes^k \langle \text{spl} \rangle \mid & \text{(overlapped tensor product)} \\
 & & \overline{\langle \text{spl} \rangle} \mid & \text{(conversion to real)} \\
 & & \ldots & \\
\langle \text{generic} \rangle & ::= & \mathrm{diag}(a_0, \ldots, a_{n-1}) \mid \ldots & \\
\langle \text{symbol} \rangle & ::= & \mathrm{I}_n \mid \mathrm{J}_n \mid \mathrm{L}^n_k \mid \mathrm{R}_\alpha \mid \mathrm{F}_2 \mid \ldots & \\
\langle \text{transform} \rangle & ::= & \mathbf{DFT}_n \mid \mathbf{WHT}_n \mid \mathbf{DCT\text{-}2}_n \mid \mathbf{Filt}_n(h[z]) \mid \ldots &
\end{array}
$$

$y = \mathbf{DCT\text{-}2}_4 x$ in three steps (corresponding to the three sparse matrices), and it is in this sense that (14) is considered as an algorithm for $\mathbf{DCT\text{-}2}_4$. Equation (14) shows further that, besides their sparseness, the matrix factors are highly structured. Identifying this structure and then making use of it is a key concept in SPIRAL and provides the motivation for SPL.

SPL is a language suitable to express products of structured sparse matrices using a small set of constructs and symbols. However, as we will see, this set is sufficient to represent a large class of different transform algorithms. Table 1 provides a grammar for SPL in Backus–Naur form (BNF) [35] as the disjoint union of different choices of rules (separated by a vertical line "|") to generate valid SPL expressions. Symbols marked by $\langle \cdot \rangle$ are *nonterminal*, $\langle \text{spl} \rangle$ is the initial nonterminal, and all the other symbols are *terminals*. We call the elements of SPL *formulas*. The meaning of the SPL constructs is explained next.

Generic matrices. SPL provides constructs to represent generic matrices, generic permutation matrices, and generic sparse matrices. Since most matrices occurring within transform algorithms have additional structure, these constructs are rarely used except diagonal matrices. These are written as $\mathrm{diag}(a_0, \ldots, a_{n-1})$, where the argument list contains the diagonal entries of the matrix. Scalars, such as the numbers a_i, can be real or complex and can be represented in a variety of ways. Examples include rational, floating-point, special constants, and *intrinsic* functions, such as 1, 3/2, 1.23, $1.23e\text{-}04$, π, $\mathrm{sqrt}(2)$, and $\sin(3\pi/2)$.

Symbols. Frequently occurring classes of matrices are represented by parameterized *symbols*. Examples include the $n \times n$ *identity matrix* I_n; the matrix J_n obtained from the identity matrix by reversing the columns (or rows); the $n \times n$ *zero matrix* 0_n; the *twiddle matrix*

$$
T^n_k = \mathrm{diag}\left(\omega_n^{0\cdot 0}, \ldots, \omega_n^{(k-1)\cdot 0}, \omega_n^{0\cdot 1}, \ldots, \omega_n^{(k-1)\cdot 1}, \ldots \right.
\left. \omega_n^{0\cdot(n/k-1)}, \ldots, \omega_n^{(k-1)\cdot(n/k-1)}\right)
$$

the *stride permutation* matrix L^n_k, which reads the input at stride k and stores it at stride 1, defined by its corresponding permutation

$$
L^n_k : i\left(\frac{n}{k}\right) + j \mapsto jk + i, \quad 0 \le i < k, \;\; 0 \le j < \frac{n}{k}
$$

the 2×2 *rotation matrix* (with angle α)

$$
R_\alpha = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}
$$

and the *butterfly matrix*, which is equal to the 2×2 DFT matrix, but not considered a transform (i.e., it is terminal)

$$
F_2 = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}.
$$

Transforms. SPL expresses transforms as introduced in Section III-A. Examples include $\mathbf{DFT}_n$, $\mathbf{DCT\text{-}2}_n$, and $\mathbf{Filt}_n(h[z])$. In our framework, transforms are fundamentally different from the symbols introduced above (as emphasized by boldfacing transforms), which will be explained in Sections III-C and D. In particular, only those formulas that do not contain transforms can be translated into code. Both, the set of transforms and the set of symbols available in SPL are user extensible.

Matrix constructs. SPL constructs can be used to form structured matrices from a set of given SPL matrices. Examples include the product of matrices AB (sometimes written as $A \cdot B$), the sum of matrices $A + B$, and the direct sum $\oplus$ and the tensor or Kronecker product $\otimes$ of two matrices A and B, defined, respectively, by

$$
A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}, \text{ and}
$$
$$
A \otimes B = [a_{k,\ell} B], \text{ where } A = [a_{k,\ell}].
$$

Two extensions to the tensor product in SPIRAL are the *row* and the *column overlapped tensor product*, defined by

$$
I_n \otimes_k A = \begin{bmatrix} \boxed{A} & & \\ & \boxed{A} & \\ & & \ddots & \\ & & & \boxed{A} \end{bmatrix}
$$
$$
I_n \otimes^k A = \begin{bmatrix} \boxed{A} & & \\ & \boxed{A} & \\ & & \ddots & \\ & & & \boxed{A} \end{bmatrix}. \tag{15}
$$

Above, $\otimes_k$ overlaps the block matrices A by k columns, while below, $\otimes^k$ overlaps the block matrices A by k rows.

Table 2
Some Rules for Trigonometric Transforms

$$\mathbf{DFT}_n \rightarrow (\mathbf{DFT}_k \otimes \mathrm{I}_m)\, \mathrm{T}^n_m (\mathrm{I}_k \otimes \mathbf{DFT}_m)\, \mathrm{L}^n_k, \quad n = km \tag{20}$$

$$\mathbf{DFT}_n \rightarrow P_n(\mathbf{DFT}_k \otimes \mathbf{DFT}_m)Q_n, \quad n = km,\ \gcd(k,m) = 1 \tag{21}$$

$$\mathbf{DFT}_p \rightarrow R_p^T(\mathrm{I}_1 \oplus \mathbf{DFT}_{p-1})D_p(\mathrm{I}_1 \oplus \mathbf{DFT}_{p-1})R_p, \quad p \text{ prime} \tag{22}$$

$$\mathbf{DCT\text{-}3}_n \rightarrow (\mathrm{I}_m \oplus \mathrm{J}_m)\, \mathrm{L}^n_m(\mathbf{DCT\text{-}3}_m(1/4) \oplus \mathbf{DCT\text{-}3}_m(3/4))(\mathrm{F}_2 \otimes \mathrm{I}_m) \begin{bmatrix} \mathrm{I}_m & 0 \oplus -\mathrm{J}_{m-1} \\ & \frac{1}{\sqrt{2}}(\mathrm{I}_1 \oplus 2\,\mathrm{I}_m) \end{bmatrix}, \quad n = 2m \tag{23}$$

$$\mathbf{DCT\text{-}4}_n \rightarrow S_n \mathbf{DCT\text{-}2}_n \operatorname{diag}_{0 \le k < n}\left(1/(2\cos\tfrac{(2k+1)\pi}{4n})\right) \tag{24}$$

$$\mathbf{IMDCT}_{2m} \rightarrow (\mathrm{J}_m \oplus \mathrm{I}_m \oplus \mathrm{I}_m \oplus \mathrm{J}_m)\left(\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes \mathrm{I}_m\right) \oplus \left(\begin{bmatrix} -1 \\ -1 \end{bmatrix} \otimes \mathrm{I}_m\right)\right) \mathrm{J}_{2m}\, \mathbf{DCT\text{-}4}_{2m} \tag{25}$$

$$\mathbf{WHT}_{2^k} \rightarrow \prod_{i=1}^{t} (\mathrm{I}_{2^{k_1+\cdots+k_{i-1}}} \otimes \mathbf{WHT}_{2^{k_i}} \otimes \mathrm{I}_{2^{k_{i+1}+\cdots+k_t}}), \quad k = k_1 + \cdots + k_t \tag{26}$$

Note that the left operand in $\otimes_k$ and $\otimes^k$ has to be the identity matrix. SPIRAL also uses a similarly defined *row* and *column overlapped direct sum* $\oplus_k$ and $\oplus^k$, respectively.

Conversion to real data format $\overline{(\cdot)}$. Complex transforms are usually implemented using real arithmetic. Various data formats are possible when converting complex into real arithmetic, the most popular being probably the *interleaved complex format*, in which a complex vector is represented by alternating real and imaginary parts of the entries. To express this conversion in the mathematical framework of SPIRAL, we first observe that the complex multiplication $(u+iv)(y+iz)$ is equivalent to the real multiplication

$$\begin{bmatrix} u & -v \\ v & u \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix}.$$

Thus, the complex matrix–vector multiplication $Mx \in \mathbb{C}^n$ corresponds to $\overline{M}x' \in \mathbb{R}^{2n}$, where $\overline{M}$ arises from M by replacing every entry $u+iv$ by the corresponding 2×2 matrix above, and x' is in interleaved complex format. Thus, to translate complex formulas into real formulas in the interleaved format, SPL introduces the new operator $\overline{(\cdot)} : M \mapsto \overline{M}$, where M is any SPL formula. Other formats can be handled similarly.

Examples. We now illustrate SPL using several simple examples. The full relevance of SPL will become clear in Section IV.

DCT, type 2, size 4. We return to the $\mathbf{DCT\text{-}2}_4$ factorization in (14). In SPL, it takes the concise form

$$\mathbf{DCT\text{-}2}_4 = L^4_2(\mathbf{DCT\text{-}2}_2 \oplus \mathbf{DCT\text{-}4}_2) \cdot (F_2 \otimes I_2)(I_2 \oplus J_2). \tag{16}$$

The stride permutation L^4_2 is the left matrix in the sparse factorization of (14) while the direct sum of the two DCTs in (16) is the middle matrix in (14). The last factor in (14) is split into the last two factors in (16).

Downsampling. The downsampling-by-two operator used, e.g., in the DWT transform (13) is given by

$$(\downarrow 2)_n = [\, I_{n/2} \quad 0_{n/2}\,]\, L^n_2.$$

Transform definitions. Using SPL, we can define some of the previously introduced transforms more concisely. Examples include the WHT in (9) and the filter transform in (11), which become

$$\mathbf{WHT}_{2^k} = F_2 \otimes \cdots \otimes F_2,\ (k\text{-fold}) \tag{17}$$

$$\mathbf{Filt}_n(h[z]) = I_n \otimes_{l+r} [h_l \cdots h_0 \cdots h_{-r}]. \tag{18}$$

Multidimensional transforms. If $\mathbf{T}_n$ is a transform, then its m-dimensional counterpart $\boldsymbol{m}\mathbf{D}\text{-}\mathbf{T}_{n_1 \times \cdots \times n_m}$ for an $n_1 \times \cdots \times n_m$ input array, arranged lexicographically into a vector, is the m-fold tensor product

$$\boldsymbol{m}\mathbf{D}\text{-}\mathbf{T}_{n_1 \times \cdots \times n_m} = \mathbf{T}_{n_1} \otimes \cdots \otimes \mathbf{T}_{n_m}. \tag{19}$$

For example, $\mathbf{2D}\text{-}\mathbf{DFT}_{n_1 \times n_2} = \mathbf{DFT}_{n_1} \otimes \mathbf{DFT}_{n_2}$ is the two-dimensional DFT on an $n_1 \times n_2$ input array arranged into a vector in row-major order.

C. Rules

We have indicated before that the language SPL was introduced to represent transform algorithms. In this section we present the framework to capture and generate these algorithms using *rules*. As we mentioned in Section II, SPIRAL has two types of rules, breakdown rules and manipulation rules, which have different purposes. Breakdown rules are used by the Formula Generation block (see Fig. 1) to generate algorithms, represented as SPL formulas. Manipulation rules are used by the Formula Optimization block to optimize algorithms. We discuss both types in detail below.

Breakdown rules. A breakdown rule is a decomposition of a transform into a product of structured sparse matrices that may contain other, usually smaller, transforms. We showed earlier an example for the $\mathbf{DCT\text{-}2}_4$ in (16). Formally, a breakdown rule is an equation of matrices, in which the left-hand side is a transform and the right-hand side is an SPL formula. We use "$\rightarrow$" instead of "$=$" to emphasize that it is a rule. A small subset of the rules for trigonometric transforms, available in SPIRAL's rule database, are listed here.

Breakdown rules: trigonometric transforms. The rules are shown in Table 2. Rule (20) is the Cooley–Tukey FFT rule. Rule (21) is the prime-factor FFT from Good-Thomas; P_n, Q_n are permutations (see [34], [36] for details). Rule (22) is Rader's FFT algorithm (see [36]) and is used for prime sizes;

Table 3
Some Rules for Filters and the DWT

$$\mathbf{Filt}_n\left(h[z]\right) \rightarrow \mathrm{I}_s \otimes_{l+r} \mathbf{Filt}_{n/s}\left(h[z]\right), \quad h[z] = \sum_{-r\le k\le l} h_k z^{-k} \tag{28}$$

$$\mathbf{Filt}_n^{\mathrm{per,per}}\left(h[z]\right) \rightarrow \mathbf{DFT}_n^{-1}\,\mathrm{diag}_{0\le k<n}(a_k)\,\mathbf{DFT}_n, \quad (a_0,\ldots,a_{n-1})^T = \mathbf{DFT}_n\,\hat{h} \tag{29}$$

$$\mathbf{DWT}_n^{f_l,f_r}\left(h[z],g[z]\right) \rightarrow \left(\mathbf{DWT}_{n/2}^{f_l,f_r}\left(h[z],g[z]\right)\oplus \mathrm{I}_{n/2}\right)(\downarrow 2)_n\,\mathbf{Filt}_n^{f_l,f_r}\left(\begin{bmatrix} h[z] \\ g[z] \end{bmatrix}\right) \tag{30}$$

$$\mathbf{DWT}_n^{f_l,f_r}\left(h[z],g[z]\right) \rightarrow \left(\mathbf{DWT}_{n/2}^{f_l,f_r}\left(h[z],g[z]\right)\oplus \mathrm{I}_{n/2}\right)\begin{bmatrix} \mathrm{I}_{n/2}\otimes_{l+r-1}[\,h_l\cdots h_0\cdots h_{-r}\,] \\ \mathrm{I}_{n/2}\otimes_{l+r-1}[\,g_l\cdots g_0\cdots g_{-r}\,] \end{bmatrix}\mathrm{E}_{n,l,r-1}^{f_l,f_r} \tag{31}$$

$$\mathbf{DWT}_n^{f_l,f_r}\left(h[z],g[z]\right) \rightarrow \left(\mathbf{DWT}_{n/2}^{f_l,f_r}\left(h[z],g[z]\right)\oplus \mathrm{I}_{n/2}\right)\mathbf{Filt}_n^{f_l,f_r}\left(\begin{bmatrix} h_{\mathrm{even}}[z] & h_{\mathrm{odd}}[z] \\ g_{\mathrm{even}}[z] & g_{\mathrm{odd}}[z] \end{bmatrix}\right)\mathrm{L}_2^n \tag{32}$$

R_p is a permutation and D_p is the direct sum of a 2×2 matrix and a diagonal. Rule (23) was recently derived [37]. Note that transposition of this rule yields a rule for $\mathbf{DCT\text{-}2}_n$. Finally, (26) is an iterative rule for the WHT.

Next, we consider rules for the filter and the DWTs.

Breakdown rules: filter transform and DWT. Filter banks can be represented by matrices of filters, [38]. For example, for two FIR filters given by $h[z]$ and $g[z]$, one stage of a corresponding filter bank is given by the transform

$$\mathbf{Filt}_n\left(\begin{bmatrix} h[z] \\ g[z] \end{bmatrix}\right) = \begin{bmatrix} \mathbf{Filt}_n(h[z]) \\ \mathbf{Filt}_n(g[z]) \end{bmatrix}. \tag{27}$$

This will be used in the breakdown rules for filter transforms and for the DWT shown in Table 3. Rule (28) is the overlap-save convolution rule [31]. Rule (29) arises from the convolution theorem of the DFT [31]. Elements of the diagonal matrix are the DFT coefficients of $\hat{h}$ where $\hat{h}[z] = h[z] \bmod (z^n-1)$. Rule (30) represents Mallat's algorithm for computation of the DWT (e.g., [39]) and could also be used to define the DWT. Rule (31) is similar to (30) but the downsampling operator is fused into the filter matrix to save half of the number of operations. Rule (32) is the polyphase decomposition for the DWT [39] and requires $f_l, f_r \in \{\mathrm{per}, \mathrm{zero}\}$. There are many other breakdown rules for the DWT included in SPIRAL, most notably the *lifting rule* that decomposes polyphase filter banks into lifting steps [40].

Terminal breakdown rules. Finally, we also use rules to terminate base cases, which usually means transforms of size 2. The right-hand side of a *terminal rule* does not contain any transform. Examples include for the trigonometric transforms

$$\begin{aligned} \mathbf{DFT}_2 &\rightarrow F_2 \\ \mathbf{DCT\text{-}2}_2 &\rightarrow \mathrm{diag}\left(1, \frac{1}{\sqrt{2}}\right) F_2 \\ \mathbf{DCT\text{-}4}_2 &\rightarrow J_2 R_{13\pi/8} \end{aligned} \tag{33}$$

and for the DWT

$$\mathbf{DWT}_2^{f_l,f_r}(h[z],g[z]) \rightarrow \begin{bmatrix} h_l & \cdots & h_0 & \cdots & h_{-r} \\ g_l & \cdots & g_0 & \cdots & g_{-r} \end{bmatrix} E_{2,l,r}^{f_l,f_r}.$$

The above breakdown rules, with the exception of (23), are well known in the literature; but they are usually expressed using elaborate expressions involving summations and with complicated index manipulations. In contrast, equations (20) to (32) are not only compact but also clearly exhibit the structure of the rules. Although these rules are very different from each other, they only include the few constructs in SPL, which makes it possible to translate the algorithms generated from these rules into code (see Section IV). As a final note, we mention that SPIRAL's database includes over 100 breakdown rules.

Manipulation rules. A manipulation rule is a matrix equation in which both sides are SPL formulas, neither of which contains any transforms. These rules are used to manipulate the structure of an SPL formula that has been fully expanded using breakdown rules. Examples involving the tensor product include

$$A_m \otimes B_n \rightarrow (A_m \otimes I_n)(I_m \otimes B_n) \tag{34}$$

$$\begin{aligned} (B_n \otimes A_m) &\rightarrow L_n^{mn}(A_m \otimes B_n)L_m^{mn} \\ &\rightarrow (A_m \otimes B_n)^{L_m^{mn}} \end{aligned} \tag{35}$$

where $(A_m \otimes B_n)^{L_m^{mn}}$ is the notation for *matrix conjugation* defined in this case by the middle term of (35). Rule (34) is referred to as the multiplicative property of the tensor product. These are some of the manipulation rules available for the tensor product, see [41].

Manipulation rules for the stride permutation [33] include the following:

$$(L_m^{mn})^{-1} \rightarrow L_n^{mn} \tag{36}$$

$$L_m^{kmn}L_n^{kmn} \rightarrow L_n^{kmn}L_m^{kmn} \rightarrow L_{mn}^{kmn} \tag{37}$$

$$L_n^{kmn} \rightarrow \left(L_n^{kn}\otimes I_m\right)(I_k \otimes L_n^{mn}) \tag{38}$$

$$L_{km}^{kmn} \rightarrow (I_k \otimes L_m^{mn})\left(L_k^{kn}\otimes I_m\right). \tag{39}$$

We introduced in Section III-B the operator $\overline{(\cdot)}$ that we used to translate complex formulas into real formulas in the complex interleaved format. Manipulation rules for this construct include

$$\begin{aligned} \overline{A} &\rightarrow A \otimes I_2, \quad \text{for } A \text{ real} \\ \overline{AB} &\rightarrow \overline{A}\,\overline{B} \\ \overline{A \oplus B} &\rightarrow \overline{A} \oplus \overline{B} \\ \overline{I_m \otimes A} &\rightarrow I_m \otimes \overline{A} \\ \overline{A \otimes I_m} &\rightarrow \left(I_n \otimes L_m^{2m}\right)\left(\overline{A} \otimes I_m\right)\left(I_n \otimes L_2^{2m}\right). \end{aligned}$$

A different data format for complex transforms leads to a different operator $\overline{(\cdot)}$ and to different manipulation rules.

SPIRAL uses currently about 20 manipulation rules; this number will increase as SPIRAL evolves.

D. Ruletrees and Formulas

Ruletrees. Recursively applying rules to a given transform to obtain a fully expanded formula leads conceptually to a tree, which in SPIRAL we call a *ruletree*. Each node of the tree contains the transform and the rule applied at this node. As a simple example, consider the $\mathbf{DCT\text{-}2}_4$, expanded first as in (16) and then completely expanded using the base case rules (33). The corresponding tree (with the rules omitted) is given by

$$\begin{array}{c} \mathbf{DCT\text{-}2_4} \\ \bigwedge \\ \mathbf{DCT\text{-}2_2} \quad \mathbf{DCT\text{-}4_2}. \end{array} \tag{40}$$

We always assume that a ruletree is fully expanded. A ruletree clearly shows which rules are used to expand the transform and, thus, uniquely defines an algorithm to compute the transform. We will show in Section III-B that, by labeling specific components of the trees with *tags*, the ruletree also fixes degrees of freedom for the resulting implementation. Ruletrees are convenient representations of the SPL formulas they represent: they keep the relevant information for creating the formula, they are storage efficient, and they can be manipulated easily, e.g., by changing the expansion of a subtree. All these issues, particularly the last one, are very important for our search methods (see Section VI-A), since they require the efficient generation of many ruletrees for the same transform. We also use the ruletree representation for defining "features" of a formula to enable learning methods, see Section VI-B. However, when translatinginto code, it is necessary to convert the ruletree into an explicit SPL formula.

Formulas. Expanding a ruletree by recursively applying the specified rules top-down, yields a *completely expanded (SPL) formula*, or simply a *formula*. Both the ruletree and the formula specify the same fast algorithm for the transform, but in a different form. The information about the intermediate expansions of the transform is lost in the formula, but the formula captures the structure and the dataflow of the computation, and hence can be mapped into code. As an example, the completely expanded formula corresponding to (14), (16), and (40) is given by

$$\mathbf{DCT\text{-}2}_4 = L^4_2 \left(\operatorname{diag}\left(1, \frac{1}{\sqrt{2}}\right) F_2 \oplus J_2 R_{13\pi/8} \right) \cdot (F_2 \otimes I_2)(I_2 \oplus J_2). \tag{41}$$

The formula in (16) cannot be translated into code in SPIRAL because it is not fully expanded: its right-hand side contains the transforms $\mathbf{DCT\text{-}2}_2$ and $\mathbf{DCT\text{-}4}_2$, which are *non*terminals. In contrast, (41) is a fully expanded formula since it expresses $\mathbf{DCT\text{-}2}_4$ exclusively in terms of terminal SPL constructs. A fully expanded formula can be translated into code.

The above rule framework defines a formal language that is a subset of SPL. The nonterminal symbols are the transforms, the rules are the breakdown rules available in SPIRAL, and the generated language consists of those formulas that are fast algorithms for the transforms.

