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 (Carnegie Mellon University, Pittsburgh, PA, USA; pueschel@ece.cmu.edu)
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 limitation are overcome. Further platform tuning is achieved by a feedback-directed search that determines the fastest solution among a space of candidates. We demonstrate the approach and the excellent performance of the generated code on on the IEEE 802.11a (WiFi) receiver and transmitter for all transmission modes.

1 Introduction

A major challenge in implementing true software-defined radios (SDRs) is meeting the real-time demands of the signal processing in the physical layer (PHY). In most cases this requires carefully choosing the proper algorithms, and hand-optimizing and hand-tuning the implementation in assembly for the chosen platform.

Until recently, most common SDR platforms used a hybrid architecture consisting of one or more field-programmable gate arrays (FPGAs), a digital signal processor (DSP) and a general purpose processor (GPP). The PHY functionality was thus partitioned across these devices. The partitioning and hand-optimization process is usually extremely time-consuming, expensive and makes the implementation nonportable.

Today, it has become possible to achieve real-time PHY performance with a single modern multi-core GPP, (e.g., Intel Core [1]), or a single multi-core DSP (e.g., the Sandblaster DSP [2, 3], Tilera). However, developing the code for the performance-critical PHY layer on these processors is an extremely difficult task due to the very complex microarchitectures, diverse cache hierarchies and memory subsystems, and different forms of on-chip parallelism, such as multiple processor cores, multiple threads per core, single-instruction multiple-data (SIMD) instructions, and instruction-level parallelism. Exacerbating the problem, these optimizations have to be redone with every major update of the platform.

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 [4–6] and automates the production and optimization of PHY source code. The input to the generator is a very high-level description of the receiver or transmitter PHY functionality, described in a domain-specific mathematical declarative language called Operator Language (OL). It proceeds with automatically generating various algorithms for the individual components (such as the discrete Fourier transform or the Viterbi decoder), represented in OL. These are then automatically restructured using a platform-cognizant OL rewrite system with the goal to match the algorithm's structure to platform features such as multi-threading and SIMD vector instructions. The obtained OL representation is then compiled into optimized C code including SIMD intrinsics and threading directives. Further platform tuning is achieved by feedback-directed search that determines the fastest solution among a space of candidates.

We demonstrate the viability of the approach on both the transmitter and receiver of IEEE 802.11a (WiFi). The generated code achieves real-time WiFi transmission speeds (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.

The program generation approach hence achieves the seemingly conflicting goals of producing highest performance code while considerably reducing the production time and hence the time it takes to port PHYs to new, more capable platforms.

Our work shows that the PHY functionality can be ex-

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

pressed mathematically at the very high-level in a way that is amenable to mechanical manipulation by a computer. Our representation is not merely a dataflow graph of a signal processing pipeline, but actually exposes all low-level details of the implementation as well, which enables the program generator to expoit all levels of on-chip parallelism. Because of that this "domain-specific compiler" is able to replace the human expert. In addition, we show that the proper representation enables further optimizations, namely block combining optimizations, that are often beyond reach of human experts due to the inherent complexity.

Organization. In Section 2 we overview prior work on implementing baseband PHY processing. Section 3 contains the core technical content of the paper: it introduces the operator language, and shows the description of PHY in OL. Section 4 briefly explains how PHY descriptions in OL can be compiled into optimized code. Finally, we show performance results of the auto-generated code and conclude the paper in Section 5.

2 Background and Prior Work

Early SDR platforms typically consisted of a combination of one or more FPGAs and a part that could run software, like a DSP or GPP. The computational intensive parts where then mostly implemented on FPGAs, which acted as reconfigurable hardware accelerators. Although such a setup has some flexibility, as the platform can accomodate a variety of PHY functionality, the PHY implementation is still mostly done in Verilog matching a non-SDR implementation in application specific integrated circuits (ASIC), and the main challenge is in fitting the computational intensive parts onto the FPGAs.

The difference compared to a true software PHY implementation is that in FPGA or ASIC design, the hardware is designed to match the algorithm, naturally using fine grained parallelism and pipelining to achieve high performance. In contrast, on a DSP or GPP, software mostly executes in a serial fashion, and although modern platforms include parallelism in form of SIMD vectorization and multiple cores, these structures are still fixed and accordingly the algorithms have to be fit to the available hardware.

First true software PHY implementations used a single DSP that is programmed serially [7–9]. Although fully flexible, 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 extnensions [3, 10, 11]. 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) [12–14]. These are mostly of interest for academic test-beds as in [14, 15] 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.

Our work on expressing the SDR PHYs in OL is similar in flavor to the Waveform Description Language (WDL) [16]. However, WDL is much broader in scope than OL, and aims at complete and formal specification of the communication protocol. OL, on the other hand, targets the specific domain of computations with regular structure, and aims to automate the work of expert programmers, by enabling the computer generation of very fast code.

3 Operator Language and PHYs

Operator Language (OL) [17] is a domain-specific declarative mathematical language used to represent certain classes of numerical algorithms. OL is an extension or superset of SPL [4, 5, 18] to cover non-linear multi-input and multi-output operations. 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 also called transform. An example is the discrete Fourier transform 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 into a product of sparse matrices.

To do so, 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 interpretation in the code. For example, the formula $A \cdot B$, 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 computations of smaller DFTs.

