Computer Generation of Platform-Adapted Physical Layer Software

Yevgen Voronenko (SpiralGen, Pittsburgh, PA, USA; yevgen@spiralgen.com)
Volodymyr Arbatov (Carnegie Mellon University, Pittsburgh, PA, USA; arbatov@cmu.edu)
Christian R. Berger (Carnegie Mellon University, Pittsburgh, PA, USA; crberger@ece.cmu.edu)
Ronghui Peng (SpiralGen, Pittsburgh, PA, USA; jonathan.peng@spiralgen.com)
Markus Püschel (ETH Zurich, Switzerland; markus.pueschel@inf.ethz.ch)
Franz Franchetti (Carnegie Mellon University, Pittsburgh, PA, USA; franzf@ece.cmu.edu)*

Abstract

In this paper, we describe a program generator for physical layer (PHY) baseband processing in a software-defined radio implementation. The input of the generator is a very high-level platform-independent description of the transmitter and receiver PHY functionality, represented in a domain-specific declarative language called Operator Language (OL). The output is performance-optimized and platform-tuned C code with single-instruction multiple-data (SIMD) vector intrinsics and threading directives. The generator performs these optimizations by restructuring the algorithms for the individual components at the OL level before mapping to code. This way known compiler limitations are overcome. We demonstrate the approach and the excellent performance of the generated code on on the IEEE 802.11a (WiFi) receiver and transmitter PHY for all transmission modes.

1 Introduction

A major challenge in implementing software-defined radios (SDRs) is meeting the real-time demands of the signal processing in the physical layer (PHY). For this reason, most common SDR platforms are not software-only, but use a hybrid architecture that, besides a digital signal processor (DSP) or a general purpose processor (GPP), also includes one or more field-programmable gate arrays (FPGA). The PHY functionality is then partitioned across these devices, making the implementation, optimization, and maintenance difficult and costly.

Recently, true SDR has come into reach with modern multi-core GPPs (e.g., Intel Core [1]), or multi-core DSPs (e.g., the Sandblaster DSP [2, 3] or Tilera [4]). However, optimizing the PHY layer for these processors is still very challenging since it requires careful tuning to the memory hierarchy, the explicit use of ν-way single-instruction multiple-data (SIMD) vector instruction, and efficient threading. Further, when the platform changes, the code still has to be reoptimized, since performance usually does not port.

Contribution of this paper. In this paper, we propose to overcome these problems using a program generator for PHYs. The generator is based on Spiral [5–8] and automates the production and optimization of PHY source code. The input to the generator is a high-level, platform-independent description of the PHY receiver or transmitter functionality, described in a domain-specific mathematical declarative language called Operator Language (OL). The output is highly optimized C code including SIMD vector intrinsics and threading directives. Difficult optimizations, such as vectorization, are performed at the OL level using a platform-cognizant rewriting system that effectively manipulates the algorithm based on a few hardware parameters to obtain a suitable structure. Similarly, our generator is able to perform cross-block optimizations and to efficiently map different parts of the computation to different data types (e.g., 16-way bytes for Viterbi decoding and 4-way floating point for the fast Fourier transform).

The main contribution of the presented work is to show that the PHY layer can be expressed in OL (which was prototypically introduced in [9]), to include the necessary support for multiple data types and cross block optimizations into the generator, and finally, the generated code, which has excellent performance.

We demonstrate the latter for both the transmitter and receiver of IEEE 802.11a (WiFi) on the Intel platforms Core and Atom (backends for other architectures are in development). For example, the generated code achieves real-time WiFi transmission speeds for all data rates up to 54 Mbps on an off-the-shelf Intel Core based system. Further, we show that the computer-generated code outperforms the best hand-optimized code.

2 Background and Prior Work

To date most SDR implementations run (fully or partially) on an FPGA [10]. Although FPGAs are reconfigurable, the PHY implementation still has to be done in Verilog or VHDL just

*This work was supported by ONR through the STTR contract N00014-09-M-0332, by NSF through awards 0325687, 0702386, and by DARPA through the DOI grant NBCH1050009.

as for application specific integrated circuits (ASIC). The difference compared to a true software PHY implementation is that in an FPGA or ASIC design, the hardware is designed to match the algorithm, naturally using fine grained parallelism and pipelining to achieve high performance.

First true software PHY implementations used a single DSP that is programmed serially [11–13]. Although fully flexible (and easy to program), these could not achieve real-time performance for computational intensive PHY standards like WiFi. Recent processors possess multiple cores, each typically possessing further parallelism in the form of SIMD vector extensions [1–4, 14, 15]. While these platforms come close to the computational power needed to run PHY standards like WiFi, it also becomes increasingly challenging to implement PHY software that exploits the full computational potential.

Finally, a quite different kind of SDR platforms has surfaced, using simple radio front-end boards attached to commodity personal computers (PCs) [16–18]. These are mostly of interest for academic test-beds as in [18, 19] and run the full functionality on a PC that commonly features an Intel or similar GPP. Hence the challenges to efficiently utilize the computational resources in terms of SIMD and multicore parallelism are quite similar to the SDR platforms described above.

3 Operator Language and PHYs

The operator Language (OL) [9] is a domain-specific declarative mathematical language used to represent certain classes of numerical algorithms. OL is an extension or superset of SPL [5,20,21] to cover non-linear multi-input and multi-output operations. The idea behind SPL and OL is to formally represent algorithm knowledge in a platform-independent way. Actual code is generated from this representation using a number of steps that depend on the target platform. We first introduce SPL and then extend the discussion to OL.