Alternatively, we can regard this framework as a term rewriting system [42]. The terms are the formulas, the variables are the transform sizes (or, more general, the transform parameters), the constants are other SPL constructs, and the rules are the breakdown rules. The transform algorithms are those formulas in normal form. If we consider only rules that decompose a transform into smaller transforms such as (20) or that terminate transforms such as (33), then it is easy to prove that formula generation terminates for a given transform. However, the existence of translation rules such as (24) may introduce infinite loops. In practice, we make sure that we only include translation rules that translate transforms of higher complexity into transforms of lower complexity to ensure termination. Obviously, the rewriting system is not confluent—and it is not meant to be—since the purpose is to combine the various rules to generate many different algorithms for each transform.

Formula space $\mathcal{F}$. In general, there are few rules (say less than ten) per transform, but the choices during the recursive expansion lead to a large number of different formulas. These choices arise from the choice of rule in each step, but also, in some cases, from different instantiations of one rule (e.g., rule (20) has a degree of freedom in factoring the transform size). When a formula is recursively generated, these choices lead to a combinatorial explosion and, in most cases, to an exponentially growing number of formulas for a given transform. The different formulas for one transform all have similar arithmetic cost (number of additions and multiplications) equal or close to the best known (due to the choice of "good" or "fast" rules), but differ in dataflow, which in turn leads to a usually large spread in runtime. Finding the best formula is the challenge.

The set of alternative formulas that can be generated by recursive application of applicable rules constitute the set of formulas $\mathcal{F}$. Even though this set is very large, its recursive structure allows search methods such as dynamic programming and evolutionary search—see Section VI-A—to operate quite efficiently.

E. Formula Generation

The framework presented in the previous section provides a clear road map on how to implement the Formula Generation block in SPIRAL (see Fig. 1). The block needs three databases to generate the formula space: one defines the transforms and the other two define the breakdown and manipulation rules, respectively. Information about transforms includes their definition (for verification of formulas and code), type and scope of parameters (at least the size), and how to formally transpose them. Information provided for rules includes their applicability (i.e., for which transform and parameters), children, and the actual rule. Ruletrees and formulas are both implemented as recursive data types. A more detailed description can be found in [43], [44].

We used the GAP 3 [45] computer algebra system to implement the high-level components of SPIRAL including the Formula Generation, the Formula Optimization, the Search, and the user interface. GAP was chosen for the following reasons:

Table 4
Examples of Templates for SPL Constructs: Symbols

L^{str}_{size} $[size \geq 1 \wedge str \geq 1 \wedge size \mod str = 0]$
`blk = size / str`
`do i0 = 0..str-1`
`  do i1 = 0..blk-1`
`  y[i1 + i0*blk] = x[i0*str + i1]`
`  end`
`end`
	J_{size} $[size \geq 1]$
	`blk = size / str`
	`do i0 = 0..str-1`
	`  do i1 = 0..blk-1`
	`  y[i1 + i0*blk] = x[i0*str + i1]`
	`  end`
	`end`

F_2
`y[0] = x[0] + x[1]`
`y[1] = x[0] - x[1]`
	$\mathrm{diag}(D)$
	`do i0 = 0..Length(D)-1`
	`  y[i0] = D[i0] * x[i0]`
	`end`

1) GAP provides data types and functions for symbolic computation, including exact arithmetic for rational numbers, square roots of rational numbers, roots of unity, and cosine and sines of angles $r\pi$, where r is a rational number. These are sufficient to represent most transforms and rules, and *exact* arithmetic can be used to formally verify rules and formulas (see Section V-B).
2) GAP can be easily extended.
3) GAP is easy to interface with other programs and the GAP kernel can be modified when necessary, since the full source code is available.

IV. From SPL Formulas to Code

In this section, we discuss the second level in SPIRAL, the Implementation Level (see Fig. 1), which comprises the two blocks Implementation and Code Optimization. We also refer to this level as the SPL Compiler, since its purpose is to translate SPL formulas into code. By generating code for a formula A, we mean generating code for the matrix vector multiplication $y = Ax$, where x and y are input and output vectors of suitable size.

Up to this point, the motivation to consider the formula representation of transforms has been purely mathematical: SPL is a natural representation of algorithms from the algorithms expert's point of view, and SPL enables the generation of many alternative formulas for the same transform. However, as we will see in this section, SPL's ruletrees and formulas also retain the necessary information to translate formulas into efficient code, including vector and parallel code. Furthermore, SPL facilitates the manipulation of algorithms using rules (see Section III-C). This manipulation enables SPIRAL to optimize the dataflow patterns of algorithms at the high, mathematical level. Current compilers strive to accomplish such optimizations on the code level but, in the domain of transforms, very often fail or optimize only to a rather limited degree. In Section VII, we will show experiments that demonstrate this problem.

In the following, we first slightly extend the language SPL as introduced in Section III-B through the notion of tags that fix implementation choices when SPL is translated into code. Then, we introduce a major concept in SPL—the *template* mechanism, which defines the code generation. Finally, we explain standard (scalar) code generation and, with less detail, vector code generation and the first experiences in SPIRAL with parallel code generation.

A. SPL and Templates

As introduced in Section III-B, Table 1, SPL describes transform algorithms as formulas in a concise mathematical notation.

Implementation choices: tags. Besides formula constructs, SPL supports *tags* in ruletrees and the corresponding formulas. The purpose of these tags is to control implementation choices, i.e., to instruct the compiler to choose a specific code generation option, thus fixing the degrees of freedom in the compiler. In the current version of SPIRAL, the most important implementation choice considered is the degree of unrolling, which can be controlled either *globally* or *locally*. The global unrolling strategy is determined by an integer threshold that specifies the smallest size of (the matrix corresponding to) a subformula to be translated into loop code. This threshold may be overridden by local tags in the formula that allow a finer control. Experiments have shown that a global setting is sufficient in most cases [44]. Tags will most likely become even more relevant in future versions of SPIRAL when more implementation strategies with indeterminate outcome are included.

Templates. The translation of SPL formulas to code is defined through *templates*. A template consists of a parameterized formula construct A, a set of conditions on the formula parameters, and a C-like code fragment. Table 4 shows templates for several SPL symbols: the stride permutation L^{mn}_m, the J_n matrix, the butterfly matrix F_2, and a generic diagonal matrix D. Table 5 shows templates for several matrix constructs.

Templates serve four main purposes in SPIRAL: 1) they specify how to translate formulas into code; 2) they are a tool for experimenting with different ways of mapping a formula into code; 3) they enable the extension of SPL with additional constructs that may be needed to express new DSP algorithms or transforms not yet included in SPIRAL; and

Table 5
Examples of Templates for SPL Constructs: Matrix Constructs

$A \cdot B$
 	
`deftemp t Rows(B)`
`call B(t, x)`
`call A(y, t)`
	$\mathrm{I}_n \otimes A$
	
	`do i0 = 0..n-1`
	`  call A(subvec(y, i0*Rows(A),`
	`  (i0+1)*Rows(A)-1),`
	`  subvec(x, i0*Cols(A),`
	`  (i0+1)*Cols(A)-1));`
	`end`

$A \oplus B$
 	
`call A(subvec(y, 0, Rows(A)-1),`
`  subvec(x, 0, Cols(A)-1))`
`call B(subvec(y, Rows(A),`
`  Rows(A)+Rows(B)-1),`
`  subvec(x, Cols(A),`
`  Cols(A)+Cols(B)-1))`
	Commonly used key words:
	
	`call A(y,x)`: inserts code for block A with input x and output y
	`subvec(v, start, end)`: returns a subvector of v
	`deftemp v N`: defines a new temporary vector v of N elements
	`Rows(A)`: returns row dimension of A
	`Cols(A)`: returns column dimension of A

4) they facilitate extending the SPL compiler to generate special code types such as code with vector instructions (see Section IV-E).

Each template is written as a separate function implementing a parameterized SPL construct with its own scope for variables. However, when incorporated into the generated code, the variables local to different templates are given unique names to disambiguate them and to incorporate them into one common name space. The template code is specialized by substituting all of the template parameters (e.g., size and str in $L^{\mathrm{size}}_{\mathrm{str}}$) by their respective values.

Although the template specialization step is similar to the partial evaluation described in [46], it does not require complicated binding-time analysis, because the only control flow statements in the code generated from formulas are loops with known bounds. This is because currently, SPIRAL does not generate code for parameterized transforms, but only for instantiations. Transform size and other parameters are already fixed in the formula generation process. This makes the specialization of the initial code generated from the formula straightforward.

B. Standard Code Generation

The SPL compiler translates a given SPL program describing a formula into C (or Fortran) code. This translation is carried out in several stages shown in Fig. 3.

Intermediate Code Generation. The first stage of the compiler traverses the SPL expression tree top-down, recursively matches subtrees with templates, and generates C-like intermediate code from the corresponding template by specializing the template parameters with the values obtained from the formula.

Next, based on the local unrolling tags and the global unrolling threshold, the compiler identifies loops that should be unrolled and marks them accordingly in the intermediate representation.

Constructs like "diag" or other generic matrices allow lists of constant scalar values as arguments. Constants are saved in constant tables, $matN$, to enable looping. These tables

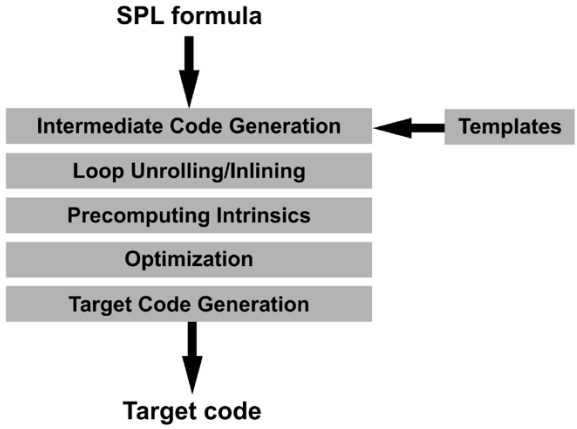


Fig. 3. SPL compiler.

are used in the subsequent compiler stages. If the loop is unrolled, the table references are expanded back into constants; if the loop is not unrolled, the table is part of the generated code.

Loop unrolling. Loops marked for unrolling are fully unrolled; currently, the SPL compiler does not support partial unrolling. A reasonably large degree of unrolling is usually very beneficial, as it creates many opportunities for optimizations. As a simple example, consider the rotation matrix

$$R_{\pi/8} = \begin{bmatrix} \cos\frac{\pi}{8} & \sin\frac{\pi}{8} \\ -\sin\frac{\pi}{8} & \cos\frac{\pi}{8} \end{bmatrix}.$$

Since there is no special template for a rotation, the compiler generates a regular matrix multiplication block with two nested loops, and a separate data table $mat0$ to contain the elements of the matrix. This code and the resulting unrolled code is shown below.

```
loop code:
 for (i0 = 0; i0 < 2; i0 + +) {
  y[i0] = 0;
  for (i1 = 0; i1 < 2; i1 + +) {
```

```
    f0 = mat0[i0*2 + i1]*x[i1];
    y[i0] = y[i0] + f0;
   }
  }
```

```
unrolled code:
 y[0] = 0;
 f0 = 0.923 879 532 511 286 7*x[0];
 y[0] = y[0] + f0;
 f0 = 0.382 683 432 365 089 8*x[1];
 y[0] = y[0] + f0;
 y[1] = 0;
 f0 = (−0.382 683 432 365 089 8)*x[0];
 y[1] = y + f0;
 f0 = 0.923 879 532 511 286 7*x[1];
 y[1] = y[1] + f0;
```

As this example shows, full unrolling enables constant table references to be inlined and additional optimizations to be performed. In this case all additions of zero can be eliminated.

Precomputation of intrinsics. Besides constants, the code may call predefined transcendental functions such as $\sin(\cdot)$ to represent scalars. These functions are called *intrinsic*, because the compiler has special support for handling them.

When the compiler encounters an intrinsic function, the function call is not inserted in the target code. Instead, all possible arguments to the function are computed by the compiler. This is possible, since all loop bounds are known at compile time. The compiler will then replace the original expressions by references to tables of constants whose values are either computed at compile time or initialized at runtime, depending on the compiler configuration. In the case they are initialized at runtime, the compiler produces an initialization function.

Optimization. The optimization stage performs dead code and common subexpression elimination (CSE), strength reduction, copy propagation, and conversion to scalars of temporary vector references with constant indexes. This stage will be discussed in detail in Section IV-C.

Target code generation. In the last stage, the compiler produces the target code. The target code is customizable with the following options.

Standard code generation backends generate standard C and Fortran code including the required function declaration, constant tables, and the initialization function for precomputing intrinsics. We focus our discussion on C code generation. The fused multiply–add (FMA) backend performs an instruction selection to produce C code that utilizes FMA instructions available on some platforms. The multiplierless backend decomposes constant multiplications into additions, subtractions, and shifts.

Graphical backends produce transform dataflow graphs. These graphs are useful to visualize, analyze, and compare different code options.

Statistical backends output statistics of the generated code, rather than the code itself. Examples include the arithmetic cost, the FMA arithmetic cost, the size of the intermediate storage required, or the estimated accuracy. These statistics can be used as alternatives to runtime for the optimization criteria used by SPIRAL (see Section V-C). The arithmetic cost backend, for instance, enables SPIRAL to search for formulas that implement the transform with the minimal number of arithmetic operations.

C. Code Optimization

In this section, we provide further detail on the optimization stage of the SPL compiler, the fourth block in Fig. 3. The reason why the SPL compiler performs these optimizations rather than leaving them to the C/Fortran compiler is that practically all of the commonly used compilers do not optimize machine generated code well, in particular, large segments of straightline code (see [11], [20], [47], [48]). The performed optimizations include array scalarization, algebraic simplification, constant and copy propagation, CSE, and dead code elimination. The first four optimizations will be investigated in more detail below. Dead code elimination will not be discussed, as there are no unusual details of our implementation that impact performance. Finally, we briefly discuss FMA code generation.

Static single assignment (SSA). All of the optimizations considered are scalar optimizations that operate on code converted to SSA form, in which each scalar variable is assigned only once to simplify the required analysis.

Array scalarization. C compilers are very conservative when dealing with array references. As can be seen from the compilation example in the previous section, the loop unrolling stage can produce many array references with constant indexes. During array scalarization, all such occurrences are replaced by scalar temporary variables.

Algebraic simplification. This part of the optimizer performs constant folding and canonicalization, which support the efforts of other optimization passes.

Constants are canonicalized by converting them to be nonnegative and by using unary negation where necessary. Expressions are canonicalized similarly by pulling unary negation as far out as possible. For example, $-x-y$ is translated to $-(x+y)$, and $(-x)*y \rightarrow -(x*y)$. Unary operators will usually combine with additive operators in the surrounding context and disappear through simplification.

These transformations, in conjunction with copy propagation, help create opportunities, previously unavailable, for CSE to further simplify the code.

Copy propagation. Copy propagation replaces occurrences of the variable on the left-hand side of a given "simple" assignment statement with the right-hand side of that assignment statement, if the right-hand side is either a constant, a scalar, or a unary negation of a scalar or a constant.

Recall that unary negation expressions are often created during algebraic simplification due to canonicalization. Copy propagation will move them so that they can combine with

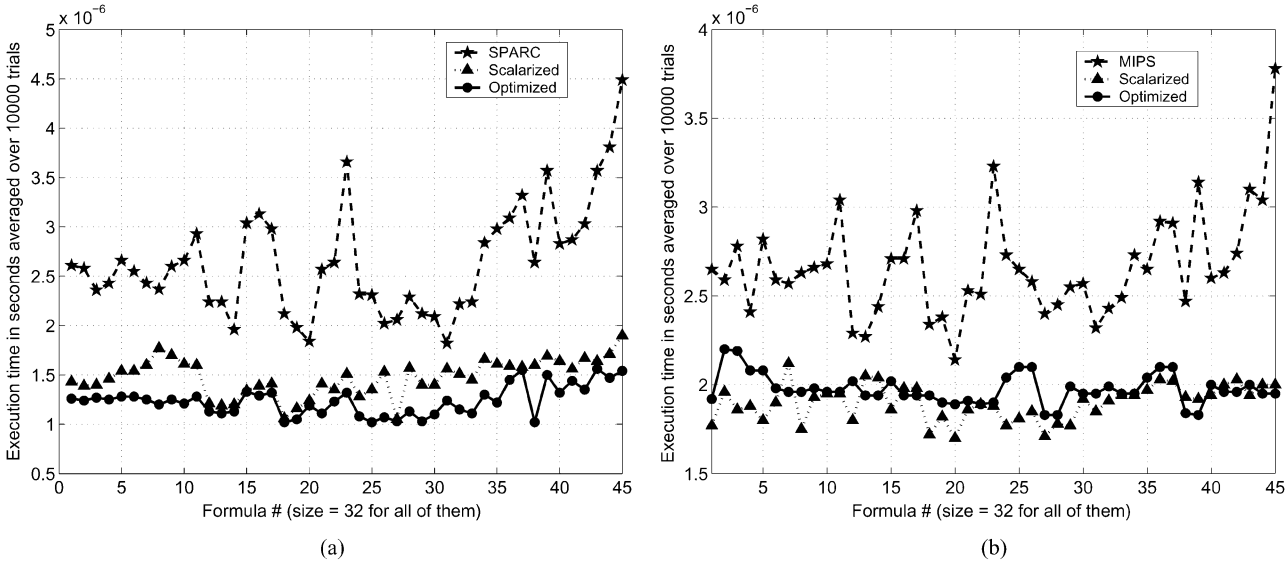


Fig. 4. DFT performance before and after SPL compiler optimizations on a SPARC and MIPS architecture. SPARC: UltraSparc III, 750 MHz, Forte Developer 7 compiler, flags -fast -xO5; MIPS:MIPS R12000, 300 MHz, MIPSpro 7.3.1.1 compiler, flag -O3. (a) SPARC. (b) MIPS.

additive operators in the new context during further algebraic simplification.

CSE. CSE tries to discover multiple occurrences of the same expression; it makes sure that these are computed only once. Our implementation treats subscripted array references as expressions and, therefore, as eligible for elimination.

Optimization strategy. The different optimizations described above have mutually beneficial relationships. For instance, algebraic simplification can bolster copy propagation, and copy propagation can then create new opportunities for algebraic simplification. Alternating between these two optimization passes, the code will eventually reach a fixed point, where it is changed no further.

Our implementation strategy is to loop over these different optimization passes in the manner prescribed, and to terminate once an entire iteration fails to change the code.

Impact of the optimizations. Merely scalarizing arrays provides a sizable performance benefit as seen in Fig. 4. These graphs depict the execution time (lower is better) of the programs generated for 45 SPIRAL generated formulas for a $\mathbf{DFT}_{32}$ on two different platforms. The line marked with stars and labeled "SPARC" in Fig. 4(a) [respectively, "MIPS" in Fig. 4(b)] shows the execution times achieved by the native SPARC (MIPS) compiler alone. The line marked with triangles and labeled "Scalarized" shows that every formula is improved by scalarizing the C code before sending it to the native compiler on both platforms. Note that we performed our MIPS experiments on an R12000 with the MIPSpro compiler. See [47] for a case where the same experiments were performed with the same compiler on an R10000, but with different results. In that case, the MIPSpro compiler already achieved good performance without scalarizing or optimizing the code first. The line marked with bullets and labeled "Optimized" in both graphs of Fig. 4 represents the performance of the DFT codes after the entire optimization following the strategy described above. We observe that the additional optimizations beyond array scalarization significantly improve the code on SPARC, but not on MIPS.

FMA code generation. Some architectures, including Itanium 1/2 and Motorola G4/G5, offer FMA instructions, which perform an instruction of the form

$$y \leftarrow \pm ax \pm b$$

as fast as a single addition or multiplication. Most standard compilers cannot generate the optimal FMA code, as it may require changing the algorithm and/or the order of computation.

To generate explicit FMA code, we use an algorithm that traverses the dataflow graph propagating multiplications and fusing them with additions where possible [49]. The algorithm has the property that the number of multiplications left "unfused" is at most the number of outputs of the transform. We implemented the algorithm by extending the *iburg* instruction selection framework for expression trees [50]. Unlike standard compilers, this algorithm can produce code that matches the best published FMA arithmetic cost for many transforms, including the DFT [51], [52].

Our FMA generation algorithm can also be performed more concisely at the formula level (similar to the vector code generation discussed below) rather than at the code level. This method is currently being integrated into SPIRAL.

D. Compilation Example

To demonstrate the most important stages of the compiler, we discuss the compilation of the SPL formula in (41). The size of the formula is four, which is smaller than the default global unrolling threshold 16. Thus, the generated code will be completely unrolled. In the unrolled code, all references to precomputed coefficient tables and transcendental functions will be inlined during the unrolling stage, and the intrinsic precomputation stage will be omitted.

Table 6
Code Generation for Formula (41). Left: Initial Code Generation. Right: After Unrolling and Inlining Constants

initial code generation:

$\underline{\mathrm{I}_2 \oplus \mathrm{J}_2}$

```
for (i0 = 0; i0 < 2; i0++) {
  t2[i0] = x[i0]; }
for (i0 = 0; i0 < 2; i0++) {
  t2[i0+2] = x[-i0+3]; }
```

$\underline{\mathrm{F}_2 \otimes \mathrm{I}_2}$

```
for (i0 = 0; i0 < 2; i0++) {
  t1[i0+2] = t2[i0] - t2[i0+2];
  t1[i0] = t2[i0] + t2[i0+2]; }
```

$\underline{\mathrm{diag}(1, \sqrt{2}/2)\,\mathrm{F}_2 \oplus \ldots}$

```
t3[1] = t1[0] - t1[1];
t3[0] = t1[0] + t1[1];
for (i0 = 0; i0 < 2; i0++) {
  t0[i0] = mat0[i0] * t3[i0]; }
```

$\underline{\ldots \oplus \mathrm{J}_2\,\mathrm{R}_\alpha}$

```
for (i0 = 0; i0 < 2; i0++) {
  t4[i0] = 0;
  for (i1 = 0; i1 < 2; i1++) {
    f0 = mat1[i0*2 + i1] * t1[i1+2];
    t4[i0] = t4[i0] + f0; } }
for (i0 = 0; i0 < 2; i0++) {
  t0[i0+2] = t4[-i0+1]; }
```

$\underline{\mathrm{L}_2^4}$

```
for (i0 = 0; i0 < 2; i0++) {
  for (i1 = 0; i1 < 2; i1++) {
    y[2*i0+i1] = t0[i0+2*i1]; } }
```

after unrolling and inlining constants:

```
t2[0] = x[0];
t2[1] = x[1];
t2[2] = x[3];
t2[3] = x[2];

t1[2] = t2[0] - t2[2];
t1[0] = t2[0] + t2[2];
t1[3] = t2[1] - t2[3];
t1[1] = t2[1] + t2[3];

t3[1] = t1[0] - t1[1];
t3[0] = t1[0] + t1[1];
t0[0] = 1.0000000000000000 * t3[0];
t0[1] = 0.7071067811865476 * t3[1];

t4[0] = 0;
f0 = 0.3826834323650898 * t1[2];
t4[0] = t4[0] + f0;
f0 = (-0.9238795325112867) * t1[3];
t4[0] = t4[0] + f0;
t4[1] = 0;
f0 = 0.9238795325112867 * t1[2];
t4[1] = t4[1] + f0;
f0 = 0.3826834323650898 * t1[3];
t4[1] = t4[1] + f0;
t0[2] = t4[1];
t0[3] = t4[0];

y[0] = t0[0];
y[1] = t0[2];
y[2] = t0[1];
y[3] = t0[3];
```

We look at the output of all the stages of the compiler for this expression.

Intermediate code generation from SPL templates. The initial stage of the compiler converts the SPL expression tree for (41) into the looped intermediate code (Table 6, left). The generated code is annotated with formula fragments to show the origin of the code.

Loop unrolling. All of the loops generated in the previous stage are unrolled because of the small transform dimension $n = 4 < 16$, where 16 is the default setting as mentioned. After full unrolling, the tables $matN$ are no longer needed, and the compiler directly substitutes the computed values (Table 6, right).

Scalar optimization and target code generation. Loop unrolling usually creates many opportunities for scalar optimizations, and also creates unnecessary temporary arrays (t0, t1, t2, t3, t4 in Table 6, right). Array scalarization converts redundant temporary arrays into scalars, and then the code is converted into SSA form (i.e., each scalar variable is assigned only once). As was mentioned earlier, this simplifies the analysis required for further optimization.

After the code optimization, the compiler outputs the target code including the transform function declaration and an initialization function. Since our unrolled code does not use any tables, the initialization function is empty. The resulting code is shown in Table 7, left. Further optional FMA optimization saves two instructions (Table 7, right).

Fig. 5 shows the two dataflow graphs, produced by the graphical backend, for the codes in Table 7. Each internal node in the graph represents either an addition (light gray circle), a multiplication by a constant (dark gray circle), or an FMA instruction (dark gray rectangle). For the latter, the input being multiplied is marked with a bold edge.

E. Vector Code Generation

Most modern processors feature short vector single-instruction, multiple-data (SIMD) extensions. This means the architecture provides data types and instructions to perform floating-point operations on short vectors at the same speed as a single, scalar operation. The short vector extensions have different names for different processors, have different vector lengths ν, and operate in single or double precision. An overview is provided in Table 8.

Short vector instructions have the potential to speed up a program considerably, provided the program's dataflow exhibits the fine-grain parallelism necessary for their application. Since vector instructions are beyond the *standard* C/Fortran programming model, it is natural to leave the vectorization to a *vectorizing* compiler. Unfortunately, to date, compiler vectorization is very limited; it fails, in particular, for the complicated access patterns usually found in transform formulas. In Section VII, for example, we will show that compiler vectorization, when used in tandem with SPIRAL, can, for the DFT, achieve moderate speedups

Table 7
Final Generated C Code and FMA Code for (41)

C code:

```
void sub(double *y, double *x) {
  double f0, f1, f2, f3, f4, f7, f8, f10, f11;
  f0 = x[0] - x[3];
  f1 = x[0] + x[3];
  f2 = x[1] - x[2];
  f3 = x[1] + x[2];
  f4 = f1 - f3;
  y[0] = f1 + f3;
  y[2] = 0.7071067811865476 * f4;
  f7 = 0.9238795325112867 * f0;
  f8 = 0.3826834323650898 * f2;
  y[1] = f7 + f8;
  f10 = 0.3826834323650898 * f0;
  f11 = (-0.9238795325112867) * f2;
  y[3] = f10 + f11;
}
void init_sub() {
}
```

FMA code:

```
void sub(double *y, double *x) {
  double f0, f1, f2, f3, f4, f1000, f1001;
  f1 = x[0] + x[3];
  f3 = x[1] + x[2];
  y[0] = f1 + f3;
  f4 = f1 - f3;
  y[2] = 0.7071067811865476 * f4;
  f0 = x[0] - x[3];
  f2 = x[1] - x[2];
  f1000 = fma(f0, 0.4142135623730951, f2);
  y[1] = 0.9238795325112867 * f1000;
  f1001 = fma(f0, (-2.4142135623730945), f2);
  y[3] = 0.3826834323650898 * f1001;
}
void init_sub() {
}
```

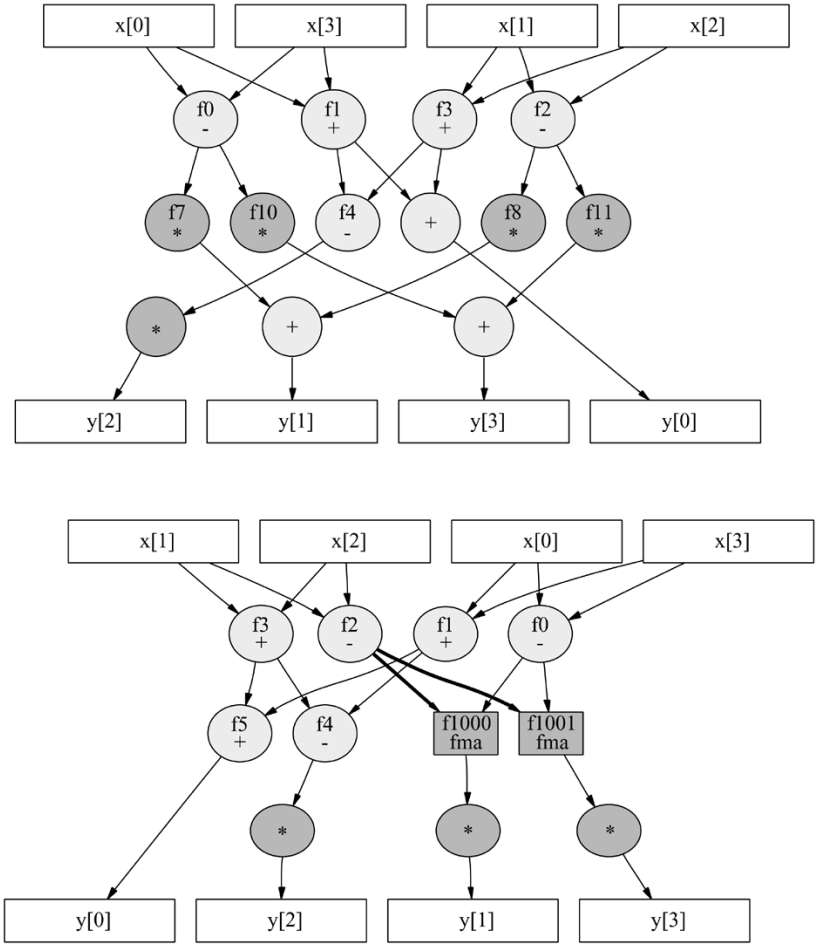


Fig. 5. Standard and FMA dataflow graphs generated by SPIRAL for (41). Multiplications are shaded dark gray, FMAs are shown as dark gray rectangles.

(about 50%), whereas the best possible code is at least a factor of two faster.

As a consequence, when striving for highest performance, the common current practice is to hand-code vector instructions. This can be done at the C level through the use of "intrinsics" provided by the respective architecture vendors, but poses major challenges to software developers.

1) Each vector extension provides different functionality and the intrinsics interface is not standardized, neither across platforms, nor across compilers, making the written code nonportable.
2) The performance of vector instructions is very sensitive to the data access; a straightforward use often *deteriorates* performance instead of improving it.

Table 8
Short Vector SIMD Extensions

Vendor	Name	ν-way	Precision	Processor
Intel	SSE	4-way	single	Pentium III Pentium 4
Intel	SSE2	2-way	double	Pentium 4
Intel	SSE3	4-way 2-way	single 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
AMD	AMD64	2-way 4-way 2-way	single single double	Athlon 64 Opteron
Motorola	AltiVec	4-way	single	MPC 74xx G4
IBM	AltiVec	4-way	single	PowerPC 970 G5
IBM	Double FPU	2-way	double	PowerPC 440 FP2

3) In a library of many transforms, each transform needs to be hand-coded individually.

In the following, we give an overview of how we overcome these problems by extending SPIRAL to *automatically* generate optimized vector code. We note that by extending SPIRAL to handle vectorization, the third difficulty is immediately taken care of. For a more detailed description, we refer to [53]–[55] and to [56] in this special issue.

Our approach to vector code generation for SPL formulas consists of two high-level steps.

- We identify which basic SPL formulas or structures within formulas can be mapped naturally into vector code; then we derive a set of *manipulation rules* that transform a given SPL formula into another formula that can be better vectorized. These manipulations are incorporated into the Formula Optimization block in Fig. 1 and can overcome compiler limitations, since they operate at the "high" mathematical level. The manipulation rules are parameterized by the vector length ν
- We define a short vector API on top of all current vector extensions, which is sufficient to vectorize a large class of SPL formulas. The API is implemented as a set of C macros. The SPL compiler is then extended to map vectorizable formulas into vector code using this API.

Formula manipulation. We start by identifying formulas that can be naturally mapped into vector code. The list is by no means exhaustive, but, as it turns out, is sufficient for a large class of formulas. We assume that the formula is real valued, i.e., if the original formula is complex, we first convert it into a real formula using the conversion operator $\overline{(\cdot)}$ and the manipulation rules introduced in Section III-C. Further, we denote the vector length with ν; on current platforms, only $\nu = 2, 4$ are available (see Table 8). We refer to a vector instruction for vectors of lengths ν also as a *ν-way* vector instruction.

The most basic construct that can be mapped exclusively into vector code is the tensor product

$$A \otimes I_\nu \tag{42}$$

where A is an *arbitrary* formula. The corresponding code is obtained by replacing each scalar operation in the code for $y = Ax$ by the corresponding ν-way vector instruction. This is best understood by visualizing the structure of $A \otimes I_\nu$; the example $F_2 \otimes I_4$ for $\nu = 4$ is provided in Table 9.

Further, the following structured matrix S

$$\begin{aligned} S &= \begin{bmatrix} \mathrm{diag}(a_0, \ldots, a_{\nu-1}) & \mathrm{diag}(b_0, \ldots, b_{\nu-1}) \\ \mathrm{diag}(c_0, \ldots, c_{\nu-1}) & \mathrm{diag}(d_0, \ldots, d_{\nu-1}) \end{bmatrix} \\ &= L_2^{2\nu} \left(\bigoplus_{0 \le i < n} \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \right) L_\nu^{2\nu} \end{aligned} \tag{43}$$

can be mapped into 4 vector multiplications and 2 vector additions. The sparse structure of S in (43) is equivalent to the structure of (42); only the actual values of the entries differ. The matrix S appears often in DSP transforms, for example, in the DFT when converting the complex twiddle factors into a real matrix using the $\overline{(\cdot)}$ operator.

Finally, we need vectorizable permutations. Clearly, permutations of the form $P \otimes I_\nu$ match (42) and are, thus, naturally mappable into vector code. Another important class consists of permutations that can be vectorized using a small number of in-register data reorganization instructions. The permutations

$$P \in \left\{ L_\nu^{\nu^2}, L_2^{2\nu}, L_\nu^{2\nu} \right\} \tag{44}$$

are of that type and play an important role in the vectorization of DFT algorithms based on the Cooley–Tukey rule (20). The actual implementations of these instructions differ across short vector architectures; however, they share the characteristics that they are done fully in-register, using only a few vector reorder instructions.

Further, if P is a vectorizable permutation of the form (42) or (44), then the same holds for $I_n \otimes P$. Finally, for $\nu = 4$, we also consider permutations of half-vectors, namely of the form $P \otimes I_2$. These permutations reorganize complex numbers into the interleaved complex format and are, thus, important for complex transforms. For example, Intel's SSE vector extension provides memory access instructions for these permutations.

Building on the constructs introduced above, we can completely vectorize any expression of the form

$$\prod_i P_i D_i (A_i \otimes I_\nu) E_i Q_i \tag{45}$$

where P_i, Q_i are vectorizable permutations, and D_i, E_i are direct sums of matrices of the form (43). The class of formulas in (45) is general enough to cover the DFT formulas based on the Cooley–Tukey breakdown rule (20), the WHT formulas based on (26), and the higher dimensional transforms (19).

Table 9
Vectorization of $y = (F_2 \otimes I_4)x$ for a Four-Way Vector Extension Using the Portable SIMD API

Matrix	Scalar Code for $y = \mathrm{F}_2\, x$	Matrix	Vector Code for $y = (\mathrm{F}_2 \otimes \mathrm{I}_4)\mathrm{x}$

$\mathrm{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	`void F2(float *y, float *x) {`
	`  y[0] = x[0] + x[1];`
	`  y[1] = x[0] - x[1];`
	`}`
		$\mathrm{F}_2 \otimes \mathrm{I}_4 = \begin{bmatrix} \mathrm{I}_4 & \mathrm{I}_4 \\ \mathrm{I}_4 & -\mathrm{I}_4 \end{bmatrix}$	`void F2xI4 (float *y, float *x) {`
			`  LOAD_VECT(x0, x + 0);`
			`  LOAD_VECT(x1, x + 4);`
			`  ADD_VECT(y0, x0, x1);`
			`  STORE_VECT(y + 0, y0);`
			`  SUB_VECT(y1, x0, x1);`
			`  STORE_VECT(y + 4, y1);`
			`}`

We briefly illustrate the vectorization manipulations with the Cooley–Tukey rule (20). To manipulate a given formula into the form (45), we use manipulation rules including (35)–(39). Using these manipulations, we can vectorize every Cooley–Tukey rule based formula, provided that for $n = km$ in (20), $\nu \mid k, m$, which implies $\nu^2 \mid n$. In this case the manipulated formula takes the following form:

$$\begin{aligned}\overline{\mathrm{DFT}}_{mn} = {} & \left(I_{mn/\nu} \otimes L^{2\nu}_{\nu}\right)\left(\overline{\mathrm{DFT}_m \otimes I_{n/\nu}} \otimes I_\nu\right)\overline{T}'^{mn}_{n} \\ & \cdot \Bigg(I_{m/\nu} \otimes \left(L^{2n}_{\nu} \otimes I_\nu\right) \\ & \qquad \cdot \left(I_{2n/\nu} \otimes L^{\nu^2}_{\nu}\right)\left(\overline{\mathrm{DFT}}_n \otimes I_\nu\right)\Bigg) \\ & \cdot \left(L^{mn/\nu}_{m/\nu} \otimes L^{2\nu}_{2}\right) \qquad (46)\end{aligned}$$

where $\overline{T}'^{mn}_{n}$ is a direct sum of matrices S shown in (43). The operator $\overline{(\cdot)}$ is as defined in Section III-B. Note that (46) matches (45) and is, hence, completely vectorizable, independently of the further expansion of the smaller occurring DFTs. This is crucial for obtaining a searchable space of formulas that exhibit different dataflows.

Code mapping. After a formula is vectorized by the means of formula manipulation, the parts of the formula matching the pattern in (45) are mapped into vector code. The remaining part of the formula is mapped into scalar code using the standard SPL compiler. For a formula matching (45), first, vector code is generated for $A_i \otimes I_\nu$ by generating scalar code for A_i and replacing the scalar operations by the corresponding ν-way vector operations (e.g., $\mathtt{t} = \mathtt{a} + \mathtt{b}$ is replaced by $\mathtt{VEC\_ADD(t,a,b)}$), and by replacing array accesses by explicit vector load and store instructions. Next, the permutations P_i and Q_i are implemented by replacing the vector loads and stores by combined load/store-and-permute macros provided by our short vector API. In the final step, the arithmetic operations required by D_i and E_i are inserted between the code for $A \otimes I_\nu$, and the vector memory access and permutations introduced in the previous step.

As a small example, we show a vector store fused with L^8_4, provided by our API, and implemented in SSE using the Intel C++ compiler intrinsics. It is one of the cases in (44) for $\nu = 4$.

```
#define STORE_L_8_4(v, w, p, q) {
 _m128 t1, t2;
 s = _mm_unpacklo_ps(v, w);
 t = _mm_unpackhi_ps(v, w);
 _mm_store_ps(p, s);
 _mm_store_ps(q, t);}
```

In this example, v, w, s, t are vectors of length 4 and the permutation is performed with the first two instructions. Assuming the vectors are indexed with 0,1,2,3, it is $s = (v_0, w_0, v_1, w_1)$ and $t = (v_2, w_2, v_3, w_3)$.

F. Code Generation for Parallel Platforms

In many situations, parallel processing may be needed due to real-time constraints or when a large amount of data needs to be processed. Despite tremendous advances, parallelizing compilers, similar to vectorizing compilers, cannot compete with the best possible hand-optimized code, even for relatively simple programs [57], [58]. In this section we show a first step toward generating parallel code using SPIRAL. The high-level approach is similar to vector code generation (see Section IV-E): 1) identify constructs that can be mapped into parallel code; 2) manipulate a given formula into these parallelizable constructs; and 3) map the manipulated formula into efficient code.

SPIRAL's constructs, in particular the tensor product and direct sum, have natural interpretations for parallel computation [33], and many of the traditional optimizations used to achieve better granularity, locality, and load balance can be achieved through formula manipulation. Using formula manipulation, SPIRAL can explore alternative formulas that may exhibit explicitly parallelizable subcomponents. Parallel implementations are obtained using parallel directives/functions in the templates for these constructs. A search can be used to find the best combination of parallel and sequential code and, thus, to minimize parallel overhead and to achieve good processor utilization.

Relatively simple extensions to the code generator can be utilized to produce parallel code for both symmetric multiprocessors (SMPs), where multiple processors share a common memory with uniform access time, and distributed-memory parallel computers, where remote memory is accessed over an interconnect with nonuniform memory access. For distributed-memory computers, code can be produced by using a shared-memory programming model where remote memory is accessed implicitly, or by using a distributed-memory programming model, where explicit message passing is required. In either approach, alternative formulas for the same transform may access memory in different patterns leading to more or less remote memory accesses.

Table 10
Pseudocode for an SMP Implementation of the WHT of Size $n = 2^k$ and Distributed-Memory Pseudocode for the Stride Permutation

SMP code for WHT:

```
#begin parallel region
r = n; s = 1; id = get_thread_id();
num = get_total_thread();
for i = 1, ..., t
   r = r / n_i;
   for id = id, ..., r * s - 1, step = num
      j = id / s;
      k = id mod s;
      x^{n_i}_{jn_is+k,s} = WHT_{n_i} * x^{n_i}_{jn_is+k,s};
   s = s * n_i;
   #parallel barrier
#end parallel region
```

distributed memory code for the stride permutation:

```
/* Step 1: Construction of MPI data type */
Arguments: localN, totalRank, stride
block = stride / totalRank
localS = localN / stride
MPI_Type_vector(block, localS, localS*totalRank,
            DOUBLE, &NEW_TYPE)
MPI_Type_commit(&NEW_TYPE)

for (round = 0; round < totalRank; ++ round)
   /* Step 2: Local data rearrangement */
   id = handshake[round]
   k = 0
   offset_s = id * block
   for (i = offset_s; i < (block + offset_s); ++ i)
      for (j = i; j < localSize; j += stride)
         buffer[k++] = x[j]

   /* Step 3: Global communication */
   offset_r = id * localS
   MPI_Sendrecv(buffer, localN/totalRank, DOUBLE,
         id, 0, y+offset_r, 1, NEW_TYPE, id, 0,
         MPI_COMM_WORLD, &status)
```

We realized these ideas in preliminary experiments with the WHT on both shared-memory multiprocessors [59] and distributed-memory computers [60]. The parallel code was generated using OpenMP [61] for shared-memory parallelism and MPI [62] for distributed-memory parallelism. We performed additional experiments using a special-purpose distributed-memory parallel computer designed for the computation of the DFT and WHT [63]. In this case, a search over a family of related formulas was used to minimize the number of remote memory accesses.

Rule (26) decomposes the WHT into a sequence of factors of the form $(I_m \otimes \mathbf{WHT}_{2^{k_i}} \otimes I_n)$ containing mn independent computations of $\mathbf{WHT}_{2^{k_i}}$ at stride n, which can be computed in parallel. A barrier synchronization must be inserted between the factors. The strided access patterns may prevent prefetching associated with cache lines and may introduce false sharing where different processors share a common cache line even though they do not access common data elements [59]. Thus, rule (26) only serves as a starting point to optimize the WHT structure for parallel target platforms.

Formula manipulation. Using the manipulation rules from Section IV-E, (26) can be modified to obtain the different structure

$$\mathbf{WHT}_{2^k} = \prod_{i=1}^{t} P_i^{-1}(I_{2^k-k_i} \otimes \mathbf{WHT}_{2^{k_i}})P_i \qquad (47)$$

where P_i is a sequence of permutations. One possibility is to choose $P_i = L^{2^k}_{2^{k_i}}$. There are also other choices, since the sequence of permutations P_i is not unique. When $t = 2$ and the permutations are computed at runtime, the algorithm of [26] is obtained. This variant can lead to better performance on SMPs due to reduced cache misses and bus traffic. In a distributed memory environment, different sequences of permutations lead to different locality and the SPIRAL search engine can be used to determine the sequence with the minimal number of remote memory accesses.

Further manipulation can be used to combine adjacent permutations to obtain

$$\mathbf{WHT}_{2^k} = \left(\prod_{i=1}^{t} Q_i(I_{2^k-k_i} \otimes \mathbf{WHT}_{2^{k_i}})\right) P_t \qquad (48)$$

where $Q_i = P_{i-1}P_i^{-1}$ (where we assume $P_0 = I_{2^k}$). This has the benefit of reducing the amount of message passing in a distributed-memory environment. Further factorization of the permutations Q_i can be used to obtain formulas that group the data into larger blocks, which can both reduce communication cost and improve cache utilization.

Code generation. Parallel code for SMPs can be generated for SPL programs through the use of parallel directives in the templates for parallel constructs such as the tensor product. It is straightforward to insert parallel loops whenever $I_n \otimes A$ occurs in a formula; however, in order to obtain good parallel efficiency, we should only introduce parallelism when it improves performance; further, it is important to avoid creating and deleting threads multiple times. It is best to create a parallel region and introduce explicit scheduling and synchronization as needed for the different constructs. Table 10 (left) shows the parallel code for an SMP implementation of the iterative rule of the WHT in equation (26); the notation $x^n_{b,s}$ indicates a subvector of x of size n

equal to $(x(b), x(b+s), \ldots, x(b+(n-1)s))$. While the code was created by using formula manipulation and the techniques of Section IV-A, the code involves features not currently supported by SPIRAL, such as variable loop bounds and in-place computation. We made experiments with this and other parallel code with a special package for computing the WHT [27], [59], [60].

Code generation for distributed memory machines is more involved. Data must be distributed amongst the processors, locality maximized, and communication minimized. If a distributed shared-memory programming model is used, explicit communication is not required; however, data access patterns must be organized to minimize remote memory access. Since SPIRAL can make modifications at the formula level, alternative data access patterns can be explored and optimized automatically. In a distributed memory programming model, explicit send/receive operations must be inserted, taking into account the data distribution. For the WHT, where the data size is a power of two, data can be distributed using the high-order bits of the data address as a processor identifier and the low-order bits as an offset into the processor's local memory. In this case, communication arises from permutations in the formula, and these permutations can be automatically converted to message-passing code (see Table 10, right, for an example). Additional details are available in [60].

V. Evaluation

After formula generation and code generation, the third conceptual key block in SPIRAL is the Evaluation Level block, which is responsible for measuring the performance of the generated code and for feeding the result into the Search/Learning block.

The Evaluation Level block fulfills three main functions: 1) compilation of the source code into machine code; 2) optional verification of the generated code; and 3) measurement of the performance of the generated code. The performance metric can be the runtime of the compiled code, or it can be some other statistics about the code such as the number of arithmetic operations, the instruction count, the number of cache misses, or the number of FMA instructions. Other performance measures such as numerical accuracy or code size can also be used. The performance evaluation block makes it easy to switch between performance measures or to add new ones.