SPL. SPL is a language to describe fast algorithms for linear transforms, which are functions of the form $x \mapsto y = Mx$ with a fixed matrix M. By slight abuse of notation we will simply refer to M as transform. An example is the discrete Fourier transform (DFT) defined by

$$M = \mathbf{DFT}_n = [\omega_n^{ij}]_{0 \le i,j < n},$$

where $\omega_n = e^{-2\pi\sqrt{-1}/n}$. An SPL program, or formula, is a fast algorithm for a transform M represented as a factorization of M into a product of sparse matrices.

SPL contains basic matrices such as the identity matrix I_n, diagonal matrices $\mathrm{diag}_{0 \le m < n}(f(m))$ with a scalar function f, or the stride permutation matrix L^n_k, which transposes an $n/k \times k$ matrix stored linearized in memory. More complex SPL formulas are built from other SPL formulas using matrix operators, such as the matrix product $A \cdot B$ or the Kronecker product $A \otimes B$ defined as

$$A \otimes B = [a_{ij}B]_{i,j}, \quad \text{for } A = [a_{i,j}]_{i,j}.$$

All SPL constructs have natural interpretations as code. For example, the formula $M = A \cdot B$, implies the two-step computation $t = Bx; y = At$.

Using SPL, the well-known Cooley-Tukey fast Fourier transform (FFT) is expressed as

$$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \otimes \mathrm{I}_m)\, \mathrm{diag}\, t^{km}_m (\mathrm{I}_k \otimes \mathbf{DFT}_m)\, \mathrm{L}^{km}_k . \tag{1}$$

It shows that $y = \mathbf{DFT}_{km}\, x$ can be computed in four steps corresponding to the four factors in (1). Two of the steps involve the recursive computation of smaller DFTs.

Further important building blocks of SPL, and later OL, are the following matrices parametrized by index mappings.

An *index mapping* is a function on integer intervals. Denote e^n_i the i-th column basis vector of size n, i.e., the column vector of n elements, with a 1 in i-th position and 0s elsewhere. Given an index mapping function f, *gather and scatter matrices* are defined as follows:

$$\begin{aligned} f &: \{0, \ldots, n-1\} \to \{0, \ldots m-1\}, \\ \mathrm{G}(f) &= \left[e^m_{f(0)} | \ldots | e^m_{f(n-1)} \right], \\ \mathrm{S}(f) &= \mathrm{G}(f)^T = \left[e^m_{f(0)} | \ldots | e^m_{f(n-1)} \right]^T . \end{aligned}$$

OL. OL is a superset of SPL. Where SPL can only describe transforms, i.e., linear single-input and single-output operations, OL removes this restriction and considers more general *operators*. An operator of arity (c, d) is a function that takes c vectors as input and produces d vectors as output. For example, a $k \times n$ matrix M, the simplest possible SPL formula, is in OL viewed as the arity $(1, 1)$ operator

$$M_{k \times n} : \mathbb{C}^n \to \mathbb{C}^k.$$

The matrix product $A_{m \times n} \cdot B_{n \times k}$ becomes in OL the operator composition, e.g.,

$$A_{m \times n} \circ B_{n \times k} : \mathbb{C}^k \to \mathbb{C}^m.$$

The tensor product of matrices generalizes to tensor product of operators, but in this paper we only need one special case. Namely for any arity $(1, 1)$ operator $A : \mathbb{C}^m \to \mathbb{C}^n;\ x \mapsto A(x)$, $\mathrm{I}_k \otimes A$ is defined as

$$\begin{aligned} &\mathrm{I}_k \otimes A : \mathbb{C}^{km} \to \mathbb{C}^{kn}; \\ &\quad x \mapsto (A(x_0, \ldots, x_m - 1), \ldots, A(x_{(k-1)m}, \ldots x_{km-1})). \end{aligned}$$

WiFi physical layer in OL. The 802.11a OFDM transmitter (TX) and receiver (RX) map data bits into a complex baseband signal and vice versa. Formally, and more precisely,

$$\mathbf{WiFiTX}_{k,m,r} : \mathbb{Z}_2^{48kmr-6} \to \mathbb{C}^{80k}, \tag{2}$$

$$\mathbf{WiFiRX}_{k,m,r} : \mathbb{C}^{80k} \to \mathbb{Z}_2^{48kmr-6}. \tag{3}$$

Namely, if a number, say ℓ, bits are to be transmitted at a transmission mode characterized by a modulation scheme with $m \in$

$$\mathbf{WiFiTX}_{k,m,r} = \Big[\mathrm{I}_k \otimes \big(\mathbf{CPIns}_{64} \circ \mathbf{IDFT}_{64} \circ \mathbf{PltIns}_{64} \circ \mathbf{Map}_{48,m} \circ \mathbf{Int}_{48m} \circ \mathbf{Punc}^r_{48m} \circ \mathbf{CvEnc}_{48m,r} \circ \mathbf{Scr}^s_\ell \big) \Big] \tag{5}$$

$$= \Big[\mathrm{I}_k \otimes \big(\underbrace{\begin{bmatrix} \mathrm{G}(h_{64,1}) \\ \mathrm{G}(h_{0,1}) \end{bmatrix} \circ \mathbf{IDFT}_{64} \circ \mathrm{S}(\mathrm{plt}_{48})}_{} \circ (\mathrm{I}_{48} \otimes \mathrm{M}_{1,m}) \circ \underbrace{\mathrm{G}(\mathrm{int}_{48m}) \circ \mathrm{G}(\mathrm{d}^r_{48m}) \circ \mathbf{CvEnc}_{48m,r}}_{} \circ (+s). \big) \Big] \tag{6}$$

$$\mathbf{WiFiRX}^{\tilde{h}}_{k,m,r} = \mathbf{Scr}^s_\ell \circ \mathbf{VitDec}_\ell \circ \mathbf{DePunc}^r_{48km} \circ \Big[\mathrm{I}_k \otimes \big(\mathbf{DeInt}_{48m} \circ \mathbf{DeMap}_{48,m} \circ \mathbf{PltRm}_{64} \circ \mathbf{Eq}_{\tilde{h}} \circ \mathbf{DFT}_{64} \circ \mathbf{CPRm}_{64} \big) \Big] \tag{7}$$