A. Compilation

To obtain a performance measure, such as the runtime, the code generated by SPIRAL is compiled, linked with the performance measuring driver, and executed. At installation time, SPIRAL detects the machine configuration and the available compilers, defaulting to vendor-supplied compilers if available.

Interfacing external programs, like C compilers, portable across platforms and operating systems, and integrating different performance measures is a nontrivial problem. In SPIRAL we have implemented a library we call "sysconf" to provide a portable and flexible solution. For example, the sysconf library stores the information about compilers available on the machine in a set of configuration *profiles*. Each profile includes the path to the compiler and to the linker, the target language (C or Fortran) and object file extensions, the compiler invocation syntax, the compiler and linker flags, the required libraries, and the test driver execution syntax. Profiles can be nested in order to create groups; for example, if the "c.gcc" profile includes all the information necessary to use gcc, "c.gcc.opt1" and "c.gcc.opt2" may be created to differentiate between option sets with different optimization levels. Configuration profiles are very useful for benchmarking different compilers and for evaluating the effects of different compiler options. Further, profiles can be configured for cross compilation and remote execution on a different platform. For example, this capability is used to produce the IPAQ results shown in Section VII. Also, additional C-to-C optimization passes are easily incorporated into a profile to accommodate various research tools. Finally, profiles allow the execution of other programs to compute various performance measures, e.g., obtained by statically analyzing the C or compiled code.

B. Verification

SPIRAL provides several modes of (optional) verification for its automatically generated code: 1) rule verification; 2) formula verification; 3) code verification; and 4) recursive code verification. We briefly discuss these modes.

Rule verification. SPIRAL requires all transforms to have a definition, which is a function that constructs the transform matrix given its parameters. Since rules decompose transforms into other transforms, each rule can be verified for fixed parameter choices. Namely, the rule is applied to the transform once, and the resulting formula, in the formula generator, is converted into a matrix and compared to the original transform. This type of verification is usually *exact*, since most transforms and their formulas have exact representations due to the symbolic computation environment provided by GAP (see Section III-E).

Formula verification. A fully expanded formula is verified similarly to a rule by converting it into the represented matrix and comparing it to the original transform. Again, this verification is usually exact.

Both, rule verification and formula verification are performed exclusively at the formula level, i.e., no code is generated. Their purpose is to verify transform algorithms and to debug the formula generator. Code verification is discussed next.

Code verification. For the verification of the generated code, SPIRAL provides a variety of tests.

The most important test applies the generated code to an input vector x and compares the output vector y to the correct result $\hat{y}$ obtained by computing the transform by definition (the code for computing a transform by definition is also generated by SPIRAL). The norm of the error $||y - \hat{y}||$ (different norms are available) is returned, and has to be below a threshold. Two modes are available. The first mode performs this comparison on the entire set of (standard) base vectors.

The correct outputs need not be computed in this case, since they are the columns of the transform matrix. For a transform of size n, the algorithms are typically $O(n\ \log(n))$; thus, this verification is $O(n^2 \log(n))$. The second mode performs this comparison either on one or on several random vectors x. Here the cost is $O(n^2)$ for computing the correct outputs by definition.

As a variant of the above tests, two generated programs can be compared against each other on the standard basis or on a set of random vectors.

The verification on the basis described above can be extended further to obtain an actual proof of correctness. Namely, the code generated by SPIRAL contains only additions and multiplications by constants as arithmetic operations. Thus, the entire program has to encode a linear function provided all the arrays are accessed within their allowed index range (which can be tested). If two linear functions coincide on a basis, they must coincide for each input vector, which proves correctness (up to a numerical error margin).

Other verification methods we have experimented with include tests for transform specific properties, such as the convolution property of the DFT [64], [65].

In practice, because of the speed, we use the verification on one random vector, which usually proves to be sufficient. By including this verification in a loop that generates random transforms, random formulas, and random implementation options, bugs in SPIRAL can be found efficiently. To facilitate debugging, once a bug in the generated code is found, another routine recursively finds the smallest subformula that produces erroneous code.

C. Performance/Cost Measures

By default, SPIRAL uses the runtime of the generated code as a performance measure, but other measures can be chosen. This property makes SPIRAL a versatile tool that can be quickly adapted or extended to solve different code optimization problems in the transform domain. Examples of considered performance measures, besides runtime, include accuracy, operation count, and instruction count. We also started preliminary work on performance models that can be applied at the algorithmic level without compiling and executing the generated code.

Runtime. There are various ways of measuring the runtime; obtaining accurate and reproducible results is a nontrivial problem. A portable way of measuring runtime uses the C clock() and computes the runtime as an average over a large number of iterations. This implies that, for small transform sizes, the runtimes do not reflect any compulsory cache misses arising from loading the input into cache. Where possible, SPIRAL uses the processor's built-in cycle counters, which are of higher resolution and, thus, allow for much faster measurement as only a few iterations need to be timed. Depending on the precision needed (for instance, timing in the search requires less precision than timing the final result), SPIRAL may need to run such measurements multiple times and take the minimum. Taking the minimum over multiple measurements and keeping the number of repetitions per measurement low, reduces the influence of other running processes, unknown cache states, and other nondeterministic effects.

Operations count. For theoretical investigations (and some applications as well) it is desirable to know the formula requiring the fewest number of operations. Most formulas that SPIRAL generates, have, by construction, minimal known (or close to minimal) operation count, however, there are a few exceptions.

The first example is the class of Winograd algorithms [66] for small convolutions and small DFT sizes, which exhibit a large spread in operation counts. We have used SPIRAL to search this space for close to optimal solutions [67].

The second example arises when generating formulas using FMA instructions (Section IV-B), since known FMA algorithms for transforms are usually hand-derived and are only available for a few transforms, e.g., [51], [52], [68]. Using SPIRAL we obtain FMA code automatically; in doing this, we *found* most of the published algorithms automatically and generated many new ones for the transforms contained in SPIRAL [49].

Accuracy. For many applications, and in particular for those using fixed point code, numerical accuracy may be of greater importance than fast runtime. SPIRAL can be easily extended to search for accurate code, simply by adding a new cost function for accuracy.

Let A be a formula for the exact transform $T = A$. When implemented in k-bit fixed point arithmetic, this formula represents an approximation of the matrix T, i.e., $A_{k\text{-bit}} \approx T$. Thus, as a measure of accuracy of the formula A, we use

$$N_k(A) = \|A - A_{k\text{-bit}}\| \tag{49}$$

where $\|\cdot\|$ is a matrix norm. There are several norms possible; good choices are the matrix norms $\|\cdot\|_p$ that are *subordinate* to the vector norms $\|\cdot\|_p$ (see [69] for more details on norms). For fast evaluation, we choose the matrix norm $||M||_\infty = \max_i \left\{\sum_j |M_{i,j}|\right\}$. Given $N_k(A)$, input dependent error bounds can be derived by assuming an input x and setting $y = Ax$ (the exact result) and $\tilde{y} = A_{k\text{-bit}}x$ (the approximate result) to get

$$\|y - \tilde{y}\|_\infty \leq \|A - A_{k\text{-bit}}\|_\infty \|x\|_\infty = N_k(A)\|x\|_\infty.$$

Cost functions for multiplierless implementations. On platforms where multiplications are significantly more expensive than additions (e.g., ASICs, but possibly also fixed point only processors), *multiplierless* implementations of small transform kernels become viable candidates. "Multiplierless" means multiplications by constants are first represented in a fixed point format and then replaced by additions and shifts. For example, a constant multiplication $y = 5x$ is replaced by $y = (x \ll 2) + x$. Since DSP transforms are linear, i.e, consist exclusively of additions and multiplications by constants, this procedure produces a program consisting of additions and shifts only. The problem of finding the least addition implementation for one given constant is NP-hard [70]. We have reimplemented and extended the best known method [71] and included it as

a backend into SPIRAL to generate multiplierless code for a given formula and for user-specified constant precisions. Clearly, if these precisions are reduced, also the arithmetic cost (measured in additions) of the resulting implementation can be reduced. This leads to the following optimization problem: for a given transform T, find the formula A with the least number of additions that still satisfies a given accuracy threshold q with respect to a given accuracy measure N, i.e., $N(A) \leq q$.

We solve this problem automatically by using SPIRAL with the following high-level steps (see [72], [73] for more details).

- Generate a numerically accurate formula A for T as described in Section V-C.
- Find the best assignment of bit-widths to the occurring constants in A such that the threshold q holds. We have solved this problem using a greedy or an evolutionary search. The code was assumed to be completely unrolled so that the bit-widths could be chosen independently for each constant.

In this optimization problem, we have considered several target accuracy measures $N(\cdot)$ including numerical error measures such as (49), and also application driven measures. An example of the latter is the optimization of the IMDCT and the DCT of type 2 in an MP3 audio decoder [74]. Here, we chose the compliance test defined by the MP3 standard as the accuracy threshold. The evaluation was done by inserting the generated code into an actual MP3 implementation.

Performance modeling. SPIRAL generally uses empirical runtimes and searches to find efficient implementations. It is beneficial, both in terms of understanding and in reducing search times, to utilize performance models and analytically solve the optimization problems for which SPIRAL finds approximate solutions. Unfortunately, determining models that accurately predict performance is very difficult because modern processors have many interdependent features that affect performance. Nonetheless, it is possible to obtain analytical results for restricted classes of formulas using simplified performance models, see [63], [75]–[77] for results applicable to the WHT and the DFT. While these results do not accurately predict performance, they give insight into the search space and provide heuristics that may reduce the search time. Moreover, they can be used to explore performance on processors that are currently not available.

To illustrate the results obtained and their limitations, consider the factorization of the WHT in equation (26). The formula can be implemented with a triply nested loop, where the outer loop iterates over the product and the inner two loops implement the tensor product. The recursive expansions of $\mathbf{WHT}_{2^{k_i}}$ are computed in a similar fashion. Even though the current version of the SPIRAL system cannot produce code with recursive calls, it is still possible to implement this formula with a recursive function (see [27]), where the recursive expansions of $\mathbf{WHT}_{2^{k_i}}$ are computed with recursive calls to the function, and, in the base case, are computed with straight-line code generated by SPIRAL. In this implementation, different instantiations of the rule, corresponding to different decompositions $k = k_1 + \cdots + k_t$, will lead to different degrees of recursion and iteration, which implies that the code may have different numbers of machine instructions even though all algorithms have the exact same arithmetic cost. Let W_{2^k} be one such WHT formula and let $A(n)$ the number of times the recursive WHT procedure is called, $A_l(k)$ the number of times a base case of size 2^l (here it is assumed that $l < 8$) is executed, and $L_1(k)$, $L_2(k)$, and $L_3(k)$ the number of times the outer, middle, and inner loops are executed throughout all recursive calls. Then the total number of instructions required to execute W_{2^k} is equal to

$$\alpha A(k) + \sum_{l=1}^{8} \alpha_l A_l(k) + \sum_{i=1}^{3} \beta_i L_i(k) \tag{50}$$

where α is the number of instructions for the code in the compiled WHT procedure executed outside the loops, α_l is the number of instructions in the compiled straight-line code implementations of the base case of size l, and β_i, $i = 1, 2, 3$, is the number of instructions executed in the outermost, middle, and innermost loops in the compiled WHT procedure. These constants can be determined by examining the generated assembly code. Suppose $k = k_1 + \cdots + k_t$ is the decomposition of k corresponding to the factorization in (26); then the functions $A(k)$, $A_l(k)$, $L_i(k)$ satisfy recurrence relations of the form $F(k) = \sum_{i=0}^{t} \{2^{k-k_i} F(k_i) + f(i)\}$, where $f(i)$ depends on the function and is equal to $1/t$, 0, 1, 2^{k-k_i}, $2^{k_1+\cdots+k_{i-1}}$, respectively. While it is not possible to obtain a closed-form solution to all of the recurrences, it is possible to determine the formula with minimal instruction count, compute the expected value and variance for the number of instructions, and calculate the limiting distribution [78], [79].

The problem with these results is that the instruction count does not accurately predict performance on modern heavily pipelined superscalar processors with deep memory hierarchies, and that it is not clear how to extend the results to more general classes of formulas. While additional results have been obtained for cache misses, a general analytic solution has only been obtained for direct-mapped caches. Additional challenges must be overcome to obtain more general analytic results and to incorporate these insights into the SPIRAL system.

VI. Feedback Optimization: Search and Learning

One of the key features of the SPIRAL architecture (see Fig. 1) is the automated feedback loop, which enables SPIRAL to autonomously explore algorithm and implementation alternatives. Intuitively, this feedback loop provides SPIRAL with the "intelligence" that produces very fast code. Since the algorithm and implementation space is too large for an exhaustive enumeration and testing, this feedback loop needs to be controlled by empirical strategies that can find close to optimal solutions while visiting only a fraction

of the possible alternatives. These strategies have to take advantage of the particular structure of the algorithms.

We consider two fundamentally different strategies, as indicated already by the name of the Search/Learning block in Fig. 1.

- *Search* methods control the enumeration of algorithms and implementations at code generation time and guide this process toward finding a fast solution. Search is the method implemented in the current SPIRAL system.
- *Learning* methods operate differently. Before the actual code generation (offline), a set of random formulas including their runtimes are generated. This set constitutes the data from which the Learning block *learns*, i.e., extracts the knowledge of how a fast formula and implementation are constructed. At code generation time, this knowledge is used to generate the desired solution deterministically. We have implemented a prototype of this approach for a specific class of transforms including the DFT.

In the following, we explain the Search/Learning in greater detail.

A. Search

The goal of the Search block in SPIRAL (see Fig. 1) is to control the generation of the formulas and the selection of implementation options, which, in the current version, is the degree of unrolling. The search: 1) has to be able to modify previously generated formulas and 2) should be transform independent in the sense that adding a new transform and/or new rules requires no modification of the search. To achieve both goals, the search interfaces with the ruletree representation of formulas and not with the formula representation (see Section III).

The current SPIRAL system features five search methods.

- *Exhaustive search* enumerates all formulas in the formula space $\mathcal{F}$ and picks the best. Due to the large formula space $\mathcal{F}$, this is only feasible for very small transform sizes.
- *Random search* enumerates a fixed number of random formulas and picks the best. Since fast formulas are usually rare, this method is not very successful.
- *Dynamic programming* lends itself as a search method due to the recursive structure of the problem. For most problems, it is our method of choice.
- *Evolutionary search* uses an evolutionary algorithm to find the best implementation. This method is particularly useful in cases where dynamic programming fails.
- *Hill climbing* is a compromise between random search and evolutionary search and has proven to be inferior to the latter. See [44] for an explanation of this technique in the context of SPIRAL.

We explain dynamic programming and the evolutionary search in greater detail.

Dynamic programming (DP). The idea of DP is to recursively construct solutions of large problems from previously constructed solutions of smaller problems. DP requires a recursive problem structure and, hence, is perfectly suited for the domain of transform algorithms.

We have implemented the DP search in a straightforward way as follows. Given a transform T, we expand T *one step* using all applicable rules and rule instantiations (for parameterized rules). The result is a set $\{RT_k | k = 1, \ldots, m\}$ of m ruletrees of depth 1 (as (40)) or 0 (if the rule is a terminal rule). For each of these ruletrees RT_k the set of children $\{C_i | i = 1, \ldots, j_k\}$ (the C_i are again transforms) is extracted, and for each of these children C_i, DP is called recursively to return a ruletree RC_i, which is fully expanded. Inserting the ruletrees RC_i into RT_k (that means replacing C_i by RC_i in RT_k), for $i = 1, \ldots, j_k$, yields a fully expanded ruletree RT'_k for T. Finally the best (minimal cost) ruletree among the RT'_k is returned as the result for T.

To see how DP reduces the search space, consider a DFT of size 2^n and only the Cooley–Tukey rule (20). Using recurrences, one can show that the number of formulas is $O(4^n/n^{3/2})$ (the number of binary trees by using Stirling's formula, [80, pp. 388–389], whereas DP visits only $O(n^2)$.

The inherent assumption of DP is that the best code for a transform is independent of the context in which it is called. This assumption holds for the arithmetic cost (which implies that DP produces the optimal solution), but not for the runtime of transform algorithms. For example, the left smaller transform (child) in the DFT rule (20) is applied at a stride, which may cause cache thrashing and may impact the choice of the optimal formula. However, in practice, DP has proven to generate good code in reasonably short time [44] and, thus, is the default search method in the current version of SPIRAL.

Finally, we note that the vector extension of SPIRAL requires a special version of DP, which is motivated by the manipulated formula (46). As explained before, (46)] is already vectorizable. In particular, the occurring DFTs can be expanded arbitrarily, since their context is $\ldots \otimes I_\nu$, which ensures they are vectorizable [matching (42)]. To account for the conceptual difference between the first and the remaining expansions, we need a variant of DP, which we introduced in [54]

Evolutionary search. It is valuable to have another search method available to evaluate DP and overcome its possible shortcomings, particularly in view of the growing number of applications of SPIRAL (e.g., Sections III and V-C). Evolutionary search operates in a mode that is entirely different from the DP mode; it attempts to mimic the mechanics of evolution, which operates (and optimizes in a sense) through cross breeding, mutation, and selection [81].

For a given transform, the evolutionary search generates an initial population P_1 of a fixed size n of randomly selected ruletrees. Then, the population is increased using *cross breeding* and *mutation*. Cross breeding is implemented by swapping subtrees with the same root (transform) of two selected ruletrees in P_1 [see Fig. 6(a)]. Three different types of mutations are used: 1) regrow expands a selected node using a different subruletree; 2) copy copies a selected subruletree to a different node representing the same transform; and

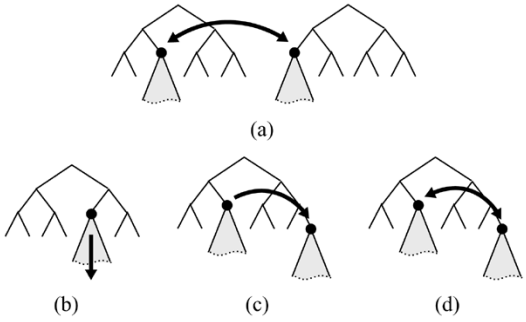


Fig. 6. Ruletree manipulation for the evolutionary search. (a) Cross breeding. (b)–(d) Three types of mutations: regrow, copy, and swap.

3) swap exchanges two subruletrees belonging to the same transform. See Fig. 6(b)–6(c) for an illustration. The trees that undergo cross breeding and mutation are randomly selected, and the number of those trees is a parameter. Finally, the increased population is shrunk to a size smaller than n by removing the slowest trees. Then the population is increased to the original size n by adding random trees to yield the population P_2. This process is repeated for a given number of iterations or until the best member of the population does not improve the minimization any further. For a more detailed discussion and evaluation of the evolutionary search, we refer to [44], [82].

The problem with evolutionary search (in general) is that it may converge to solutions that are only locally optimal.

B. Learning

Search becomes more difficult as the number of possible ruletrees increases. However, it is easy to collect a set of runtimes for random implementations of a given transform. This data could be used to learn how to construct a fast ruletree for that transform. Further, we have found that this knowledge can be applied to generate fast implementations of different sizes of a given transform, even when the knowledge was gathered from only a single transform size.

Our approach consists of two stages.

- *Modeling Performance of Individual Nodes.* The first step begins by collecting timing information for each individual node in a set of random ruletrees. From this data, we then learn how to construct a model that accurately predicts the runtimes for nodes in ruletrees. This effort requires a well-chosen set of features that describe a node and its context within the larger ruletree.
- *Generating Fast Implementations.* The second step uses the model developed in the first step to then generate ruletrees that have fast running times.

Our discussion will focus on the WHT and the DFT. For the WHT we consider only ruletrees based on rule (26) with the restriction $t = 2$ (two children); for the DFT we consider only ruletrees based on the Cooley–Tukey rule (20). Both rules have similar structure, in particular, for a DFT or a WHT of size n, and $n = km$, the left child $\mathbf{T}_k$ in both cases appears in a tensor product of the form $\mathbf{T}_k \otimes I_m$, which means $\mathbf{T}_k$ is computed m times at stride m. In the following, we call m the stride of the ruletree node $\mathbf{T}_k$. As a transform is expanded recursively, the strides accumulate, e.g., for $n = km$, $m = k'm'$, two applications of the rule lead to a left child with stride $m + m'$. The focus in this section are large transform sizes. Thus, to further restrict the algorithm space, we used SPIRAL to pregenerate straightline code implementations of WHTs and DFTs of sizes $2^1, \ldots, 2^7$. These are used as leaves in the ruletrees. This means that if the ruletree is generated, in each step either a rule is applied or, if the node is small enough, a leaf can be chosen to terminate.

Modeling Performance. It is possible to carefully time each individual node of a ruletree as it runs. The runtime for an internal node is calculated by subtracting off the runtimes of the subtrees under the node from the total runtime for the tree rooted at the given internal node. To allow our methods to learn across different transform sizes, we divide the actual runtimes by the size of the overall transform and learn on these values.

In order to model the runtimes for different nodes, we must define a set of features that describe nodes in ruletrees. To allow the modeling to generalize to previously unseen ruletrees, the features should not completely describe the ruletree in which the node is located. However, a single simple feature such as the node's size may not provide enough context to allow for an accurate model to be learned. Intuitively, our features are chosen to provide our method with the domain knowledge about the transform algorithms.

Clearly, the size of the transform at the given node is an important feature as the size indicates the amount of data that the node must process. The node's position in the ruletree is also an important factor in determining the node's runtime. This position often determines the stride at which the node accesses its input and output as well as the state of the cache when the node's computation begins. However, it is not as easy to capture a node's position in a ruletree as it is to capture its size.

A node's stride can be computed easily and provides information about the node's position in a ruletree and also about how the transform at this node accesses its input and output.

To provide more context, the size and stride of the parent of the given node can also be used as features. These features provide some information about how much data will be shared with siblings and how that data is laid out in memory. Further, for internal nodes the sizes and strides of the node's children and grandchildren may also be used. These features describe how the given node is initially split. If a node does not have a given parent, child, or grandchild, then the corresponding features are set to -1.

Knowing which leaf in the ruletree was computed prior to a given node may provide information about what data is in memory and its organization. Let the common parent be the first common node in the parent chains of both a given node and the last leaf computed prior to this node. The size and stride of this common parent actually provides the best information about memory prior to the given node beginning execution. The common parent's size indicates how much

Table 11
Error Rates for Predicting Runtimes for WHT Leaves

Pentium III

Binary No-2^1-Leaf

Size	Errors
2^{13}	13.0%
2^{14}	13.8%
2^{15}	15.8%
2^{16}	14.6%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	11.4%
2^{18}	12.9%
2^{19}	12.6%
2^{20}	12.7%

Sun UltraSparc IIi

Binary No-2^1-Leaf

Size	Errors
2^{13}	8.7%
2^{14}	8.7%
2^{15}	10.9%
2^{16}	7.3%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	16.5%
2^{18}	16.9%
2^{19}	18.9%
2^{20}	20.0%

Table 12
Error Rates for Predicting Runtimes for Entire WHT Ruletrees

Pentium III

Binary No-2^1-Leaf

Size	Errors
2^{13}	20.1%
2^{14}	22.6%
2^{15}	25.0%
2^{16}	18.1%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	14.4%
2^{18}	14.1%
2^{19}	12.5%
2^{20}	10.1%

Sun UltraSparc IIi

Binary No-2^1-Leaf

Size	Errors
2^{13}	23.5%
2^{14}	17.6%
2^{15}	25.8%
2^{16}	36.5%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	13.3%
2^{18}	15.2%
2^{19}	19.8%
2^{20}	21.2%

data has been recently accessed by the previous leaf and at what stride the data has been accessed.