$$= (+s) \circ \underbrace{\mathbf{VitDec}_\ell \circ \mathrm{S}(\mathrm{d}^r_{48km})}_{} \circ \Big[\mathrm{I}_k \otimes \big(\underbrace{\mathrm{S}(\mathrm{int}_{48m}) \circ (\mathrm{I}_{48} \otimes \mathrm{M}^{-1}_{8,m}) \circ \mathrm{G}(\mathrm{plt}_{48})}_{} \circ \underbrace{\mathrm{diag}(\tilde{h}) \circ \mathbf{DFT}_{64} \circ \mathrm{G}(h_{0,1})}_{} \big) \Big] \tag{8}$$

Rate Mbps	Modulation	Bits/sc. m	Code rate r	Coded bits/ symb.	Data bits/ symb., N_{DBPS}
6	BPSK	1	1/2	48	24
9	BPSK	1	3/4	48	36
12	QPSK	2	1/2	96	48
18	QPSK	2	3/4	96	72
24	16-QAM	4	1/2	192	96
36	16-QAM	4	3/4	192	144
48	64-QAM	6	2/3	288	192
54	64-QAM	6	3/4	288	216

Table 1: Data rates in IEEE 802.11a with corresponding modulation schemes and coding rates [22]. (sc. = subcarrier; symb. = symbol)

$\{1, 2, 4, 6\}$ bits per subcarrier and coding rate $r \in \{1/2,\ 3/4,\ 2/3\}$, they will take up

$$k = \lceil \ell / N_{\mathrm{DBPS}} + 6 \rceil = \lceil \ell / (48mr) + 6 \rceil \tag{4}$$

OFDM symbols, where $N_{\mathrm{DBPS}} = 48mr$ is the number of data bits per OFDM symbol, see Table 1. The data bits are appended with zero bits to fill exactly k ODFM symbols and we will always assume the TX and RX operate on this extended bit sequence.

An important difference of physical layer computations from other types of numerical codes is the diversity of used data types, which is also evident above. The transmitter maps bits (denoted with $\mathbb{Z}_2$) to complex (floating point) values (denoted with $\mathbb{C}$), and the receiver vice versa. The actual implementation will use one or more additional data types during the course of computation to optimally use the available vector instruction set. This makes the domain and range specifications of blocks, as in (2)–(3) very important.

We now translate the entire computation data flow of the receiver and transmitter PHY as defined in [22] into OL. The result is (5) and (7), and the occurring blocks (marked bold) are defined in Table 2. We show a further decomposition in (6) and (8), where some of the blocks are broken down themselves. Braces show the grouping of operations in the later implementation.

Table 2 defines all of the blocks in the receiver and transmitter. Most of the blocks are linear, and perform a matrix vector product; the OL definition can thus be interpreted as a matrix. The non-linear blocks are **Map**, **DeMap**, **PltIns**, **VitDec**, and **Scr**.

All of the blocks, except for the Viterbi decoder **VitDec**, can be defined in terms of primitive OL constructs and matrices, and most of the blocks are normally computed by definition. The important exceptions are the DFT, and the Viterbi decoder, for which several alternative fast algorithms exist (e.g., for the DFT the choice of k in (1)), which are again expressed in OL.

PltIns, **PltRm**, **Int**, **DeInt**, **Punc** and **DePunc** are all basically data reorderings or padding operations and thus can be expressed as a gather or scatter with the corresponding index mapping function plt, int, and d respectively for pilot removal/insertion, (de)interleaver, and (de)puncturing. The precise form is not relevant here. We do show the matrix structures of (de)interleaver and (de)puncturer, but in translating these to code, these structures are not used.

The modulator and demodulator are defined by the scalar functions M and M^{-1} that map m hard bits to a complex number, and, vice-versa, a complex number to m soft bit estimates.

Implementation degrees of freedom. Before the OL formulas (6) and (8) can be mapped to code, all remaining unexpanded blocks (in bold) must be expressed in primitive OL constructs. There are multiple ways of doing so that correspond to different computational algorithms and the internal degrees of freedom within the algorithms. Spiral employs feedback driven search to make the best, i.e., fastest choice on the given platform. This search effectively performs platform adaptation. In the interest of space, we only briefly discuss some of these degrees of freedom next.

The tensor product $I_k \otimes A$ itself is not a primitive construct. It has four alternative implementations: as a loop over A, as a parallel loop, as a vectorized loop, or via loop splitting (discussed below). The parallelization and vectorization is implemented using special tags, explained in [6, 7]. If the tensor product is not vectorized, the vectorization is performed "internally," i.e., in A, via rewrite rules (not shown here). An interesting twist in the case of both internal and external vectorization, is that the inner subformula of the tensor product operates on different data types, each having different associated vector length, which makes the vectorization more complex. The equivalent of loop splitting is the transformation $\mathrm{I} \otimes AB \rightarrow (\mathrm{I} \otimes A)(\mathrm{I} \otimes B)$. In our experiments, such "vertical" implementations always performed better.