Thus, we use the following features:

- size and stride of the given node;
- size and stride of the given node's parent;
- size and stride of each of the given node's children and grandchildren;
- size and stride of the given node's common parent.

For the WHT, all of the work is performed in the leaves with no work being done in the internal nodes, so the features for the children and grandchildren were excluded for the WHT since the leaves were the only interesting nodes to consider. However, internal nodes in an DFT ruletree do perform work; thus, the full set of features was used for the DFT.

Given these features for ruletree nodes, we can now use standard machine learning techniques to learn to predict runtimes for nodes. Our algorithm for a given transform is as follows:

1) Run a subset of ruletrees for the given transform, collecting runtimes for every node in the ruletree.
2) Divide each of these runtimes by the size of the overall transform.
3) Describe each of the nodes with the features outlined earlier.
4) Train a function approximation algorithm to predict for nodes the ratio of their runtime to the overall transform size.

We have used the regression tree learner RT4.0 [83] for a function approximation algorithm in the results presented here. Regression trees are similar to decision trees except that they can predict real valued outputs instead of just categories. However, any good function approximation method could have been used.

We trained two regression trees on data collected from running a random set of size 2^{16} WHT implementations, one from data for a Pentium III and one from data for a Sun UltraSparc IIi (later often referred to simply as Pentium and Sun). We also trained another regression tree on data collected from running a random set of size 2^{16} DFT implementations on Pentium. Specifically, we chose a random 10% of the nodes of all possible binary ruletrees with no leaves of size 2^1 to train our regression trees (we had previously found that the subset of binary ruletrees with no leaves of size 2^1 usually contains the fastest implementations).

To test the performance of our regression trees, we evaluated their predictions for ruletrees of sizes 2^{12} to 2^{20}. Unfortunately, we could not evaluate them against all possible ruletrees since collecting that many runtimes would take prohibitively long. Instead we timed subsets of ruletrees that previous experience has shown to contain the fastest ruletrees. Specifically, for the WHT, for sizes 2^{16} and smaller we used binary ruletrees with no leaves of size 2^1 and for larger sizes we used binary rightmost ruletrees (trees where every left child is a leaf) with no leaves of size 2^1. For the DFT, we were not certain that rightmost ruletrees were best; so we only evaluate up to size 2^{18} over all binary ruletrees with no leaves of size 2^1.

For each ruletree in our test set, we used the regression trees to predict the runtimes for each of the nodes in the ruletree, summing the results to produce a total predicted runtime for the ruletree. We evaluate the performance of our WHT regression trees both at predicting runtimes for individual nodes and for predicting runtimes for entire ruletrees. We report average percentage error over all nodes/ruletrees in our given test set, calculated as

$$\frac{1}{|\text{TestSet}|}\sum_{i\in\text{TestSet}}\frac{|a_i-p_i|}{a_i}$$

where a_i and p_i are the actual and predicted runtimes for node/ruletree i.

Table 11 presents the error rates for predicting runtimes for individual WHT leaves. In all cases, the error rate is never greater than 20%. This is good considering that the regression trees were trained only on data collected from running size 2^{16} WHT transforms.

Table 12 presents the error rates for predicting runtimes for entire WHT ruletrees. Not surprisingly, the results here are not as good as for individual leaves, but still good considering

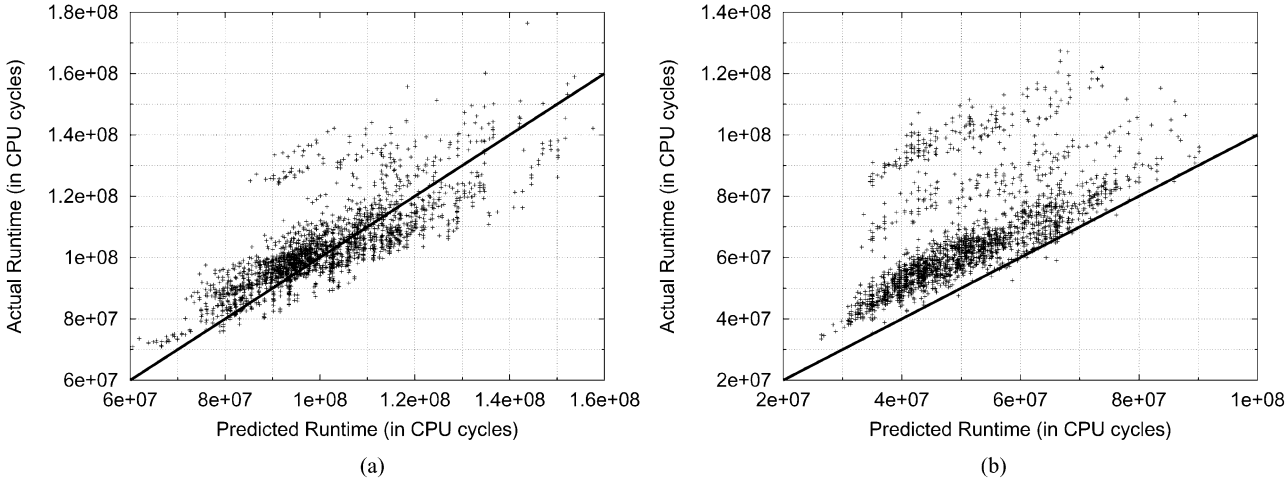


Fig. 7. Actual runtime versus predicted runtime for all binary rightmost $\mathrm{WHT}_{2^{19}}$ ruletrees with no leaves of size 2^1 on Pentium and Sun. The displayed line $y = x$ in both plots represents perfect prediction. (a) Pentium III. (b) Sun UltraSparc IIi.

Table 13
Error Rates for Predicting Runtimes for Entire DFT Ruletrees on Pentium

Size	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
Errors	19.3%	9.3%	10.7%	7.3%	5.0%	7.3%	7.9%

that different ruletrees can have runtimes that vary by a factor of 2–10.

Fortunately, the runtime predictor only needs to be able to order the runtimes of ruletrees correctly to aid in optimization. The exact runtime of a ruletree is not necessary; just a correct ordering of ruletrees is necessary to generate fast ruletrees. To evaluate this, we plotted the actual runtimes of ruletrees against their predicted runtimes. Fig. 7 shows plots for size 2^{19} WHT transforms (the plots for the other sizes look similar). Each dot in the scatter plots corresponds to one ruletree. The dot is placed vertically according to its actual runtime and horizontally according to the predicted runtime from the regression tree. The line $y = x$ is also plotted for reference. The plots show that as the actual runtimes decrease for ruletrees, so do their predicted runtimes. Further, the ruletrees that are predicted to be the fastest can be seen to also be the ruletrees with the fastest actual runtimes. Thus, the runtime predictors perform well at ordering ruletrees according to their actual runtimes.

Table 13 shows the error rates for predicting runtimes for entire DFT ruletrees running on Pentium. Except for size 2^{12}, the error rates here are quite excellent, especially considering that the learned regression tree was only trained on data of size 2^{16}. The scatter plots for DFTs look very similar to those for the WHT already displayed. They clearly show that the learned regression tree is ordering formulas correctly and that particularly the ruletrees with the fastest predicted runtimes actually have the fastest runtimes.

Generating Fast Implementations. While the previous work presents a way to accurately predict runtimes for WHT and DFT ruletrees, it still does not solve the problem of constructing fast ruletrees. At larger sizes, there are many possible ruletrees and it can be difficult to even enumerate all the ruletrees, let alone obtain a prediction for each one. We now describe a method for generating ruletrees that have fast runtimes.

Table 14
Algorithm for Computing Values of States

```
ComputeValues(State)
  if V(State) already memorized
    return V(State)
  Min = ∞
  if State can be a leaf
    Min = PredictedPerformance(State)
  for SetOfChildren in PossibleSetsOfChildren(State)
    Sum = 0
    for Child in SetOfChildren
      Sum += ComputeValues(Child)
    Sum += PredictedPerformance(State)
    if Sum < Min
      Min = Sum
  V(State) = Min
  return Min
```

Generation of ruletrees begins with a given transform and size for which a fast implementation is desired. We then need to choose a factorization of this transform, producing children for the root node. Recursively, we again choose children for each of the root node's children, and so on until we decide to leave a particular node as a leaf.

Our approach is to define a set of states encountered during the construction of fast ruletrees. We define a value function over these states and show how that value function can be quickly computed. We then show how to construct fast ruletrees given the computed value function.

In the previous modeling work, we designed a set of features that allowed for accurate prediction of runtimes of ruletree nodes. Thus, these features seemed ideal for describing our state space. During the construction of ruletrees, we describe nodes by their features and consider this to be the node's state. So it is possible for two nodes in different ruletrees to be considered the same state and for two nodes of the same transform and size to be considered different states.

We now define the optimal value function over this state space. For a given state, we consider all possible subtrees that

Table 15
Algorithm for Generating Fast Ruletrees

```
FastTrees(State)
  Trees = {}
  if State can be a leaf
    if V(State) == PredictedPerformance(State)
      Trees = { Leaf(State) }
  for RightChild in PossibleRightChildren(State)
    LeftChild = MatchingChild(State, RightChild)
    if V(LeftChild) + V(RightChild)
        + PredictedPerformance(State) == V(State)
      for RightSubtree in FastTrees(RightChild)
        for LeftSubtree in FastTrees(LeftChild)
          Trees = Trees ∪ { Node(LeftSubtree, RightSubtree) }
  return Trees
```

could be grown under that node along with the possibility of leaving the node as a leaf. We then define the value of this state to be the minimum sum of the predicted runtimes for each of the nodes in a subtree, taken over all possible subtrees. These predicted runtimes are determined by the regression trees trained in the previous section. Mathematically

$$V(\text{state}) = \min_{\text{subtrees}} \sum_{\text{node}\in\text{subtree}} \text{PredictedPerformance}(\text{node}).$$

Note that the state of a node indicates its children and grandchildren for the DFT while we excluded these features for the WHT. So for the DFT the minimum is really only taken over valid subtrees given the state.

We can rewrite this value function recursively. For a given state, we consider all possible one-level splittings of the current node along with the possibility of leaving the node as a leaf. The value of this state is then the minimum of the predicted performance of the current node plus the sum of the values of any immediate children of the node for the best splitting. That is

$$V(\text{state}) = \min_{\text{splittings}} \left(\text{PredictedPerformance}(\text{node}) + \sum_{\text{child}\in\text{splitting}} V(\text{child}) \right).$$

For the DFT, the state already describes the immediate children. However, the full state description of the children is not known, since it includes the grandchildren, i.e., the great-grandchildren of the original node. Thus, for the DFT, the minimum is actually taken over possible great-grandchildren of the given node.

This recursive formulation of the value function suggests using dynamic programming to efficiently compute the value function. Table 14 displays the dynamic programming algorithm for computing values of states. Again the algorithm needs to be slightly modified for the DFT where the state description includes its children. The outer "for" loop is actually computed over the possible great-grandchildren instead of just the children. It should also be noted that this dynamic programming is different from that presented earlier in the section on search (Section VI-A) in that this algorithm is considering states described by many features besides just a node's transform and size and that values are obtained from the learned regression trees. Due to the memoization of values of states, this algorithm is significantly subexhaustive since during an exhaustive search the same state would appear in many different ruletrees.

Now with a computed value function on all states, it is possible to generate fast ruletrees. Table 15 presents our algorithm for generating fast ruletrees, restricting to binary ruletrees for simplicity of presentation. For each possible set of children for a given node, the algorithm looks up their values. These values are added to the predicted performance of the current node and compared against the value function of the current state. If equal, we then generate the subtrees under the children recursively. Again for the DFT, the algorithm needs to be modified to loop over the possible great-grandchildren instead of the children.

Since our regression tree models are not perfect, we may wish to generate more than just the single ruletree with the fastest predicted runtime. If a small set of ruletrees were generated, we could then time all the generated ruletrees and choose the one with the fastest runtime. We have implemented an extended version of the FastTrees algorithm that allows for a tolerance and generates all ruletrees that are within that tolerance of the predicted optimal runtime.

Tables 16 and 17 show the results of generating fast WHT ruletrees for Pentium and for Sun respectively. To evaluate our methods, we again exhaust over subspaces of ruletrees known to contain fast implementations since it is impossible to obtain runtimes for all possible ruletrees in a reasonable amount of time. In both tables, the first column indicates the transform size. The second column shows how many ruletrees need to be generated before the fastest ruletree is generated. The third column indicates how much slower the first ruletree generated is compared to the fastest ruletree. Let G be the set of the first 100 ruletrees generated by our methods and let B be the set of the best 100 ruletrees found by exhaustive search. The fourth column displays the number of items in the intersection of G and B. Finally, the last column shows the rank of the first element in B not contained in G.

In all cases, the fastest ruletree for a given WHT transform size was generated in the first 50 formulas produced. This is excellent considering the huge space of possible ruletrees and the fact that this process only used runtime information

Table 16
Evaluation of Generation Method Using a WHT Runtime Predictor for a Pentium

Size	Generated ruletree number X is best known ruletree	First generated ruletree is $X\%$ slower than best known ruletree	Number of top 100 best known ruletrees in top 100 generated ruletrees	First best known ruletree not in top 100 generated ruletrees
2^{13}	5	3.4%	69	19
2^{14}	4	3.0%	63	19
2^{15}	3	2.1%	68	16
2^{16}	4	1.7%	63	18
2^{17}	5	0.1%	54	36
2^{18}	4	2.0%	60	24
2^{19}	1	0.0%	44	36
2^{20}	4	1.7%	64	24

Table 17
Evaluation of Generation Method Using a WHT Runtime Predictor for a Sun

Size	Generated ruletree number X is best known ruletree	First generated ruletree is $X\%$ slower than best known ruletree	Number of top 100 best known ruletrees in top 100 generated ruletrees	First best known ruletree not in top 100 generated ruletrees
2^{13}	14	77.7%	20	6
2^{14}	20	12.8%	70	24
2^{15}	1	0.0%	68	38
2^{16}	2	4.3%	70	20
2^{17}	7	18.0%	47	10
2^{18}	38	5.9%	46	7
2^{19}	17	3.3%	46	4
2^{20}	47	1.4%	52	4

Table 18
Evaluation of Generation Method Using DFT Runtime Predictors for Pentium

Size	Generated ruletree number X is best known ruletree	First generated ruletree is $X\%$ slower than best known ruletree
2^{12}	16	14.3%
2^{13}	1	0.0%
2^{14}	2	13.6%
2^{15}	1	0.0%
2^{16}	1	0.0%
2^{17}	82	3.6%
2^{18}	11	6.5%

gained by timing ruletrees of size 2^{16}. Except for a few cases on the Sun, the very first ruletree generated by our method had a runtime within 6% of the fastest runtime. Further, in all but one case, at least 40 of the 100 fastest ruletrees known to us were generated as one of the first 100 ruletrees. On occasion, the fourth fastest ruletree was not generated in the first 100 ruletrees.

Table 18 shows the results for generating fast DFT ruletrees on Pentium. The results are excellent with the fastest ruletree being generating usually within the first 20 and often as the very first ruletree. Further, the first ruletree to be generated had a runtime always within 15% of the runtime of the fastest formula.

In this section, we have described a method that automatically generates fast WHT and DFT ruletrees. To do this, we also presented a method that accurately predicts runtimes for ruletrees. More details and results can be found in [84]–[86].

VII. Experimental Results

In this section we present a selected set of experiments and performance benchmarks with SPIRAL's generated code. We remind the reader that in the SPIRAL lingo the expression "completely expanded formula," or simply "formula," means a transform algorithm.

We start with an overview of the presented experiments.

- *Performance spread.* We show the performance spread, with respect to runtime and other measures, within the formula space for a given transform.
- *Benchmarking: DFT.* We benchmark the runtime of SPIRAL generated DFT code (including fixed-point code) against the best available libraries.
- *Benchmarking: other transforms.* We benchmark SPIRAL generated code for other transforms: the DCT and the WHT.
- *Runtime studies of FIR filters and the DWT.* We compare different algorithmic choices for filters and the DWT.
- *Platform tuning.* We demonstrate the importance of platform tuning, i.e., the dependency of the best algorithm and code on the platform and the data type.
- *Compiler flags.* We show the impact of choosing compiler flags.
- *Parallel platforms.* We present prototypical results with adapting the WHT to an SMP platform.
- *Multiplierless code.* We show runtime experiments with generated multiplierless fixed-point DFT code.
- *Runtime of code generation.* We discuss the time it takes SPIRAL to generate code.

Table 19
Platforms Used for Experiments. "HT" Means Hyper Threading; L1 Cache Refers to the Data Cache. The Compilers Are: ICC (Intel C++ Compiler); GCC (GNU C Compiler); CC_R (IBM XL C Compiler, SMP Mode)

name	CPU	GHz	OS	caches	compiler	compiler flags
p4-3.0-win	Pentium 4 (HT)	3.0	WinXP	8 KB L1, 512 KB L2	icc 8.0	/QxKW /G7 /O3
p4-3.0-lin	Pentium 4 (HT)	3.0	Linux	8 KB L1, 512 KB L2	gcc 3.2.1	–O6 –fomit-frame-pointer –malign-double –fstrict-aliasing –mcpu=pentiumpro
p4-2.53-win	Pentium 4	2.53	Win 2000	8 KB L1, 512 KB L2	icc 6.0	/QxW /G7 /O3
p3-1.0-win	Pentium III	1.0	Win 2000	16 KB L1, 256 KB L2	icc 6.0	/QxW /G6 /O3
xeon-1.7-lin	Xeon	1.7	Linux	8 KB L1, 256 KB L2	gcc 3.2.1	–O6 –fomit-frame-pointer –malign-double –fstrict-aliasing –mcpu=pentiumpro
xp-1.73-win	AthlonXP 2100+	1.73	Win 2000	64 KB L1, 256 KB L2	icc 6.0	/QxW /G6 /O3
ibms80-0.45-aix	PowerPC RS64C (12 processors)	0.45	AIX	128 KB L1, 8 MB L2	cc_r 5.0.5	-qsmp=omp -O5 -q64
ipaq-0.4-lin	XScale PXA250 (IPAQ HP 3950)	0.4	Linux	32+2 KB L1	gcc 3.3.2	-O1 -fomit-frame-pointer -fstrict-aliasing -march=armv5te -mtune=xscale

The platforms we used for our experiments are shown in Table 19. For each platform, we provide the following: a descriptive mnemonic name, the most important microarchitectural information, and the compiler and compiler flags used. We used DP (dynamic programming) for all searches. For vector code we used the vector version of DP (see Section VI-A).

Performance spread. The first experiment investigates the spread in runtime as well as the spread with respect to other performance measures of different formulas generated by SPIRAL for the same transform on p4-3.0-lin.

In the first example, we consider a small transform, namely a $\mathbf{DCT\text{-}2}_{2^5}$, for which SPIRAL reports 1 639 236 012 different formulas. We select a random subset of 10 000 formulas and generate scalar code. By "random formula" we mean that a rule is chosen randomly at each step in the formula generation (note that this method is fast but selects ruletrees nonuniformly). Fig. 8(a) shows a histogram of the obtained runtimes, and Fig. 8(b) shows a histogram of the number of assembly instructions in the compiled C code. The spread of runtimes is approximately a factor of two, and the spread of the number of instructions is about 1.5, whereas the spread in arithmetic cost is less than 10% as shown in Fig. 8(c). The large spread in runtime and assembly instruction counts is surprising given that each implementation is high-quality code that underwent SPL and C compiler optimizations. Also, for transforms of this size and on this platform no cache problems arise. Conversion into FMA code (explained in Section IV-C) reduces the operations count [see Fig. 8(d)], but increases the spread to about 25%. This means that different formulas are differently well suited for FMA architectures. In Fig. 8(e) we plot runtime versus arithmetic cost. Surprisingly, the formulas with lowest arithmetic cost yield both slowest and fastest runtimes, which implies that arithmetic cost is not a predictor of runtime in this case. Finally, Fig. 8(f) shows the accuracy spread when the constants are cut to 8 bits; it is about a factor of ten with most formulas clustered within a factor of two.

In the second example, we show a runtime histogram for 20 000 random SPIRAL generated formulas for a large transform, namely, $\mathbf{DFT}_{2^{16}}$, using only the Cooley–Tukey rule (20) on p4-3.0-win. The formulas are implemented in scalar code [see Fig. 9(a)] and in vector code [see Fig. 9(b)]. The spread of runtimes in both cases is about a factor of five, with most formulas within a factor of three. The best 30% formulas are scarce. The plots show that, even after the extensive code optimizations performed by SPIRAL, the runtime performance of the implementation is still critically dependent on the chosen formula. Further, histogram Fig. 9(b) looks very much like a translation to the left (shorter runtime) of the histogram Fig. 9(a). This demonstrates that the vectorization approach in SPIRAL is quite general: although different formulas are differently well suited to vectorization, the performance of all tested 20 000 formulas, including the slowest, is improved by SPIRAL's vectorization.

Conclusion: performance spread. Although different formulas for one transform have a similar operation count [see Fig. 8(c)], their scalar or vector code implementations in SPIRAL have a significant spread in runtime (Figs. 8(a) and 9). This makes a strong case for the need of tuning implementations to platforms, including proper algorithm selection, as discussed in Section II. The same conclusion applies to other performance costs as illustrated by the significant spread in Fig. 8(d) for the FMA optimized arithmetic cost and in Fig. 8(f) for the accuracy performance cost.

Benchmarking: DFT. We first consider benchmarks of the code generated by SPIRAL for the DFT on p4-3.0-win against the best available DFT libraries including MKL 6.1 and IPP 4.0 (both Intel's vendor libraries), and FFTW 3.0.1. For most other transforms in SPIRAL, there are no such readily available high-quality implementations.