Operator	Notation	Domain → Range	Definition
Cyclic prefix insertion	$\mathbf{CPIns}_{64}$	$\mathbb{C}^{64} \to \mathbb{C}^{80}$	$\begin{bmatrix} & \mathrm{I}_{16} \\ \mathrm{I}_{64} & \end{bmatrix} = \begin{bmatrix} \mathrm{G}(h_{64,1}) \\ \mathrm{G}(h_{0,1}) \end{bmatrix}$
Cyclic prefix removal	$\mathbf{CPRm}_{64}$	$\mathbb{C}^{80} \to \mathbb{C}^{64}$	$\begin{bmatrix} 0_{64\times 16} & \mathrm{I}_{64} \end{bmatrix} = \mathrm{G}(h_{16,1})$
Forward DFT	$\mathbf{DFT}_{64}$	$\mathbb{C}^{64} \to \mathbb{C}^{64}$	$[\omega_{64}^{ij}]_{0\le i,j<64}$
Inverse DFT	$\mathbf{IDFT}_{64}$	$\mathbb{C}^{64} \to \mathbb{C}^{64}$	$[\omega_{64}^{-ij}]_{0\le i,j<64}$
Pilot tone insertion	$\mathbf{PltIns}_{64}$	$\mathbb{C}^{48} \to \mathbb{C}^{64}$	$(+P) \circ \mathrm{S}(\mathrm{plt}_{48})$
Pilot tone removal	$\mathbf{PltRm}_{64}$	$\mathbb{C}^{64} \to \mathbb{C}^{48}$	$\mathrm{G}(\mathrm{plt}_{48})$
Symbol mapping	$\mathbf{Map}_{48,m}$	$\mathbb{Z}_2^{48m} \to \mathbb{C}^{48}$	$\mathrm{I}_{48} \otimes \mathrm{M}_{m,1}$
Symbol demapping	$\mathbf{DeMap}_{48,m}$	$\mathbb{C}^{48} \to \mathbb{Z}_{2^8}^{48m}$	$\mathrm{I}_{48} \otimes \mathrm{M}_{m,8}^{-1}$
Bit interleaving	$\mathbf{Int}_{48m}$	$\mathbb{Z}_2^{48m} \to \mathbb{Z}_2^{48m}$	$\mathrm{G}(\mathrm{int}_{48m}) = \begin{cases} \mathrm{L}_3^{48m}, & m \le 2, \\ (\mathrm{I}_{16} \otimes_i (\mathrm{I}_6 \otimes \mathrm{Z}_i^{m/2}))\, \mathrm{L}_3^{48m}, & m > 2. \end{cases}$
Bit deinterleaving	$\mathbf{DeInt}_{48m}$	$\mathbb{Z}_{2^8}^{48m} \to \mathbb{Z}_{2^8}^{48m}$	$\mathrm{S}(\mathrm{int}_{48m}) = (\mathbf{Int}_{48m})^T$
Puncturing	$\mathbf{Punc}_{48m}^r$	$\mathbb{Z}_2^{2\cdot 48mr} \to \mathbb{Z}_2^{48m}$	$\mathrm{G}(d_{48m}^r) = (\mathrm{I}_{48m/6} \otimes \mathrm{S}_r)$
Depuncturing	$\mathbf{DePunc}_{48km}^r$	$\mathbb{Z}_{2^8}^{48km} \to \mathbb{Z}_{2^8}^{2\cdot 48kmr}$	$\mathrm{S}(d_{48km}^r) = (\mathrm{I}_{48km/6} \otimes \mathrm{S}_r^T) = (\mathbf{Punc}_{48km}^r)^T$
Convolutional encoding	$\mathbf{CvEnc}_{48m,r}$	$\mathbb{Z}_2^{48mr} \to \mathbb{Z}_2^{2\cdot 48mr}$	$\mathrm{L}_{mr}^{2mr}\, [\mathrm{C}_0 \;\; \mathrm{C}_1]^T$
Viterbi decoding	$\mathbf{VitDec}_{k,m,r}$	$\mathbb{Z}_{2^8}^{2\cdot 48kmr} \to \mathbb{Z}_2^{48kmr-6}$	
Channel equalization	$\mathbf{Eq}_{\tilde{h}}$	$\mathbb{C}^{64} \to \mathbb{C}^{64}$	$\mathrm{diag}\, \tilde{h}$
(De)Scrambling	$\mathbf{Scr}_\ell^s$	$\mathbb{Z}_2^\ell \to \mathbb{Z}_2^\ell$	$\mathrm{I}_l \otimes_i (+s)$

$$S_{1/2} = \mathrm{I}_{12}, \quad S_{2/3} = \mathrm{I}_3 \otimes \begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \end{bmatrix}, \quad S_{3/4} = \mathrm{I}_2 \otimes \begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&1 \end{bmatrix}, \quad \mathrm{Z}_i^n = \begin{bmatrix} & \mathrm{I}_i \\ \mathrm{I}_{n-i} & \end{bmatrix}.$$

$h_{b,s} : i \mapsto b + is$ *(stride s index mapping)*, $(+a) : X \mapsto X + a$ *("add constant" operator)*

$\mathrm{M}_{m,1} : \mathbb{Z}_2^m \to \mathbb{C}$ *is the m-bit modulation operator*, $\mathrm{M}_{m,8}^{-1} : \mathbb{C} \to \mathbb{Z}_{2^8}^m$ *is the demodulation operator to m 8-bit estimates*

$\tilde{h}$ *is the inverted channel freq. response, 64-element vector*, $\mathrm{plt}, \mathrm{int}, \mathrm{d}$ *are index mapping functions, not shown here*.

$\mathrm{C}_0, \mathrm{C}_1$ *are the Toeplitz matrices formed from the degree-7 bit polynomials*

Table 2: Definition of the block operators used in (5)– (8).

Next, the **DFT** and **IDFT** blocks have different vectorization possibilities [6], different choices of radices in the Cooley-Tukey algorithm, and alternative algorithms.

For the Viterbi decoder, [23] gives the OL description of the standard decoding algorithm. However, there exist other algorithms, amenable to parallelization, e.g. [24], which could provide scaling beyond 2 threads enabled by pipelined parallelism. In our implementation of the Viterbi decoder we only consider the degree of unrolling.