Fig. 10 shows the results for the $\mathbf{DFT}_{2^4}$–$\mathbf{DFT}_{2^{16}}$. The performance is given in pseudomegaflops computed as $5n\log_2(n)/(10^6 \times \text{runtime})$, which is somewhat larger than real megaflops, but preserves the runtime relations. This is important for comparison, since different implementations may have slightly different arithmetic cost. (Note that for all other transforms we use real megaflops.) The peak performance of p4-3.0-win is, for scalar code, 3 Gflops (single and double precision), and for vector code 12 Gflops (single

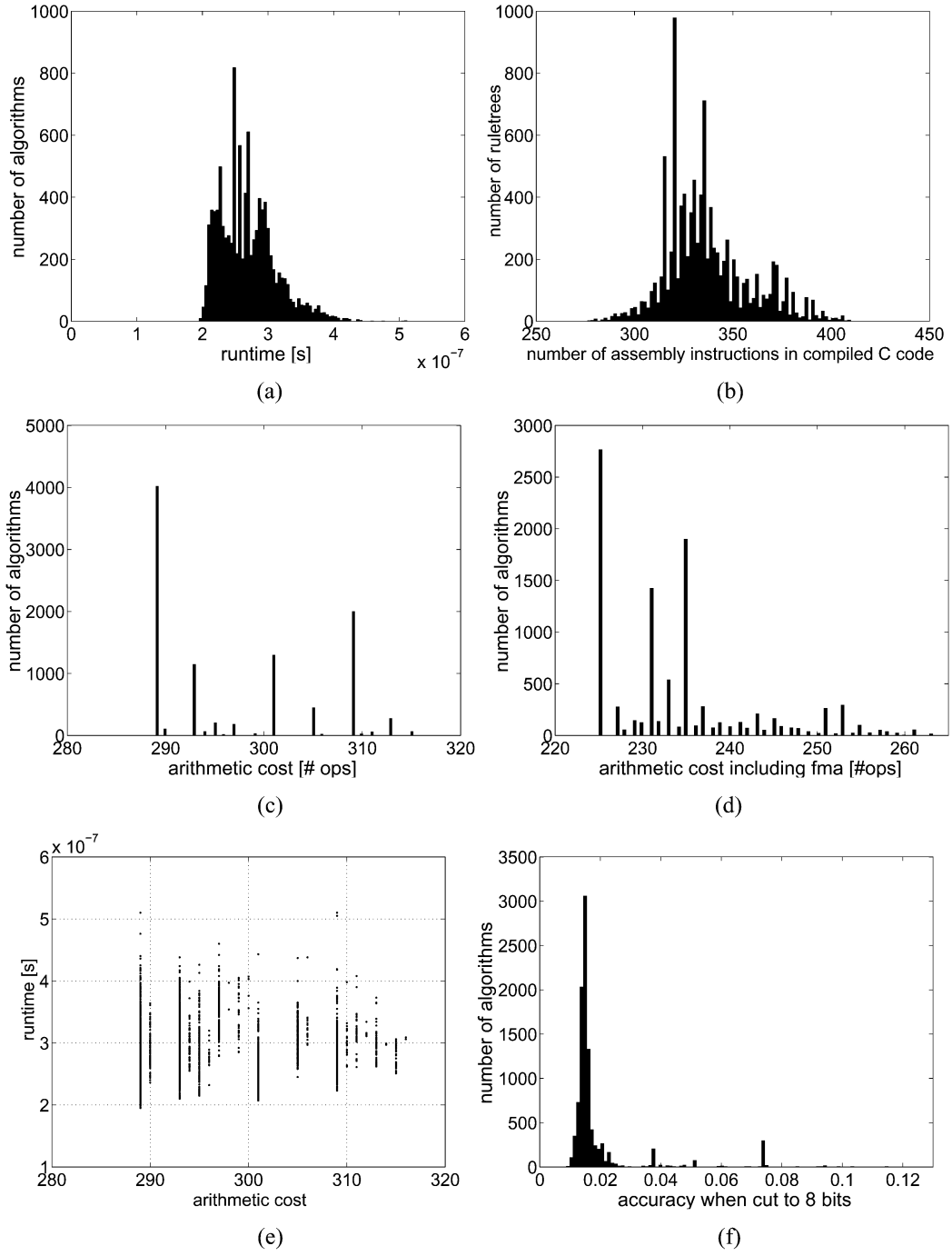


Fig. 8. Histograms of various data for 10 000 random fast formulas for a $\mathbf{DCT\text{-}2}_{32}$. (a) Runtime. (b) Number of assembly instructions in the compiled C code. (c) Arithmetic cost. (d) FMA optimized arithmetic cost. (e) Runtime versus arithmetic cost. (f) Accuracy when cut down to 8-bit fixed point; Platform: p4-3.0-lin.

precision) and 6 Gflops (double precision). The DFT is computed out of place with the exception of the IPP code and the Numerical Recipes code [87], which are computed inplace. In these figures, higher numbers correspond to better performance. Solid lines correspond to SPIRAL generated code, dotted lines to the Intel libraries, and dashed lines to FFTW and other libraries. We focus the discussion on Fig. 10(a), starting from the bottom up. The lowest line is the GNU library, which is a reimplementation of FFTPACK, a library that was frequently used a decade ago. The library is a reasonable C implementation but without any adaptation mechanism or use of vector instructions. The next two lines are FFTW 3.0.1 and SPIRAL generated scalar C code, which are about equal in performance. Considerably higher performance is achievable only by using vector instructions. The next line shows the speedup obtained through compiler vectorization, as enabled by a flag, used in tandem with SPIRAL. This is a fair evaluation of compiler vectorization as the Search block will find those formulas the compiler can handle best. The speedup is about 50%, obtained with

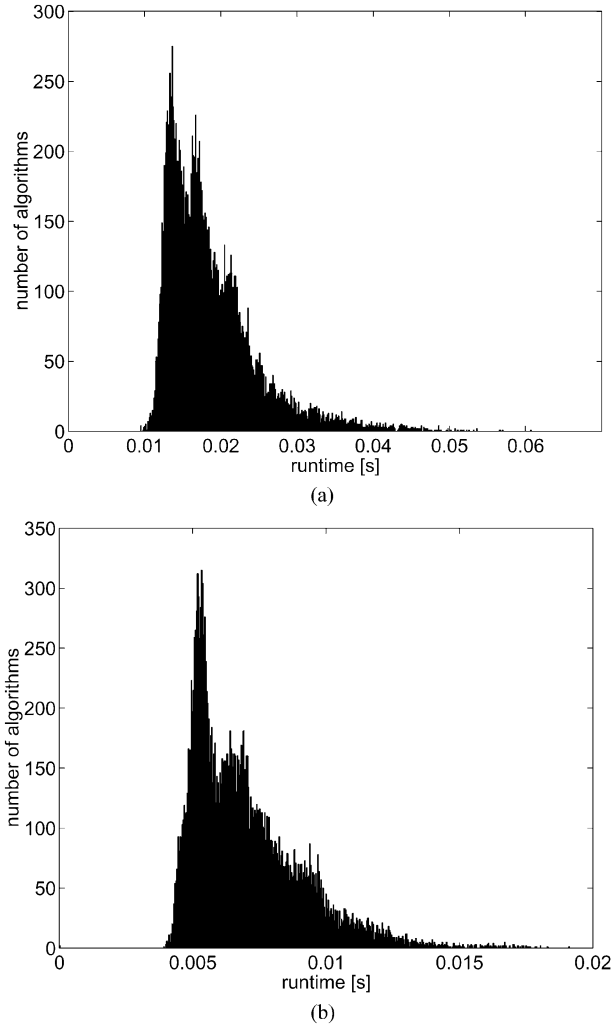


Fig. 9. Histogram of 20 000 random SPIRAL generated scalar and SSE vector implementations for a DFT of size 2^{16}. Platform: p4-3.0-win. (a) Scalar code (double precision). (b) SSE vector code (single precision).

no additional effort. We note that FFTW cannot be compiler vectorized due to its complex infrastructure. This 50% speedup is, however, only a fraction of the speedup achieved by the best possible vector code, which is about a factor of two faster, or a factor of three over the best scalar code. This performance is achieved by MKL, IPP, FFTW, and SPIRAL (the top four lines). We speculate on the reason for their relative performance.

- For small sizes, within L1 cache, SPIRAL code is best by a margin, most likely due to the combination of algorithm search, code level optimizations, and the simplest code structure.
- Outside L1 but inside L2 cache, the Intel libraries are fastest, most likely since the code is inplace and possibly due to optimizations that require microarchitectural information not freely available.
- For larger sizes, FFTW seems to hold up the best, due to a number of optimization specifically introduced for large sizes in FFTW 3.0 [18].

Similar observations can be made for double precision code; see Fig. 10(b).

Regarding cache effects, we mention that for single precision, approximately 32 B per complex vector entry are needed (input vector, output vector, constants and spill space) while for double precision 64 B are needed. Taking into account the Pentium 4's 8 KB of L1 data cache, this implies that FFTs of size 256 (single precision) and 128 (double precision) can be computed completely within L1 data cache. Similarly, the 512 KB L2 cache translates into sizes of 2^{14} (for single precision) and 2^{13} (for double precision), respectively.

Finally, we also consider implementations of the DFT on ipaq-0.4-lin, which provides only fixed point arithmetic. We compare the performance of SPIRAL's generated code with the IPP vendor library code for this platform. For most sizes, IPP fares considerably worse, see Fig. 11, which shows the (pseudo) Mflops achieved across a range of DFT sizes: 2^1 to 2^{12}.

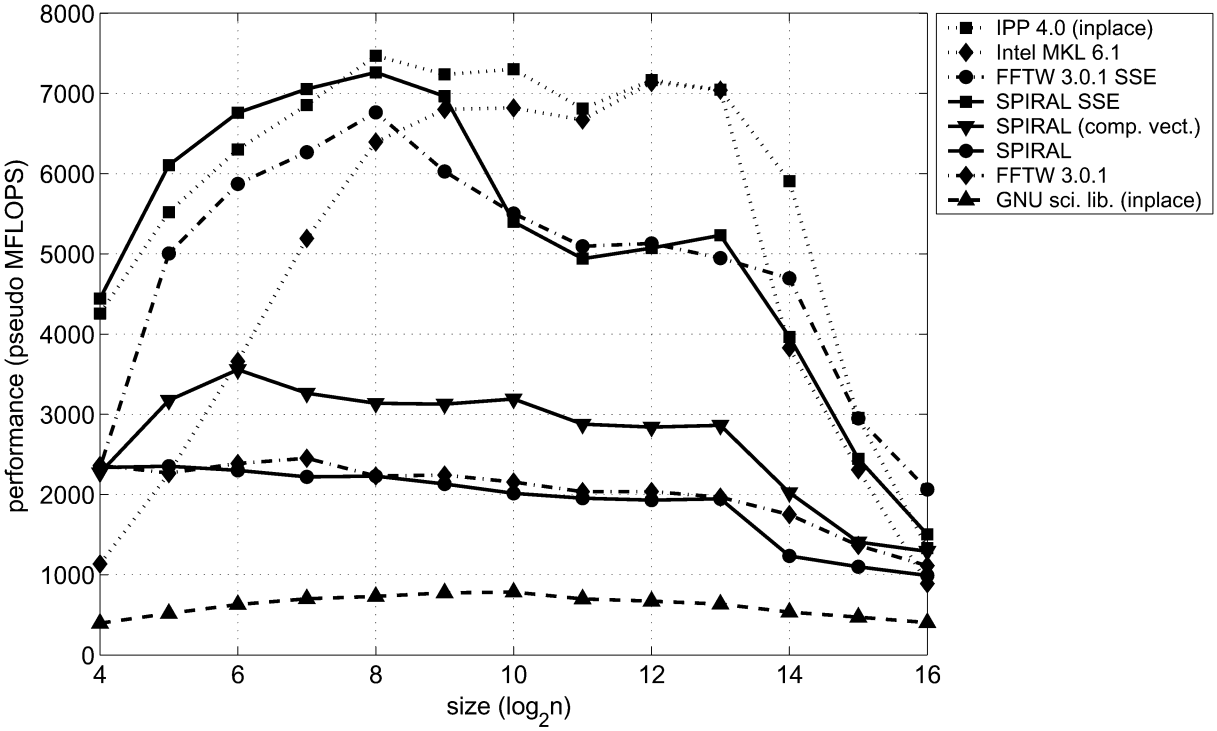
Conclusion: DFT benchmarking. For the DFT, SPIRAL scalar code is as competitive as the best code available. On p4-3.0-win, SPIRAL automatically generated vector code is faster by a factor of two to three compared to the scalar code, on par with IPP and MKL, Intel's hand-tuned vendor libraries. On ipaq-0.4-lin, SPIRAL generated code can be as much as four times faster than the IPP code.

Benchmarking: other transforms. We compare IPP to SPIRAL on p4-3.0-win for the DCT, type 2, in Fig. 12(a). Both for single and double precisions, the SPIRAL code is about a factor of two faster than the vendor library code, achieving up to 1500 Mflops (scalar code).

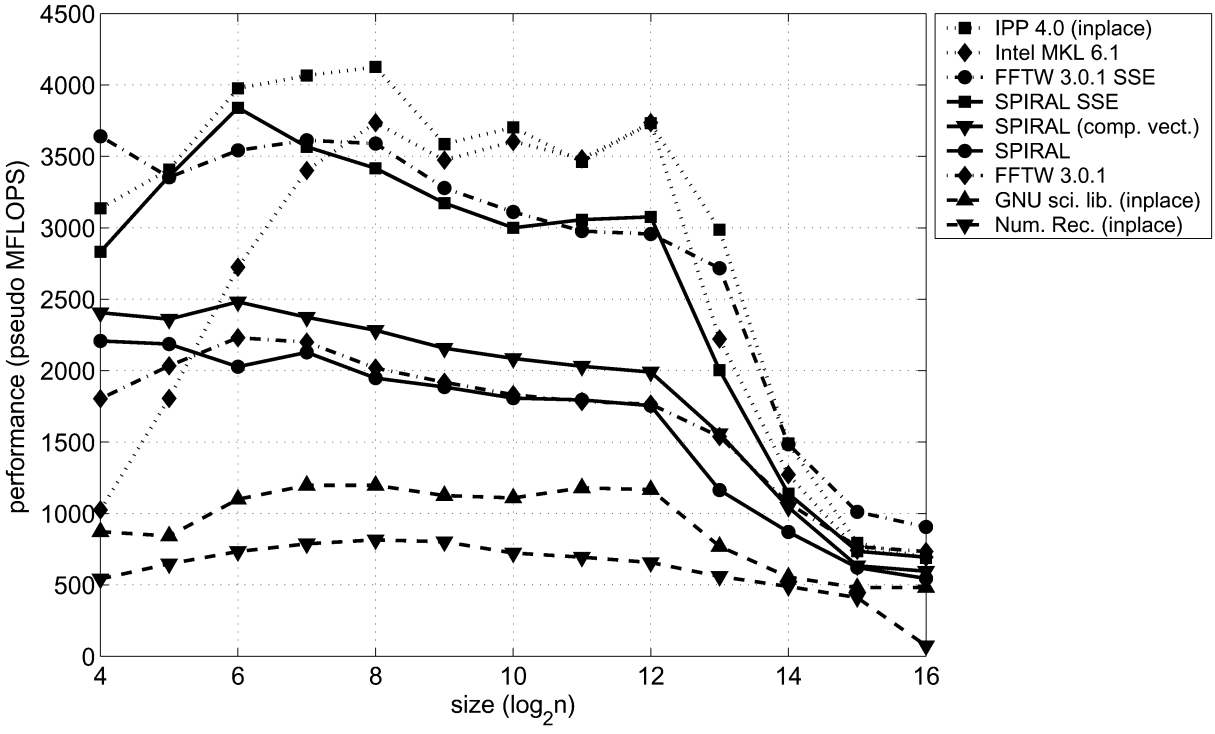
Fig. 12(b) and (c) study the performance of the corresponding 2D-DCT, which has the tensor product structure (19) that enables SPIRAL vectorization. Again we compare generated scalar code, compiler vectorized code, and SPIRAL vectorized code. Compiler vectorization fails for single precision, i.e., SSE [Fig. 12(b)], but yields a speedup for double precision, i.e., SSE2 [Fig. 12(c)]. SPIRAL generated vector code is clearly best in both cases and across all considered sizes. For SSE, up to 4500 Mflops and up to a factor of three speedup over scalar code are achieved.

We consider now the WHT, whose formulas have the simplest structure among all trigonometric transforms. Fig. 13(a) considers single precision and Fig. 13(b) double precision implementations, respectively. These figures show that, again, vectorization by SPIRAL produces efficient code, up to a factor of 2.5 and 1.5 faster than scalar code for single and double precision, respectively. Interestingly, vectorization of the SPIRAL code by the compiler is in this case also successful, with gains that are comparable to the gains achieved by SPIRAL vectorization.

Runtime studies of FIR filters and the DWT. Fig. 14(a) compares different SPIRAL generated scalar implementations of an FIR filter with 16 taps and input sizes varying in the range 2^1–2^{20} on xeon-1.7-lin. The plot shows runtimes normalized by the runtime of a base method. The base method is a straightforward implementation of the filter transform using overlap-add with block size 1; its performance is given by the top horizontal line at 1 and not



(a)



(b)

Fig. 10. FFT performance comparison (in pseudo Mflops) of the best available libraries. Platform: p4-3.0-win. (a) Single precision. (b) Double precision.

shown. In this figure, lower is better (meaning faster than the base method). The dashed line (squares) shows the relative runtime if only the overlap-add rule with arbitrary block sizes is enabled—a gain of about 85% over the base method. Further gains of 10%–20% are achieved if in addition the overlap-save rule and the blocking rule are enabled (triangles and bullets, respectively).

We consider now Fig. 14(b), which compares the effect of different rules on the DWT runtime performance. We choose the variant known as Daubechies 9–7 wavelet, enforce three different rules for the top-level expansion, with Mallat's rule being the baseline (horizontal line at 1), and compare the generated codes in each case. The polyphase rule (squares) is consistently inferior, whereas the lifting steps rule (triangles)

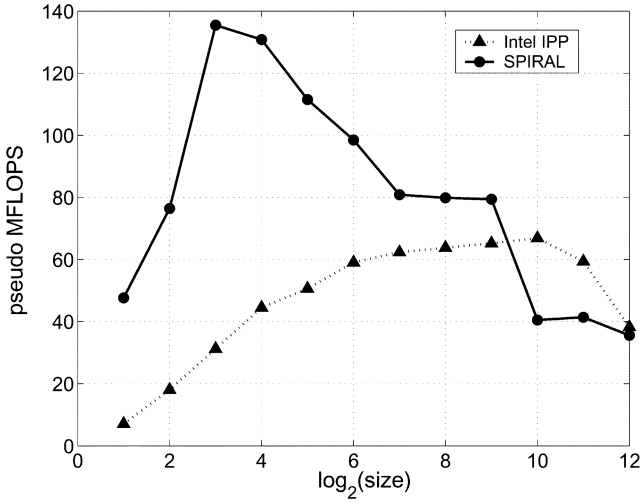


Fig. 11. Performance of SPIRAL generated fixed-point DFT code for sizes 2^1–2^{12}, on IPAQ versus Intel IPP 3.0. Platform: ipaq-0.4-lin.

improves over Mallat's rule for input sizes between 2^6 and 2^{12}. Beyond this size, Mallat's rule is clearly best as top-level rule. See [88] for a comprehensive evaluation of SPIRAL generated filter and DWT code.

Platform tuning. We now investigate the impact of performance tuning (see the table and the plot in Fig. 15). The table shows the (upper part of the) best ruletrees found for a DFT of size 2^{10} using only the Cooley–Tukey rule (20), for p4-2.53-win (single and double precision), p3-1.0-win (single precision), and xp-1.73-win (single precision). Each node in the trees is labeled with the exponent of the DFT size at this node; for example, the root node in all trees is labeled by 10, the exponent of the size of the transform 2^{10}. Most of the 12 ruletrees in this table are different from each other, meaning that SPIRAL finds different trees when searching for the best tuned formula for a given machine. Particularly worth noting is the difference between the balanced ruletrees found by SPIRAL for p3-1.0-win and xp-1.73-win, and the unbalanced ruletrees found for p4-2.53-win.

The plot on the right of Fig. 15 further explores the effect of tuning the implementation of $\mathbf{DFT}_{2^{10}}$: how does an implementation $\widehat{\mathbf{I}}(\mathbf{P}_1)$ tuned to a given platform $\mathbf{P}_1$ perform on another target platform $\mathbf{P}_2$? In particular, is $\widehat{\mathbf{I}}(\mathbf{P}_1)$ still tuned to the target platform $\mathbf{P}_2$? The answer is no as we explain next.

For DFT sizes $2^5, \ldots, 2^{13}$ we use SPIRAL to generate the best code for five different combinations of platforms and data types: p4-2.53-win SSE, p4-2.53-win SSE2, xp-1.73-win SSE, p3-1.0-win SSE, and p4-2.53-win float. Then, we generate SSE code for each of the obtained formulas and run it on p4-2.53-win. The slowdown factor compared to the code tuned to p4-2.53-win SSE is shown in the plot in Fig. 15 (i.e., higher is worse in this plot).

First, we observe that, as expected, the best code is the one tuned for p4-2.53-win SSE (bottom line equal to one). Beyond that, we focus on two special cases.

- *Same platform, different data type.* The best algorithm generated for p4-2.53-win SSE2, when implemented in SSE, performs up to 320% slower than the tuned implementation for p4-2.53-win SSE. The reason for this large gap is the different vector length of SSE2 and SSE (2 versus 4), which requires very different algorithm structures.
- *Same data type, different platform.* Code generated for p3-1.0-win SSE and run on the binary compatible p4-2.53-win SSE performs up to 50% slower than the SSE code tuned for p4-2.53-win. This is a very good example of the loss in performance when porting code to newer generation platforms. SPIRAL regenerates the code and overcomes this problem.

Compiler flags. In all prior experiments, we have always used a predefined and fixed combination of C compiler flags to compile the SPIRAL generated code (see Table 19). Assessing the effects on performance of compiler options is difficult, because: 1) there are many different options (the extreme case is gcc 3.3 with a total of more than 500 different documented flags, more than 60 of which are related to optimization); 2) different options can interact and/or conflict with each other in nontrivial ways; 3) the best options usually depend on the program being compiled. In SPIRAL, we have not yet addressed this problem; in fact, for gcc, SPIRAL uses the same optimization options as FFTW by default.

In the absence of clear guidelines, choosing the right set of compiler flags from the large set of possibilities poses another optimization problem that can be solved by a heuristic search. Analysis of Compiler Options via Evolutionary Algorithm (ACOVEA) [89] is an open-source project that uses an evolutionary algorithm to find the best compiler options for a given C program.

We apply ACOVEA to SPIRAL generated code for the DCT, type 2, of sizes $2^1, \ldots, 2^6$ on p4-3.0-lin. First, we generate the best (scalar) implementations using the default configuration (denoted by "gcc -O3" in the plot; the complete set of flags is in Table 19). Second, we retime the obtained implementations with a lower level of optimization (denoted by "gcc -O1," in reality "-O1 -fomit-frame-pointer -malign-double -march = pentium4"), and also with the Intel Compiler 8.0 (denoted by "icc /O3," the options were "/O3 /tpp7"). Finally, we run the ACOVEA evolutionary search for gcc compiler flags for each implementation. The results are shown in Fig. 16(a), which displays the speedup compared to "gcc -O1" (higher is better) for each of the 6 DCT codes. All sets of flags found by ACOVEA include at least "-O1 -march = pentium4." This justifies our choice of "gcc -O1" as the baseline. Note that "gcc -O3" is always slower than "gcc -O1," which means that some of the more advanced optimizations can make the code slower. In summary, ACOVEA gives an additional speedup ranging from 8% to 15% for the relevant larger DCT sizes (≥ 8) in this experiment.