The convolutional encoder can be treated as an FIR filter on blocks, and most FIR filter breakdown rules in Spiral apply. Most importantly, these include different vectorization strategies; less important are FIR blocking rules that improve register reuse.

Alternatively, the convolutional encoder can be grouped with the adjacent matrices, resulting in a single matrix-vector product with a less structured matrix. This matrix-vector product has several degrees of freedom including different ways of blocking for locality, and different vectorization methods (tiling into vector-sized diagonals, cyclic diagonals, or vertical stripes).

Operator grouping. The underbraces in equations (8)–(8) show the operator grouping which was used by the code generator. Currently, the grouping is guided by rewrite rules, which can be somewhat ad-hoc, or can be controlled manually by explicitly forcing it. Grouping is very advantageous when the code can be unrolled and resulting temporary buffers scalarized.

4 Optimized Code Generation

Standard code generation process. The standard code generation process in Spiral for SPL / $\sum$-SPL is demonstrated in Fig. 1. In this paper, we use OL / $\sum$-OL, which are extensions, but the code generation process is identical.

In the "Algorithm generation" block, the required functionality (in this case the receiver or transmitter) is expanded using the algorithmic breakdown rules as well as the so-called *paradigm breakdown rules*, which will apply the high-level parallelization and vectorization transformations, based on the characterization of the target platform. Note that the platform

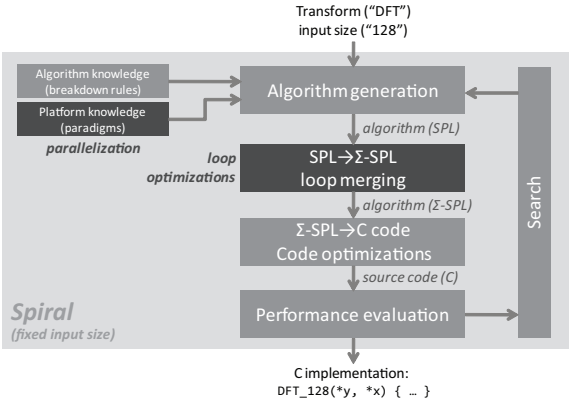

Figure 1: The code generation process in Spiral. The "Search" block closes the performance-driven feedback back loop and enables exploring the space of implementation degrees of freedom and achieve automated platform tuning. In this paper, we use OL / $\sum$-OL instead of SPL / $\sum$-SPL, but the code generation process is identical.

knowledge is used from the very beginning of the code generation process, and enables the generator to undertake major restructuring, if needed.

First, the system will apply (5)–(7), next expand them using the algorithm rules, such as (1), and identities in Table 2, and further restructure the formulas using the paradigm breakdown rules. The process continues until all blocks in bold (called *non-terminals*) are expanded. This yields (6)–(8) initially, and then separately processes the non-terminals that still remain in the formula.

Additional loop level restructuring is performed in the "SPL $\rightarrow$ $\sum$-SPL" block, as explained in [25]. "$\sum$-SPL $\rightarrow$ Code" block generates the final optimized code, as explained in detail below. Finally, the "Performance Evaluation" measures the runtime of generated code, and closes the performance-driven feedback loop that achieves automatic platform tuning.

Note, that here we show *code generation time* platform tuning, while other more general tuning modes are possible, such as *installation time* and *initialization time* (or runtime) tuning, see [26] for details.

Extensions. We had to extend Spiral to be able to generate optimized code for the formulas (5)–(8). This also required extensions for dealing with mixed data-type formulas, bit-level and byte-level SIMD vectorization, and mixed vector length vectorization.

For full vectorization, additional rewriting rules for the vectorization of the modulator, and general bit-matrices were required.

Spiral was able to parallelize the transmitter without major extensions. The parallelization of the receiver required a special breakdown rule that exposed the pipeline parallelism, to mimic the implementation in [27]. This was needed to break the dependency caused by a sequential nature of a Viterbi decoder. This is not the best and most elegant way to deal with this problem. Most probably the fastest implementation (and also the one that would scale to more than 2 threads) will be obtained by directly parallelizing the Viterbi decoder and performing further loop fusion with data processing that precedes the Viterbi block. This can be done by implementing algorithms, such as [24].

Target platform description. Paradigm breakdown rules are guided by the platform knowledge, encapsulated in the platform backend module. Each backend contains both high-level and low-level information about the platform. The high-level information is a set of basic parameters, such as the number of threads, threading model, vector length, cache line length, and others. The low-level information supplies additional information such as the mapping of in-register vector permutations to the machine instructions of the underlying vector instruction set architecture, vector instruction mnemonics, and the syntactic sugaring required for threading and thread synchronization. More details can be found in [6, 7]

Memory hierarchy information is not explicitly used in the platform description, with the exception of the cache line length, which is used by the parallelization to prevent false sharing. However, memory hierarchy adaptation is performed as part of the overall feedback-driven algorithm selection process, i.e., by iterating over different degrees of freedom, which leads to differently structured implementations.

Code generation and optimization. A special Spiral OL compiler generates code from OL formulas. OL compiler, briefly explained in [9] is an extension of the SPL compiler, described in detail in [5, 20, 25].

To generate optimized code, the OL compiler first converts OL into $\sum$-OL, a lower level representation. In this stage constructs like $\otimes$ are converted into iterative sums with gather and scatter matrices. Next, initial code is created by using code generation rules, as shown in the table below. (x and y denote the input and output vectors, t is a temporary vector.)

Parametrized matrices (assume $\mathrm{domain}(f) = n$)

$\mathbf{code}(\mathrm{G}(f), y, x)$	`for(j=0..n-1) y[j] = x[f(j)];`
$\mathbf{code}(\mathrm{S}(f), y, x)$	`for(j=0..n-1) y[f(j)] = x[j];`
$\mathbf{code}(\mathrm{diag}(f), y, x)$	`for(j=0..n-1) y[j] = f(j)*x[j];`

Operators (assume $A : \mathbb{C}^n \rightarrow \mathbb{C}^m$)

$\mathbf{code}(A \circ B, y, x)$	$\mathbf{code}(B, t, x)$; $\mathbf{code}(A, y, t)$;
$\mathbf{code}(\mathrm{I}_k \otimes A, y, x)$	`for(j=0..k-1)` $\mathbf{code}(A, y + mj, x + nj)$;

Finally, the compiler applies a set of standard compiler optimizations, such as loop unrolling, copy propagation, constant folding, and strength reduction. Many of the latter optimizations are enabled by completely unrolling the inner loops with fixed bounds and small number of iterations. For example, the fastest implementation of $\mathbf{DFT}_{64}$, is a fully unrolled "flat" implementation with no control flow at all. The same holds for most other blocks in the PHYs. The degree of such unrolling can be controlled, if needed.

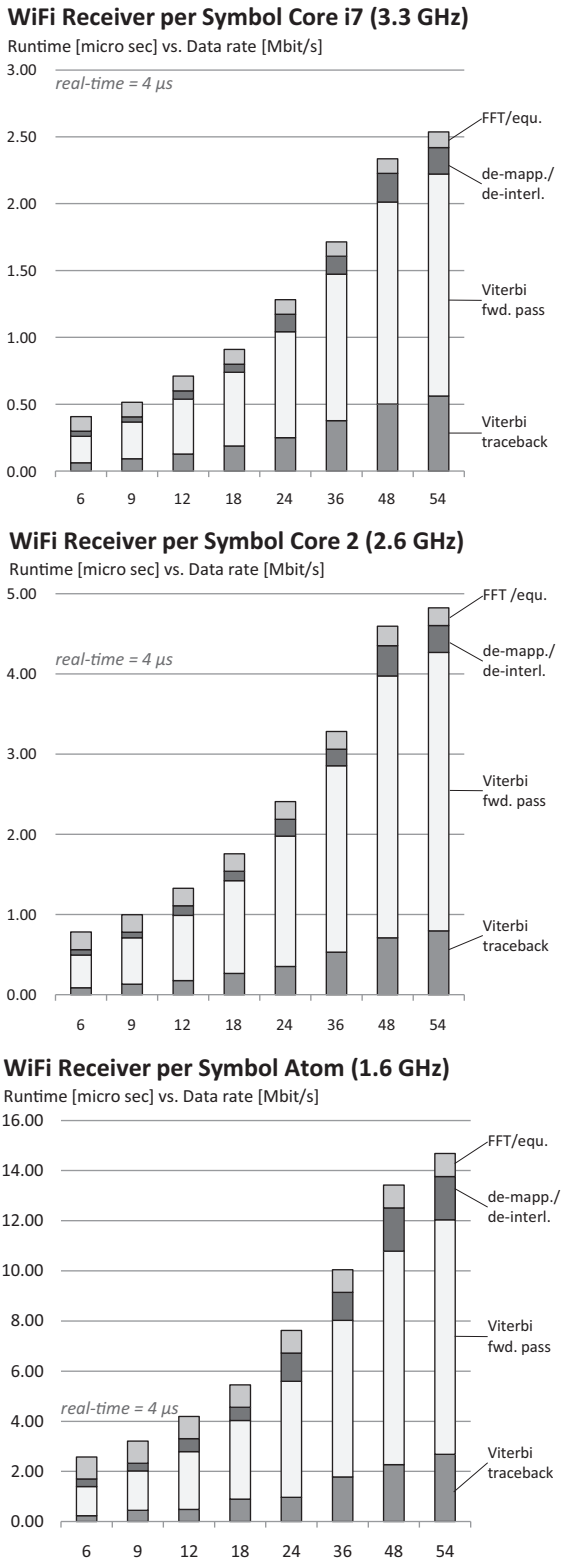

Figure 2: Composition of runtime per OFDM symbol across different architectures in a single-threaded receiver. Viterbi decoder is the most computationally demanding block, and its runtime share increases with the nominal data rate.