The plot in Fig. 16(b) was also produced with the help of ACOVEA. Instead of performing an evolutionary search, we create an initial random population of 2000 compiler flag combinations, each of them again including at least "-O1 -march = pentium4," and produce a runtime histogram for

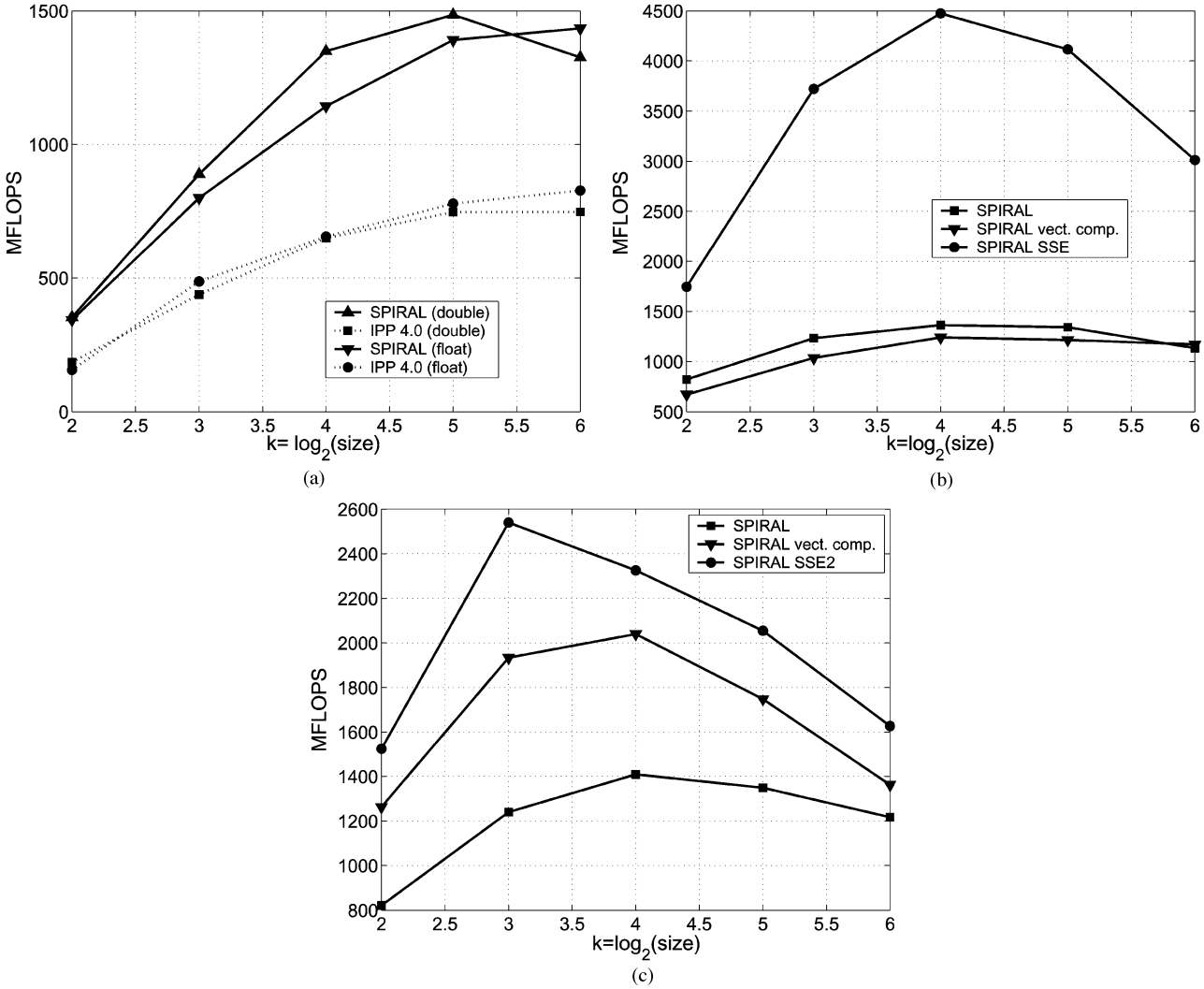


Fig. 12. (a) Comparing the performance (in Mflops) of SPIRAL generated code and IPP for a DCT-2 of size 2^k, $2 \leq k \leq 6$ for single and double precision. (b) and (c) 2D-DCT float and double precision: scalar SPIRAL code, scalar SPIRAL code compiler vectorized, and SPIRAL vector code. Platform: p4-3.0-win.

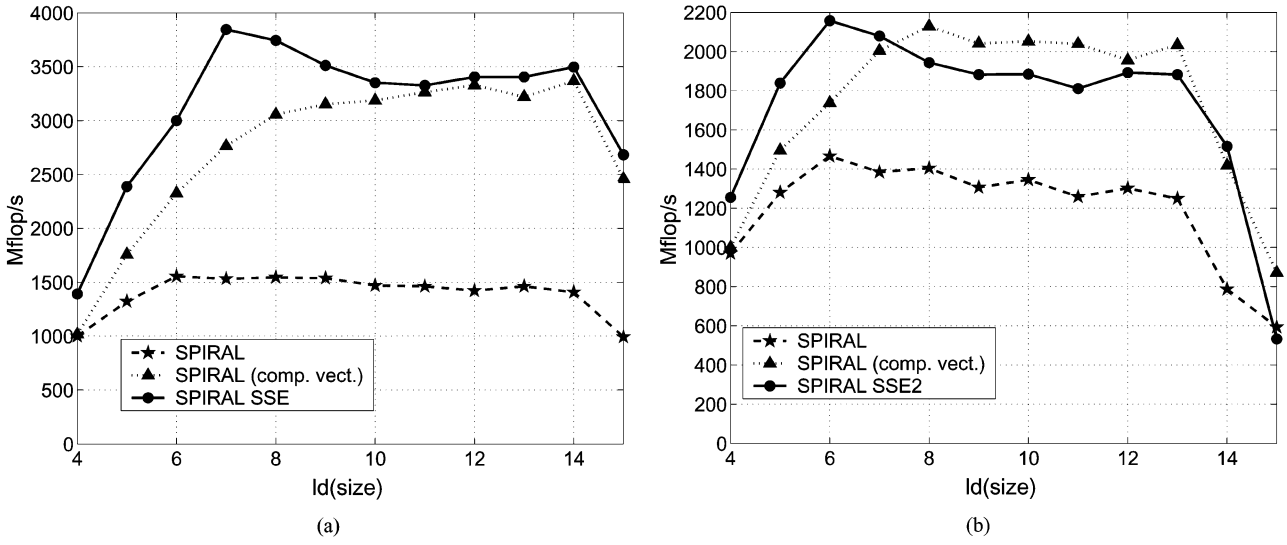


Fig. 13. WHT performance (in Mflops) of SPIRAL generated scalar code, compiler vectorized code, and vector code for: (a) single and (b) double precision. Platform: p4-3.0-win.

the $\mathbf{DCT\text{-}2}_{32}$ implementation generated in the previous experiment. The spread in runtimes of more than a factor of three demonstrates the big impact of the choice of compiler flags. The best compiler options in this histogram produce

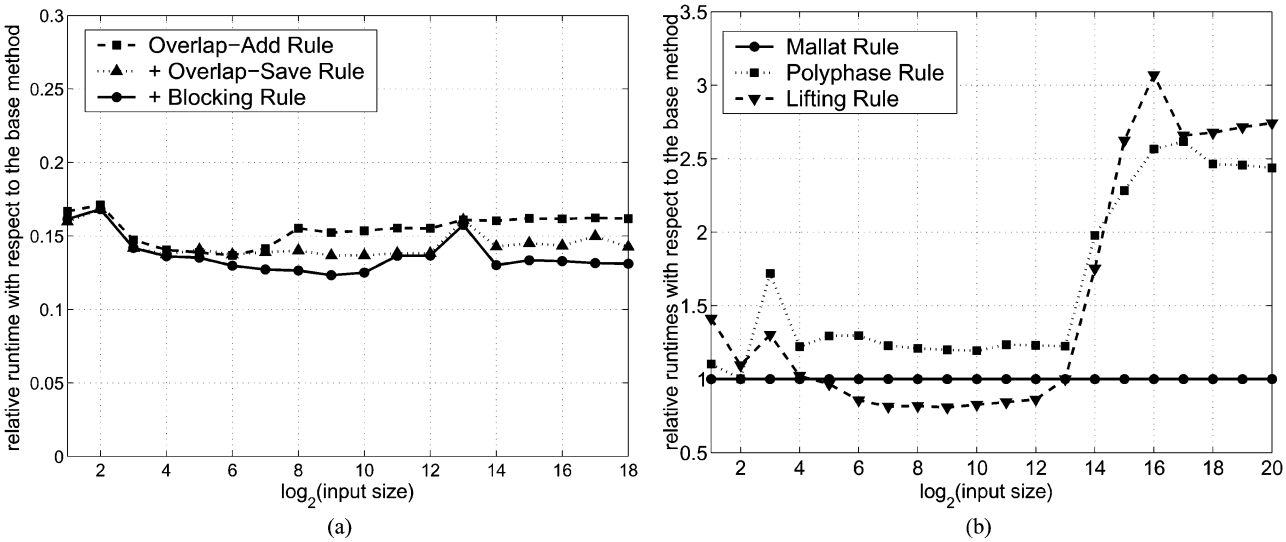
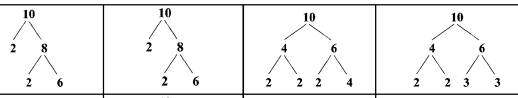
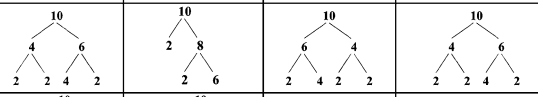
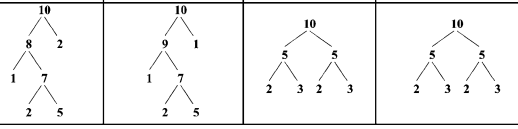


Fig. 14. (a) Runtime comparison of generated filter code (16 taps) found with increasing sets of rules enabled, normalized by the straightforward implementation. (b) Runtime comparison of the best found DWT implementation for three different choices of the uppermost rule, normalized by Mallat's rule. Platform: xeon-1.7-lin.

	P4 (single)	P4 (double)	PIII (single)	Athlon XP (single)
scalar code				
scalar code compiler vect.				
vector code single = SSE double = SSE2				

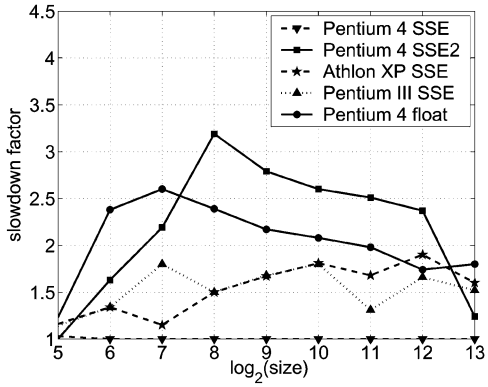


Fig. 15. Left: The best found DFT formulas for $n = 2^{10}$, represented as breakdown trees; right: crosstiming of best DFT ruletree, sizes $2^5, \ldots, 2^{13}$, generated for various platforms, implemented and measured on Pentium 4 using SSE. Platforms: p4-2.53-win, p3-1.0-win, xp-1.73-win.

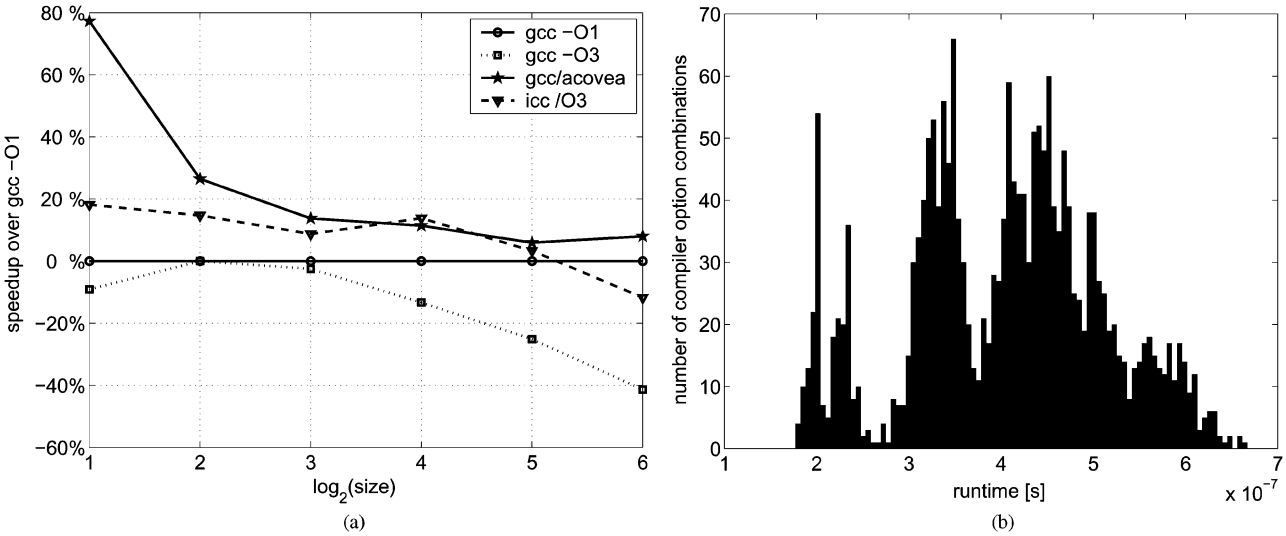


Fig. 16. Analysis of the impact of the choice of compiler flags using ACOVEA. (a) Improvement from compiler options search for **DCT-2** of sizes $2^1, \ldots, 2^6$. (b) Histogram of 2000 random compiler flags combinations for the best found implementation for $\mathbf{DCT\text{-}2}_{32}$.

a runtime (in seconds) of about $1.8 \cdot 10^{-7}$, whereas the best flags found by ACOVEA in the previous experiment produce $1.67 \cdot 10^{-7}$.

Parallel platforms. Section IV-F showed how SPIRAL could be used to generate parallel code and showed a family of shared-memory and distributed-memory parallel

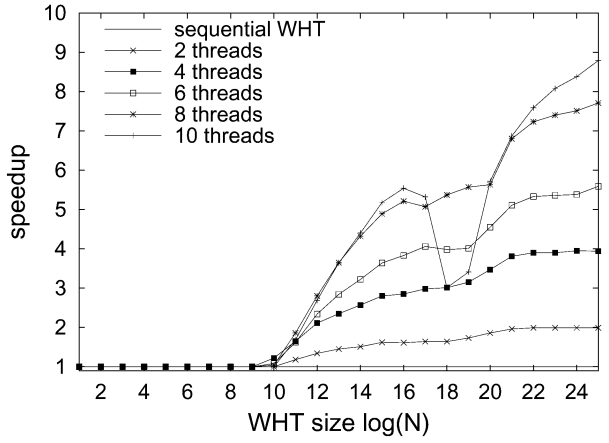


Fig. 17. Speedup for parallel code generated for $\mathbf{WHT}_{2^k}$, $1 \leq k \leq 24$ for up to ten threads. Platform: ibms80-0.45-aix with 12 processors.

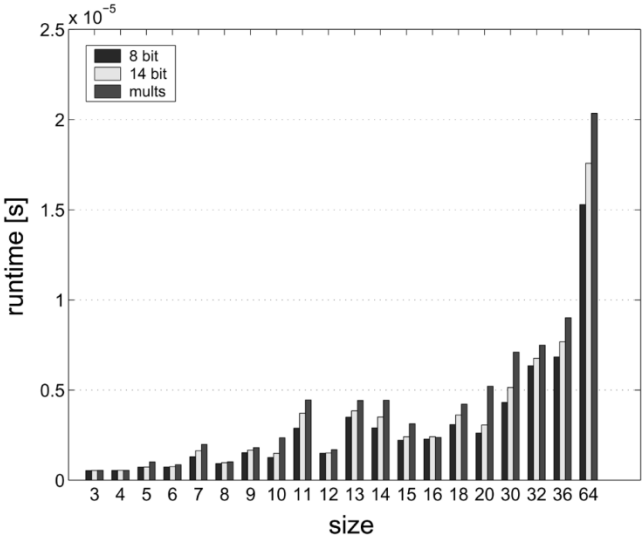


Fig. 18. Runtime performance (lower is better) of various DFTs of sizes between 3 and 64. For each size, the rightmost, middle, and leftmost bar shows (fixed point) code using multiplications and 14-bit and 8-bit multiplierless code, respectively. Platform: ipaq-0.4-lin.

algorithms for the WHT. Fig. 17 considers the WHT sizes $2^1, \ldots, 2^{24}$ and shows the speedup obtained with the generated routines. Speedup is computed for each number of threads as the ratio of the best sequential algorithm/implementation found compared to the best parallel algorithm/implementation found. We used dynamic programming in each case to automatically optimize granularity, load balance, cache utilization, and the selection of appropriately optimized sequential code. The platform is a 12 processor shared-memory multiprocessor platform ibms80-0.45-aix [90].

Fig. 17 shows that, for up to ten threads, nearly linear speedup is obtained for large transform size and parallelization is found to be beneficial for transforms as small as 2^{10}. The performance reported here is better than that reported in [59], due to searching through additional schedules and using loop interleaving [23] to reduce cache misses and false sharing. A straightforward parallelization method leads to far inferior performance. For example, for ten threads, only a factor of about three is achieved this way; a parallelizing compiler fares even worse than that. These results are not shown; please refer to [59] for more details. In summary, even for as simple a transform as the WHT, search through a relevant algorithm space is crucial to obtain the optimal performance.

Multiplierless code. SPIRAL can generate multiplierless code (see Section V). This is important for platforms that feature a fixed point processor such as the IPAQ and showcases a unique advantage of SPIRAL, as we are not aware of other multiplierless high-performance libraries. In a multiplierless implementation, a lower accuracy approximation of the constants leads to fewer additions and, thus, potentially faster runtime. This effect is shown in Fig. 18 for DFTs of various sizes, $3 \leq n \leq 64$, implemented in each case using either multiplications or additions and shifts with the constants approximated to 14 or 8 bits, respectively. Note that the code has to be unrolled to allow for this technique. The figure shows an improvement of up to 10% and 20%, respectively, for the 14-bit and 8-bit constant multiplierless code.

Runtime of code generation. SPIRAL requires only compile-time adaptation; thus, at runtime, no time is spent in further optimizing the code. Depending on the optimization strategy, the problem size, and the timer used, the optimization may take from the order of seconds to the order of hours. For instance, the generation of a scalar DFT library for two-powers up to 2^{20} is done in 20–30 min on a Pentium 4, while the corresponding vector code generation takes on the order of hours. Problem sizes around 64 are optimized within a few minutes. Note that SPIRAL generates code entirely from scratch, i.e., no code or code fragments for any transform are already pregenerated or handcoded in SPIRAL. In this respect, SPIRAL is similar to ATLAS with roughly similar code generation times. Compared to FFTW, SPIRAL needs longer to produce optimized code. However, in FFTW, real code generation (i.e., from scratch) is done only for small transform sizes and for unrolled code. These codelets (in FFTW lingo) are pregenerated and distributed with the package. Further, the codelet generation is deterministic, i.e., produces the same result independently of the machine. The optimization for larger FFT sizes in FFTW is done at runtime by determining, through dynamic programming, the best recursion strategy among those supported by FFTW. The available recursions are built into the rather complex infrastructure of FFTW. For example, for a one-dimensional DFT of composite size and in SPIRAL lingo, these recursion strategies are all the right-most ruletrees based on the Cooley–Tukey breakdown rule (20), where the left leaf is a codelet. Restricting the DFT computation to this restricted class of algorithms is a decision based on the experience of the FFTW developers. In SPIRAL, the candidate algorithms are deliberately as little constrained as possible, leaving the selection entirely to the system.

Conclusions. We draw the following main conclusions from our experiments.

- For any given transform, even for a small size, there is a large number of alternative formulas with a large spread in code quality, even after applying various code optimizations (Figs. 8 and 9).
- The difference in runtime between a "reasonable" implementation and the best possible can be an order of magnitude (e.g., a factor of ten in Fig. 10(a) between the GNU library and the IPP/FFTW/ SPIRAL code).
- Compiler vectorization is limited to code of very simple structure (e.g., Fig. 13), but fails to produce competitive code for more complex dataflows, e.g., Figs. 10 and 12(b) and (c). SPIRAL overcomes this problem through manipulations at the mathematical formula level; all other vector libraries involve hand coding.
- The performance of SPIRAL generated code is comparable with the performance of the best available library code.

VIII. Limitations of SPIRAL; Ongoing and Future Work

SPIRAL is an ongoing project and continues to increase in scope with respect to the transforms included, the types of code generated, and the optimization strategies included. We give a brief overview of the limitations of the current SPIRAL system and the ongoing and planned future work to resolve them.

- As we explained before, SPIRAL is currently restricted to discrete linear signal transforms. As a longer term effort we just started to research the applicability of SPIRAL-like approaches to other classes of mathematical algorithms from signal processing, communication, and cryptography. Clearly, the current system makes heavy use of the particular structure of transform algorithms in all of its components. However, most mathematical algorithms do possess structure, which, at least in principle, could be exploited to develop a SPIRAL-like code generator following the approach in Section II-A. Questions that need to be answered for a given algorithm domain then include the following.
  - How to develop a declarative structural representation of the relevant algorithms?
  - How to generate alternative algorithms and how to translate these algorithms into code?
  - How to formalize algorithm level optimizations as rewriting rules?
  - How to search the algorithm space with reasonable effort?
- Currently, SPIRAL can only generate code for one specific instance of a transform, e.g., for a transform of fixed size. This is desirable in applications where only a few sizes are needed which can be generated and bundled into a lightweight library. For applications with frequently changing input size, a package is preferable, which implements a transform for all, or a large number of sizes. To achieve this, recursive code needs to be generated that represents the breakdown rules, which is ongoing research. Once this is achieved, our goal is to generate entire packages, similar to FFTW for the DFT, on demand from scratch.
- The current vectorization framework can handle a large class of transforms, but only those whose algorithms are built from tensor products to a large extent. In this case, as we have shown, a small set of manipulation rules is sufficient to produce good code. We are currently working on extending the class of vectorizable transforms, e.g., to include large DCTs and wavelets. To achieve this, we will identify the necessary formula manipulation rules and include them into SPIRAL. With a large manipulation rule database ensuring convergence and uniqueness of the result (confluence) also becomes a problem. To ensure these properties, we will need a more rigorous approach based on the theory of rewriting systems [42].
- Similarly, and with an analogous strategy, we are in the process of extending SPIRAL's code generation capabilities for parallel platforms. These extensions are currently still in the prototype stage.
- Besides vector code, current platforms provide other potentially performance enhancing features, such as hyperthreading (Pentium 4) or prefetch instructions. Hyperthreading can be exploited by generating code with explicit threads, which was the previous goal; we aim to explicitly generate prefetch instructions through a combination of formula manipulation and loop analysis on the code level [91].
- For some applications it is desirable to compute a transform inplace, i.e., with the input and output vector residing in the same memory location. SPIRAL currently only generates out-of-place code. We aim to generate inplace code directly after a formula level only analysis.
- SPIRAL can generate fixed point code, but the decision for the chosen range and precision, i.e., the fixed-point format, has to be provided by the user. Clearly, the necessary range depends on the range of the input values. We are currently developing a backend [92] that chooses the optimal fixed point format once the input range is specified. The format can be chosen globally, or locally for each temporary variable to enhance precision.
- To date, the learning in SPIRAL is restricted to the selection of WHT ruletrees and DFT ruletrees based on the Cooley–Tukey rule. An important direction in our research is to extend the learning framework to learn and control a broader scope of transforms and to encompass more degrees of freedoms in the code generation.
- For many transforms, in particular the DFT, there are many different variants that differ only by the chosen scaling or assumptions on input properties such as symmetries. Most packages provide only a small number of

these variants due to the considerable hand-coding effort. In SPIRAL many of these variants can be handled by just including the specification and one or several rules. We are in the process of extending SPIRAL in this direction.