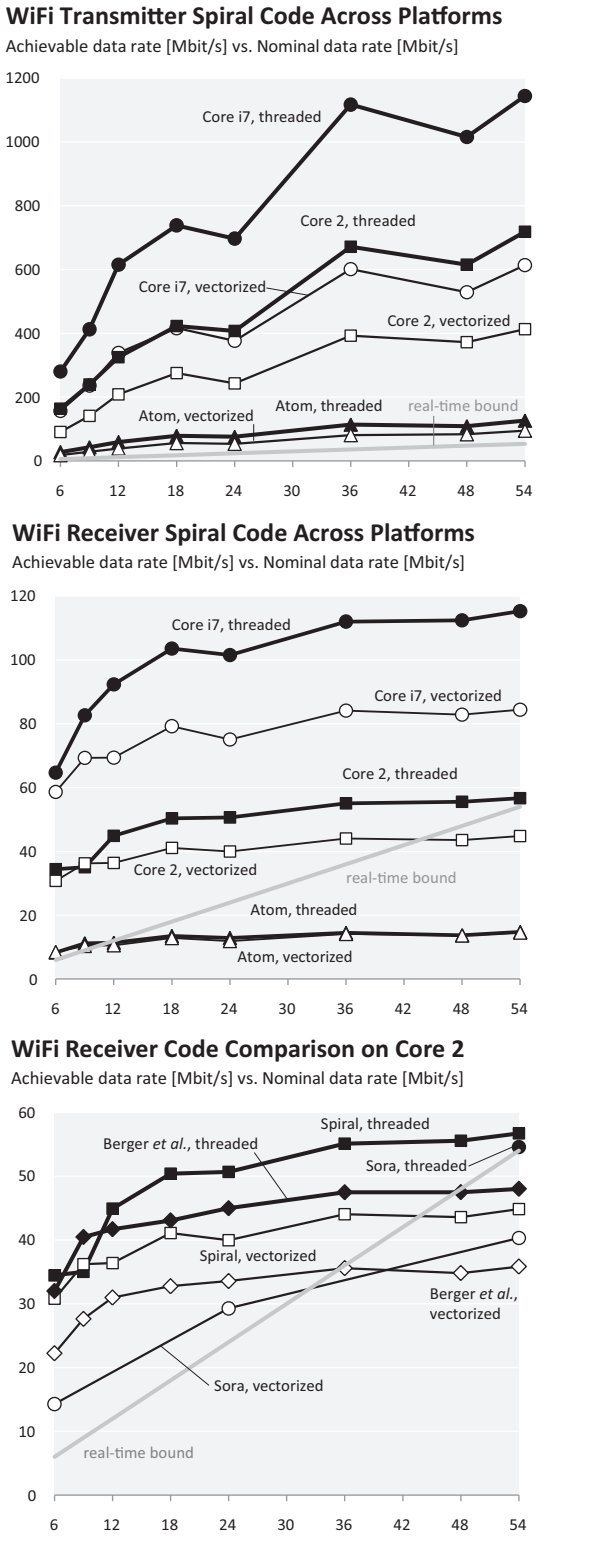

Figure 3: Comparison of the achievable vs. nominal throughput in the transmitter and receiver. The hand-written implementations are Berger [27] and Sora [18]. Implementation marked as "threaded" is both threaded and vectorized. Even though Atom only achieves real-time at 9 Mbps, with its TDP of 2.5W, it provides the best performance per Watt.

In addition, complete loop unrolling eliminates temporary array storage thorough scalar replacement, which in some cases is necessary for optimal performance. Array scalarization also helps when multiple blocks are combined into a single piece of code through grouping.

5 Performance Results

Experimental Setup. In our experiments we performed receiver and transmitter simulation on a synthetic channel data, stored on the hard drive. Thus, the external I/O time from main memory to the hypothetical radio card is not accounted for. The real world measurements, however, should be quite close, as long as external I/O is able to sustain the required data rate, because it can be efficiently done in the background, with data coming to main memory via DMA.

The measured program is the Spiral-generated C99 code with SIMD vector intrinsics. The code was compiled using the Intel Compiler icc 11.0. In all cases, the benchmarks were performed under 64-bit Linux with a 2.6.x series kernel. Hyperthreading was not used, since it only degrades performance for compute-bound workloads such as WiFi.

We benchmarked the generated code on three Intel platforms listed below, by comparing the achievable rate based solely on the required baseband processing runtimes (TDP indicates the thermal design power of the processor):

- Intel Core i7-975, 3.33 Ghz, TDP 130W, 4 cores;
- Intel Core 2 Quad Q6700, 2.66 Ghz, TDP 95W, 4 cores;
- Intel Atom N270, 1.6 Ghz, TDP 2.5W, 1 core.

Benchmarks. The first set of benchmarks analyzes the runtime of the receiver and transmitter per symbol, across different data rates. The recorded runtimes are given in Fig. 2. The bar plots also show the breakdown of runtimes across the computational blocks of the WiFi Receiver running using a single thread. The Viterbi decoder is the most time consuming block, and requires a larger proportion of the runtime as the data rate increases. At 54 Mbps it is 88% of the runtime on the Core platforms and 82% on the Atom; at 6 Mbps, it is 64% on the Cores, and 54% on the Atom. The other blocks are still important and aggressive optimizations and block fusions are needed to reduce their relative runtime to the current level.

The second set of plots in Fig. 3 compares achievable vs. nominal data rates. In these plots, besides varying the platform, we also consider two versions of the generated code, one that was used in the previous set of plots (denoted as "vectorized"), and one that is vectorized and in addition uses two threads as described in Section 4 (denoted as "threaded"). The generated code outperforms both of the hand-coded implementations [27] and [18] we compared against. The two biggest enabling factors are the ability to generate multiple algorithmic code alternatives and search within the available degrees of freedom, and the ability to combine multiple blocks.

In Fig. 3, achievable data rate above nominal means that the processor can process the data fast enough to meet the nominal (required) data rate (i.e., the real-time bound is met). If this is the case, the ratio of nominal data rate over achievable will be approximately equal the CPU load. For example, on Core i7, the generated vectorized receiver achieves 84 Mbps at the nominal rate of 54 Mbps, which means that the CPU load is $54/84 \times 100\% = 64\%$. The threaded and vectorized implementation achieves 115 Mbps, or $54/115 \times 100\% = 46\%$ CPU load on each of 2 used cores.

Generally, the achievable data rate grows with the nominal, because the number of data bits per OFDM symbol increases with the increased nominal data rate, and hence there is less computation per data bit. This increase is non-monotonic, because the amount of computation per bit also depends on the code rate r (see Table 1). For example, there are drops in achievable data rates at nominal speeds of 24 Mbps and 48 Mbps due to reduced r, which slightly bumps up the amount of computation per data bit.

Support for different instruction sets. The next generation instruction set on x86 platforms is AVX, and we already have a validated AVX backend in the generator. However, at the time AVX hardware was not publicly available from either AMD or Intel. Access to pre-release hardware was only possible under NDA to select Intel customers, and thus runtimes cannot be shown in this paper.

Notably, the AVX instruction set poses additional challenges. It doubles the available vector length, without providing any integer data instructions on the wider vectors. Thus, the most important block of the WiFi receiver, the Viterbi decoder, can't benefit, with the additional complications of using wider floating point instructions, for blocks such as the DFT.

The support of multiple instruction sets is even needed on a single platform due to the mixture of different data types. Every one of our generated PHY implementations uses at least 8-bit integer, 32-bit integer, and 32-bit floating point data types. Each data type requires a distinct set of vector instructions to manipulate the data, and the vector length also differs, it is 16 for 8-bit data, 8 for 16-bit data, and 4 for 32-bit data. Finally, for bit-level operations, the vector length is 128.

An extension to other shared-memory multi-core vector architectures is straightforward, as explained in Sec. 4.

6 Conclusion

As true SDR comes into reach, every available performance optimization technique has to be used to achieve maximal performance and real-time. The programming burden hence becomes considerable since the software developer has to use different vector instruction sets, threading, consider available choices, and use a variety of other techniques. A program generator like Spiral is an attractive solution to this problem. As we have demonstrated, the input description is platform-independent and hence has to be created only once. Mapping

to different processors is done by simply changing platform parameters and the backend and regenerating the code. This way, migration can be accelerated, while maintaining excellent performance. We believe that program generators are an important part of a solution to the performance/productivity problem that plagues signal processing and communication applications.

References

[1] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 2, pp. 26–37, Dec. 2009.

[2] V. Ramadurai, S. Jinturkar, S. Agarwal, M. Moudgill, and J. Glossner, "Software implementation of 802.11a blocks on SandBlaster DSP," in *Proc. SDR'06 Technical Conference and Product Exposition*, 2006.

[3] D. Iancu, H. Ye, E. Surducan, M. Senthilvelan, J. Glossner, V. Surducan, V. Kotlyar, A. Iancu, G. Nacer, and J. Takala, "Software implementation of WiMAX on the Sandbridge SandBlaster platform," *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. LNCS 4017, pp. 435–446, 2006.

[4] L. J. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans, "Trends in multicore DSP platforms," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 38–49, Nov. 2009.

[5] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[6] F. Franchetti, Y. Voronenko, and M. Püschel, "A rewriting system for the vectorization of signal transforms," in *Proc. High Perf. Computing for Computational Science (VECPAR)*, 2006.

[7] ——, "FFT program generation for shared memory: SMP and multicore," in *Proc. Supercomputing*, 2006.

[8] Y. Voronenko, "Library generation for linear transforms," Ph.D. dissertation, Dept. of Electrical and Computer Eng., Carnegie Mellon University, 2008.

[9] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *IFIP Working Conference on Domain Specific Languages*, ser. LNCS, vol. 5658. Springer, 2009, pp. 385–410.

[10] P. Koch and R. Prasad, "The universal handset," *IEEE Spectrum*, vol. 46, no. 4, pp. 36–41, Apr. 2009.

[11] M. J. Meeuwsen, O. Sattari, and B. Baas, "A full-rate software implementation of an IEEE 802.11a compliant digital baseband transmitter," in *Proc. IEEE Workshop Signal Processing Systems (SIPS)*, Oct. 2004.

[12] Y. Tang, L. Qian, and Y. Wang, "Optimized software implementation of a full-rate IEEE 802.11a compliant digital baseband transmitter on a digital signal processor," in *Proc. GLOBECOM*, Nov. 2005.

[13] A. L. Cinquino and Y. R. Shayan, "A real-time software implementation of an OFDM modem suitable for software defined radios," in *Proc. Canadian Conf. Electrical and Computer Engineering*, May 2004.

[14] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A high-performance DSP architecture for software-defined radio," *IEEE Micro*, vol. 27, no. 1, pp. 114–123, Jan. 2007.

[15] A. T. Tran, D. N. Truong, and B. M. Baas, "A complete real-time 802.11a baseband receiver implemented on an array of programmable processors," in *Proc. of Asilomar Conf. on Signals, Systems, and Computers*, Nov. 2008.

[16] GNU Radio. [Online]. Available: http://gnuradio.org/

[17] Wireless Open-Access Research Platform (WARP). [Online]. Available: http://warp.rice.edu/

[18] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. Voelke, "Sora: High performance software radio using general purpose multi-core processors," in *Proc. 6th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2009.

[19] M. L. Dickens, B. P. Dunn, and J. N. Laneman, "Design and implementation of a portable software radio," *IEEE Communications Magazine*, vol. 46, no. 8, pp. 58–66, Aug. 2008.

[20] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. PLDI*, 2001, pp. 298–308.

[21] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *IEEE Trans. Circuits and Systems*, vol. 9, pp. 449–500, 1990.

[22] IEEE Computer Society, "IEEE Std 802.11-2007, Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, Revision of IEEE Std 802.11-1999," June 2007.

[23] F. de Mesmay, S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of efficient software Viterbi decoders," in *High Performance Embedded Architectures and Compilers (HiPEAC)*, ser. Lecture Notes in Computer Science, vol. 5952. Springer, 2010, pp. 353–368.

[24] G. Fettweis and H. Meyr, "High-speed parallel viterbi decoding: Algorithm and VLSI-architecture," *IEEE Communications Magazine*, vol. 29, no. 5, pp. 46–55, May 1991.

[25] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. PLDI*, 2005, pp. 315–326.

[26] F. de Mesmay, "On the computer generation of adaptive numerical libraries," Ph.D. dissertation, Dept. of Electrical and Computer Eng., Carnegie Mellon University, 2010.

[27] C. R. Berger, V. Arbatov, Y. Voronenko, F. Franchetti, and M. Püschel, "Real-time software implementation of an IEEE 802.11a baseband receiver on Intel multicore," submitted for publication. [Online]. Available: http://www.ece.cmu.edu/~crberger/paper_allerton.pdf