- We are just in the process of finishing an improved redesign of the SPIRAL system with considerably increased modularity to enable all the above extensions with reasonable effort. The possibility of extending SPIRAL, e.g., by inserting a backend code optimization module, or by connecting it to an architecture simulator, has led to its occasional use in class projects in algorithm, compiler, and architecture courses. The vertical integration of all stages of software development in SPIRAL allows the students to study the complex interaction of algorithms mathematics, compiler technology, and microarchitecture at hand of an important class of applications.
- Finally, as a longer term research effort and leaving the scope of this paper and this special issue, we have started to develop a SPIRAL-like generator for hardware designs of transforms for FPGAs or ASIC's.

IX. Conclusion

We presented SPIRAL, a code generation system for DSP transforms. Like a human expert in both DSP mathematics and code tuning, SPIRAL autonomously explores algorithm and implementation choices, optimizes at the algorithmic and at the code level, and exploits platform-specific features to create the best implementation for a given computer. Further, SPIRAL can be extended and adapted to generate code for new transforms, to exploit platform-specific special instructions, and to optimize for various performance metrics. We have shown that SPIRAL generated code can compete with, and sometimes even outperform the best available handwritten code. SPIRAL's approach provides performance portability across platforms and facilitates porting the entire transform domain across time.

The main ideas behind SPIRAL are to formulate the problem of code generation and tuning of transforms as an optimization problem over a relevant set of implementations. The implementation set is structured using a domain-specific language that allows the computer representation, generation, and optimization of algorithms and corresponding code. The platform-specific optimization is solved through an empirical feedback-driven exploration of the algorithm and implementation space. The exploration is guided by search and learning methods that exploit the structure of the domain.

While the current version of SPIRAL is restricted to transforms, we believe that its framework is more generally applicable and may provide ideas how to create the next generation of more "intelligent" software tools that push the limits of automation far beyond of what is currently possible and that may, at some point in the future, free humans from programming numerical kernels altogether.

Acknowledgment

The authors would like to thank L. Auslander for their early discussions on the automatic implementation of the DFT and other transforms. The authors would also like to thank A. Tsao for teaming them up. Further, the authors acknowledge the many interactions with A. Tsao, D. Healy, D. Cochran, and more recently with F. Darema, during the development of SPIRAL.

References

[1] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2003, pp. 63–76.

[2] T. Kisuki, P. Knijnenburg, and M. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proc. Parallel Architectures and Compilation Techniques (PACT)*, 2000, pp. 237–246.

[3] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, "Automatic program transformations for virtual memory computers," in *Proc. Nat. Computer Conf.*, 1979, pp. 969–974.

[4] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, 1981, pp. 207–218.

[5] F. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 1–30.

[6] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 1991, pp. 30–44.

[7] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric multi-level blocking," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 1997, pp. 346–357.

[8] K. Kennedy and R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA: Morgan Kaufmann, 2001.

[9] R. Metzger and Z. Wen, *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization*. Cambridge, MA: MIT Press, 2000.

[10] D. Barthou, P. Feautrier, and X. Redon, "On the equivalence of two systems of affine recurrence equations," in *Lecture Notes in Computer Science, Euro-Par 2002*. Heidelberg, Germany: Springer-Verlag, 2002, vol. 2400, pp. 309–313.

[11] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Comput.*, vol. 27, no. 1–2, pp. 3–35, 2001.

[12] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proc. IEEE*, vol. 93, no. 2, pp. 293–312, Feb. 2005.

[13] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[14] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "A comparison of empirical and model-driven optimization," *Proc. IEEE*, vol. 93, no. 2, pp. 358–386, Feb. 2005.

[15] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, 2004.

[16] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. IEEE*, vol. 93, no. 2, pp. 276–292, Feb. 2005.

[17] G. Baumgartner, D. Bernholdt, D. Cociovora, R. Harrison, M. Nooijen, J. Ramanujan, and P. Sadayappan, "A performance optimization framework for compilation of tensor contraction expressions into parallel programs," in *Proc. Int. Workshop High-Level Parallel Programming Models and Supportive Environments [Held in Conjunction With IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)]*, 2002, pp. 106–114.

[18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[19] ——, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 3, 1998, pp. 1381–1384. [Online]. Available: http://www.fftw.org.

[20] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 1999, pp. 169–180.

[21] D. Mirković and S. L. Johnsson, "Automatic performance tuning in the UHFFT library," in *Lecture Notes in Computer Science, Computational Science—ICCS 2001*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2073, pp. 71–80.

[22] S. Egner, "Zur Algorithmischen Zerlegungstheorie Linearer Transformationen Mit Symmetrie (On the algorithmic decomposition theory of linear transforms with symmetry)," Ph.D. dissertation, Institut für Informatik, Univ. Karlsruhe, Karlsruhe, Germany, 1997.

[23] K. S. Gatlin and L. Carter, "Faster FFT's via architecture cognizance," in *Proc. Parallel Architectures and Compilation Techniques (PACT)*, 2000, pp. 249–260.

[24] ——, "Architecture-cognizant divide and conquer algorithms," presented at the Conf. Supercomputing, Portland, OR, 1999.

[25] D. H. Bailey, "Unfavorable strides in cache memory systems," *Sci. Program.*, vol. 4, pp. 53–58, 1995.

[26] N. Park, D. Kang, K. Bondalapati, and V. K. Prasanna, "Dynamic data layouts for cache-conscious factorization of DFT," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2000, pp. 693–701.

[27] J. Johnson and M. Püschel, "In search for the optimal Walsh–Hadamard transform," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, 2000, pp. 3347–3350.

[28] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge, "Parallel VSIPL++: An open standard software library for high-performance parallel signal processing," *Proc. IEEE*, vol. 93, no. 2, pp. 313–330, Feb. 2005.

[29] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney, "A comparative study of static and profile-based heuristics for inlining," in *Proc. ACM SIGPLAN Workshop Dynamic and Adaptive Compilation and Optimization*, 2000, pp. 52–64.

[30] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. M. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Softw. Pract. Exper.*, vol. 22, no. 5, pp. 349–369, 1992.

[31] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1999.

[32] "Information Technology—JPEG 2000 Image Coding System—Part 1: Core Coding System," Int. Org. Standardization/Int. Electrotech. Comm., ISO/IEC 15 444-1 : 2000.

[33] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *Circuits, Syst., Signal Process.*, vol. 9, no. 4, pp. 449–500, 1990.

[34] C. Van Loan, *Computational Framework of the Fast Fourier Transform*. Philadelphia, PA: SIAM, 1992.

[35] G. E. Révész, *Introduction to Formal Languages*. New York: McGraw-Hill, 1983.

[36] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transforms and Convolution*, 2nd ed. New York: Springer-Verlag, 1997.

[37] M. Püschel, "Cooley–Tukey FFT like algorithms for the DCT," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2, 2003, pp. 501–504.

[38] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[39] G. Strang and T. Nguyen, *Wavelets and Filter Banks*. Reading, MA: Addison-Wesley, 1998.

[40] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Anal. Appl.*, vol. 4, no. 3, pp. 247–269, 1998.

[41] A. Graham, *Kronecker Products and Matrix Calculus With Applications*. New York: Wiley, 1981.

[42] N. Dershowitz and D. A. Plaisted, "Rewriting," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. New York: Elsevier, 2001, vol. 1, ch. 9, pp. 535–610.

[43] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura, "Fast automatic generation of DSP algorithms," in *Lecture Notes in Computer Science, Computational Science—ICCS 2001*. Heidelberg, Germany: Springer—Verlag, 2001, vol. 2073, pp. 97–106.

[44] 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," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 21–45, 2004.

[45] (1997) GAP—Groups, algorithms, and programming. GAP Team, Univ. St. Andrews, St. Andrews, U.K.. [Online]. Available: http://www-gap.dcs.st-and.ac.uk/~gap/

[46] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[47] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2001, pp. 298–308.

[48] N. Rizzolo and D. Padua, "Hilo: High level optimization of FFTs," presented at the Workshop Languages and Compilers for Parallel Computing (LCPC), West Lafayette, IN, 2004.

[49] Y. Voronenko and M. Püschel, "Automatic generation of implementations for DSP transforms on fused multiply-add architectures," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, 2004, pp. V-101–V-104.

[50] C. W. Fraser, D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code-generator generator," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 3, pp. 213–226, 1992.

[51] E. Linzer and E. Feig, "Implementation of efficient FFT algorithms on fused multiply-add architectures," *IEEE Trans. Signal Process.*, vol. 41, no. 1, p. 93, Jan. 1993.

[52] C. Lu, "Implementation of multiply-add FFT algorithms for complex and real data sequences," in *Proc. Int. Symp. Circuits and Systems (ISCAS)*, vol. 1, 1991, pp. 480–483.

[53] F. Franchetti and M. Püschel, "A SIMD vectorizing compiler for digital signal processing algorithms," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2002, pp. 20–26.

[54] ——, "Short vector code generation for the discrete Fourier transform," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2003, pp. 58–67.

[55] F. Franchetti, "Performance portable short vector transforms," Ph.D. dissertation, Inst. Appl. Math. Numer.Anal., Vienna Univ. Technol., 2003.

[56] F. Franchetti, S. Kral, J. Lorenz, and C. Ueberhuber, "Efficient utilization of SIMD extensions," *Proc. IEEE*, vol. 93, no. 2, pp. 409–425, Feb. 2005.

[57] R. E. J. Hoeflinger, Z. Li, and D. Padua, "Experience in the automatic parallelization of four perfect benchmark programs," in *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*, vol. 589. Heidelberg, Germany, 1992, pp. 65–83.

[58] R. E. J. Hoeflinger and D. Padua, "On the automatic parallelization of the perfect benchmarks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 1, pp. 5–23, Jan. 1998.

[59] K. Chen and J. R. Johnson, "A prototypical self-optimizing package for parallel implementation of fast signal transforms," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2002, pp. 58–63.

[60] ——, "A self-adapting distributed memory package for fast signal transforms," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2004, p. 44.

[61] (1998) OpenMP C and C++ Application Program Interface, Version 1.0. OpenMP. [Online]. Available: http://www.openmp.org

[62] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI: The Complete Reference*, 2nd ed. Cambridge, MA: MIT Press, 1998.

[63] P. Kumhom, "Design, optimization, and implementation of a universal FFT processor," Ph.D. dissertation, Dept. Elect. Comput. Eng., Drexel Univ., Philadelphia, PA, 2001.
[64] F. Ergün, "Testing multivariate linear functions: Overcoming the generator bottleneck," in *Proc. ACM Symp. Theory of Computing (STOC)*, vol. 2, 1995, pp. 407–416.
[65] J. Johnson, M. Püschel, and Y. Voronenko, "Verification of linear programs," presented at the Int. Symp. Symbolic and Algebraic Computation (ISSAC), London, ON, Canada, 2001.
[66] S. Winograd, *Arithmetic Complexity of Computations*, ser. CBMS-NSF Regional Conf. Ser. Appl. Math. Philadelphia, PA: SIAM, 1980.
[67] J. R. Johnson and A. F. Breitzman, "Automatic derivation and implementation of fast convolution algorithms," *J. Symbol. Comput.*, vol. 37, no. 2, pp. 261–293, 2004.
[68] E. Linzer and E. Feig, "New scaled DCT algorithms for fused multiply/add architectures," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 3, 1991, pp. 2201–2204.
[69] N. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA: SIAM, 2002.
[70] P. R. Cappello and K. Steiglitz, "Some complexity issues in digital signal processing," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-32, no. 5, pp. 1037–1041, Oct. 1984.
[71] O. Gustafsson, A. Dempster, and L. Wanhammar, "Extended results for minimum-adder constant integer multipliers," in *IEEE Int. Symp. Circuits and Systems*, vol. 1, 2002, pp. I-73–I-76.
[72] A. C. Zelinski, M. Püschel, S. Misra, and J. C. Hoe, "Automatic cost minimization for multiplierless implementations of discrete signal transforms," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, 2004, pp. V-221–V-224.
[73] M. Püschel, A. Zelinski, and J. C. Hoe, "Custom-optimized multiplierless implementations of DSP algorithms," presented at the Int. Conf. Computer Aided Design (ICCAD), San Jose, CA, 2004.
[74] "Information technology-coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbits/s," Int. Org. Standardization/Int. Electrotech. Comm., ISO/IEC 11 172, 1995.
[75] H.-J. Huang, "Performance analysis of an adaptive algorithm for the Walsh–Hadamard transform," M.S. thesis, Dept. Comput. Sci., Drexel Univ., Philadelphia, PA, 2002.
[76] M. Furis, "Cache miss analysis of Walsh–Hadamard transform algorithms," M.S. thesis, Dept. Comput. Sci., Drexel Univ., Philadelphia, PA, 2003.
[77] A. Parekh and J. R. Johnson, "Dataflow analysis of the FFT," Dept. Comput. Sci., Drexel Univ., Philadelphia, PA, Tech. Rep. DU-CS-2004-01, 2004.
[78] J. Johnson, P. Hitczenko, and H.-J. Huang, "Distribution of a class of divide and conquer recurrences arising from the computation of the Walsh–Hadamard transform," presented at the 3rd Colloq. Mathematics and Computer Science: Algorithms, Trees, Combinatorics and Probabilities, Vienna, Austria, 2004.
[79] P. Hitczenko, H.-J. Huang, and J. R. Johnson, "Distribution of a class of divide and conquer recurrences arising from the computation of the Walsh–Hadamard transform," *Theor. Comput. Sci.*, 2003, submitted for publication.
[80] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997, vol. 1.
[81] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[82] B. Singer and M. Veloso, "Stochastic search for signal processing algorithm optimization," *Proc. Supercomputing*, 2001.
[83] L. Torgo, "Inductive learning of tree-based regression models," Ph.D. dissertation, Dept. Comput. Sci., Faculty Sci., Univ. Porto, Porto, Portugal, 1999.
[84] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *J. Mach. Learn. Res.*, vol. 3, pp. 887–919, 2002.
[85] ——, "Learning to generate fast signal processing implementations," in *Proc. Int. Conf. Machine Learning*, 2001, pp. 529–536.
[86] B. Singer and M. M. Veloso, "Automating the modeling and optimization of the performance of signal transforms," *IEEE Trans. Signal Process.*, vol. 50, no. 8, pp. 2003–2014, Aug. 2002.
[87] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 1992.
[88] Ph.D. dissertation, Dept. Elect. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA.
[89] S. R. Ladd. (2004) ACOVEA: Analysis of Compiler Options via Evolutionary Algorithm. [Online]. Available: http://www.coyotegulch.com/acovea/
[90] RS/6000 Enterprise Server Model S80, Technology and Architecture. IBM. [Online]. Available: http://www.rs6000.ibm.com/resource/technology/s80techarch.html
[91] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 62–73.
[92] L. J. Chang, I. Hong, Y. Voronenko, and M. Püschel, "Adaptive mapping of linear DSP algorithms to fixed-point arithmetic," presented at the Workshop High Performance Embedded Computing (HPEC), Lexington, MA, 2004.

Markus Püschel (Member, IEEE) received the Diploma (M.Sc.) degree in mathematics and the Ph.D. degree in computer science from the University of Karlsruhe, Karlsruhe, Germany, in 1995 and 1998, respectively.

From 1998 to 1999, he was a Postdoctoral Researcher in the Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA. Since 2000, he has held a Research Faculty position in the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA. He was a Guest Editor of the *Journal of Symbolic Computation*. His research interests include scientific computing, compilers, applied mathematics and algebra, and signal processing theory/software/hardware. More details can be found at http://www.ece.cmu.edu/~pueschel.

Dr. Püschel is on the Editorial Board of the IEEE SIGNAL PROCESSING LETTERS and was a Guest Editor of the PROCEEDINGS OF THE IEEE.

José M. F. Moura (Fellow, IEEE) received the engenheiro electrotécnico degree from Instituto Superior Técnico (IST), Lisbon, Portugal, in 1969 and the M.Sc., E.E., and D.Sc. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, in 1973, 1973, and 1975, respectively.

He has been a Professor of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA, since 1986. He held visiting faculty appointments with MIT (1984–1986 and 1999–2000) and was on the faculty of IST (1975–1984). His research interests include statistical and algebraic signal and image processing and digital communications. He has published over 270 technical journal and conference papers, is the coeditor of two books, and holds six U.S. patents. He currently serves on the Editorial Board of the *ACM Transactions on Sensor Networks* (2004-).

Dr. Moura is a Corresponding Member of the Academy of Sciences of Portugal (Section of Sciences). He was awarded the 2003 IEEE Signal Processing Society Meritorious Service Award and the IEEE Millennium Medal. He has served the IEEE in several positions, including Vice-President for Publications for the IEEE Signal Processing Society (SPS) (2000–2002), Chair of the IEEE TAB Transactions Committee (2002–2003), *Editor-in-Chief* for the IEEE TRANSACTIONS ON SIGNAL PROCESSING (1975–1999), and Interim Editor-in-Chief for the IEEE SIGNAL PROCESSING LETTERS (December 2001–May 2002). He currently serves on the Editorial Boards of the PROCEEDINGS OF THE IEEE (2000-) and the *IEEE Signal Processing Magazine* (2003-).

Jeremy R. Johnson (Member, IEEE) received the B.A. degree in mathematics from the University of Wisconsin, Madison, in 1985, the M.S. degree in computer science from the University of Delaware, Newark, in 1988, and the Ph.D. degree in computer science from Ohio State University, Columbus, in 1991.

He is Professor and Department Head of Computer Science at Drexel University, Philadelphia, PA, with a joint appointment in Electrical and Computer Engineering. He is on the Editorial Board of *Applicable Algebra in Engineering, Communication and Computing* and has served as a Guest Editor for the *Journal of Symbolic Computation*. His research interests include algebraic algorithms, computer algebra systems, problem-solving environments, programming languages and compilers, high-performance computing, hardware generation, and automated performance tuning.

David Padua (Fellow, IEEE) received the Ph.D. degree in computer science from University of Illinois, Urbana-Champaign, in 1980.

He is a Professor of Computer Science at the University of Illinois, Urbana-Champaign, where he has been a faculty member since 1985. At Illinois, he has been Associate Director of the Center for Supercomputing Research and Development, a member of the Science Steering Committee of the Center for Simulation of Advanced Rockets, Vice-Chair of the College of Engineering Executive Committee, and a member of the Campus Research Board. His areas of interest include compilers, machine organization, and parallel computing. He has published more than 130 papers in those areas. He has served as Editor-in-Chief of the *International Journal of Parallel Programming* (IJPP). He is a Member of the Editorial Boards of the *Journal of Parallel and Distributed Computing* and IJPP.

Prof. Padua has served as a Program Committee Member, Program Chair, or General Chair for more than 40 conferences and workshops. He served on the Editorial Board of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. He is currently Steering Committee Chair of ACM SIGPLAN's Principles and Practice of Parallel Programming.

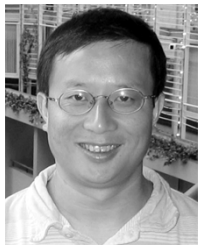
Manuela M. Veloso received the B.S. degree in electrical engineering and the M.Sc. degree in electrical and computer engineering from the Instituto Superior Técnico, Lisbon, Portugal, in 1980 and 1984, respectively, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1992.

She is Professor of Computer Science at Carnegie Mellon University. She researches in the area of artificial intelligence with focus on planning, control learning, and execution for single and multirobot teams. Her algorithms address uncertain, dynamic, and adversarial environments. Prof. Veloso has developed teams of robot soccer agents, which have been RoboCup world champions several times. She investigates learning approaches to a variety of control problems, in particular the performance optimization of algorithm implementations, and plan recognition in complex data sets.

Prof. Veloso is a Fellow of the American Association of Artificial Intelligence. She is Vice President of the RoboCup International Federation. She was awarded an NSF Career Award in 1995 and the Allen Newell Medal for Excellence in Research in 1997. She is Program Cochair of 2005 National Conference on Artificial Intelligence and the Program Chair of the 2007 International Joint Conference on Artificial Intelligence.

Bryan W. Singer was born in Indiana in 1974. He received the B.S. degree in computer engineering from Purdue University, West Lafayette, IN, in 1997 and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 2001.

His research interests include machine learning and automatic performance tuning.



Jianxin Xiong received the B.E. and M.E. degrees in computer science from Tsinghua University, Beijing, China, in 1992 and 1996, respectively, and the Ph.D. degree in computer science from the University of Illinois, Urbana-Champaign, in 2001.

From 1996 to 1998, he was a Lecturer at Tsinghua University. From 2001 to 2002 he worked as a Compiler Architect at StarCore Technology Center (Agere Systems), Atlanta, GA. He is currently a Postdoctoral Research Associate in the Department of Computer Science, University of Illinois, Urbana-Champaign. His research interests include parallel/distributed computing, programming languages, compiler techniques, and software development tools.

Franz Franchetti received the Dipl.-Ing. degree and the Ph.D. degree in technical mathematics from the Vienna University of Technology, Vienna, Austria, in 2000 and 2003, respectively.

From 1997 to 2003, he was with the Vienna University of Technology. Since 2004, he has been a Research Associate with the Department of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA. His research interests center on the development of high-performance digital signal processing algorithms.

Aca Gačić (Student Member, IEEE) received the Dipl.-Ing. degree in electrical engineering from the University of Novi Sad, Novi Sad, Serbia, in 1997 and the M.Sc. degree in electrical engineering from the University of Pittsburgh, Pittsburgh, PA, in 2000. He is currently working toward the Ph.D. degree in electrical and computer engineering at Carnegie Mellon University, Pittsburgh, PA, working on automatic generation and implementation of digital signal processing (DSP) algorithms.

His research interests include representation and implementation of algorithms for signal and image processing, automatic performance tuning for DSP kernels, sensor networks, multiagent control systems, and applications of game theory.

Yevgen Voronenko received the B.S. degree in computer science from Drexel University, Philadelphia, PA, in 2003. He is currently working toward the Ph.D. degree in electrical and computer engineering at Carnegie Mellon University, Pittsburgh, PA.

His research interests include software engineering, programming languages, and compiler design.

Kang Chen received the M.S. degree in computer science from Drexel University, Philadelphia, PA, in 2002, where he worked on the SPIRAL project and did an M.S. thesis on "A prototypical self-optimizing package for parallel implementation of fast signal transforms."

He is currently employed as a Software Design Engineer by STMicroelectronics, Inc., Malvern, PA, and is working on embedded systems for video processing.

Robert W. Johnson received the A.B. degree from Columbia University, New York, in 1962, the M.S. degree from the City College of New York in 1965, and the Ph.D. degree in mathematics from the City University of New York in 1969.

He is Professor Emeritus of Computer Science at St. Cloud State University and is Founder and President of Qwarry Inc. a company devoted to providing hardware/software solutions in math-intensive digital signal processing (DSP) applications. He is also a Cofounder and former Chief Scientific Officer of MathStar, Inc., a fabless semiconductor manufacturer of DSP devices. He has also been the principal or coprincipal investigator for numerous Defense Advanced Research Projects Agency research and development grants over the last two decades. His recent research has centered on the application of abstract algebra to the design and implementation of DSP algorithms.

Nicholas Rizzolo received the B.S. and M.S. degrees in computer science from the University of Illinois, Urbana-Champaign (UIUC), in 2002 and 2004, respectively. He is currently working toward the Ph.D. degree at UIUC, where his research interests include machine learning, programming languages, and compilers.