Further important building blocks of SPL, and later OL, are the following matrices parameterized 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

Rate Mbps	Modulation	Bits/sc. m	Code rate r	Coded bits / sym.	Data bits / sym., 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 [19]. (sc. = subcarrier)

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:

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

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

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

The tensor product of matrices generalizes to tensor product of operators, but in this paper we only need one special case:

$$A : \mathbb{C}_m \to \mathbb{C}_n$$
$$\mathrm{I}_k \otimes A : \mathbb{C}_{km} \to \mathbb{C}_{kn}$$
$$x \mapsto (A(x_0, \ldots, x_m - 1), \ldots, A(x_{(k-1)m}, \ldots x_{km-1}))$$

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.

$$\mathbf{WiFiTX}_{k,m,r} : \mathbb{Z}_2^{48kmr-6} \to \mathbb{C}^{80k}, \quad (6)$$
$$\mathbf{WiFiRX}_{k,m,r} : \mathbb{C}^{80k} \to \mathbb{Z}_2^{48kmr-6}. \quad (7)$$

If a number, say ℓ, bits are to be transmitted at a transmission mode characterized by a modulation scheme with $m \in \{1, 2, 4, 6\}$ bits per subcarrier and coding rate $r \in \{1/2,\ 3/4,\ 2/3\}$, they will take up

$$k = \lceil \ell / N_{\mathrm{DBPS}} + 6 \rceil = \lceil \ell / (48mr) + 6 \rceil \quad (8)$$

OFDM symbols, where $N_{\mathrm{DBPS}} = 48mr$ is the number of data bits per OFDM symbol. 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 to complex (floating point) values, and the receiver vice versa. The actual implementation will use one or more tertiary data types during the course of computation. This makes domain and range specifications of blocks, as in (6)–(7) very important.

We can directly transcribe the entire computation data flow of receiver and transmitter PHY, as it is explained in [19] into OL using the blocks from Table 2 and this is shown in Fig. 1.

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

All of the blocks, except 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.

PltIns, **PltRm**, **Int**, **DeInt**, **Punc** and **DePunc** are all basically data reordering and/or padding operations and thus can be expressed as a gather or scatter with the corresponding index mapping function (plt, int, and d correspondingly for pilot removal/insertion, (de)interleaver, and (de)puncturing), whose precise form is not relevant here. We do show the matrix structure 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 bits, in the latter soft bit estimates.

Implementation degrees of freedom. Before the OL formulas (3) and (5) 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. For example, Spiral employs feedback driven search to make the best choices on a given platform. We discuss these degrees of freedom next.

The outer tensor product itself is not a primitive construct. Normally, it should become a loop over the right factor. But alternatively, the associated loop can be parallelized, or vectorized [20]. An interesting twist in this case, is that the inner subformula of the tensor product operates on different data

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)*, $\quad (+a) : X \mapsto X + a$ *("add constant" operator)*

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

$\tilde{h}$ *is the inverted channel freq. response, 64-element vector*, $\quad \mathrm{plt}, \mathrm{int}, \mathrm{d}$ *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 (2)– (4).

$$\mathbf{WiFiTX}_{k,m,r} = \Big[\mathrm{I}_k \otimes \big(\mathbf{CPIns}_{64} \circ \mathbf{IDFT}_{64} \circ \mathbf{PltIns}_{64} \circ \mathbf{Map}_{48,m} \circ \mathbf{Int}_{48m} \circ \mathbf{Punc}_{48m}^r \circ \mathbf{CvEnc}_{48m,r} \circ \mathbf{Scr}_\ell^s \big) \Big] \tag{2}$$

$$= \Big[\mathrm{I}_k \otimes \big(\underbrace{\begin{bmatrix} \mathrm{G}(h_{64,1}) \\ \mathrm{G}(h_{0,1}) \end{bmatrix} \circ \mathbf{IDFT}_{64} \circ \mathrm{S}(\mathrm{plt}_{48})}_{} \circ (\mathrm{I}_{48} \otimes \mathrm{M}_{1,m}) \circ \underbrace{\mathrm{G}(\mathrm{int}_{48m}) \circ \mathrm{G}(\mathrm{d}_{48m}^r) \circ \mathbf{CvEnc}_{48m,r}}_{} \circ (+s). \big) \Big] \tag{3}$$

$$\mathbf{WiFiRX}_{k,m,r}^{\tilde{h}} = \mathbf{Scr}_\ell^s \circ \mathbf{VitDec}_\ell \circ \mathbf{DePunc}_{48km}^r \circ \Big[\mathrm{I}_k \otimes \big(\mathbf{DeInt}_{48m} \circ \mathbf{DeMap}_{48,m} \circ \mathbf{PltRm}_{64} \circ \mathbf{Eq}_{\tilde{h}} \circ \mathbf{DFT}_{64} \circ \mathbf{CPRm}_{64} \big) \Big] \tag{4}$$

$$= (+s) \circ \underbrace{\mathbf{VitDec}_\ell \circ \mathrm{S}(\mathrm{d}_{48km}^r)}_{} \circ \Big[\mathrm{I}_k \otimes \big(\underbrace{\mathrm{S}(\mathrm{int}_{48m}) \circ (\mathrm{I}_{48} \otimes \mathrm{M}_{8,m}^{-1}) \circ \mathrm{G}(\mathrm{plt}_{48})}_{} \circ \underbrace{\mathrm{diag}(\tilde{h}) \circ \mathbf{DFT}_{64} \circ \mathrm{G}(h_{0,1})}_{} \big) \Big] \tag{5}$$

Figure 1: Plain OL definitions of WiFi transmitter / receiver ((2) and (4)) and the definitions using OL constructs for simpler blocks ((3) and (5)). Braces show grouping of operations.

types, each having different associated vector length, which makes the vectorization more complex, and we did not implement this in the generator. In addition, an equivalent of loop splitting can be applied: $\mathrm{I} \otimes AB \to (\mathrm{I} \otimes A)(\mathrm{I} \otimes B)$. In our experiments, such "vertical" implementation always performed better.

Next, the **DFT** and **IDFT** blocks have different vectorization possibilities [20], different choices of radices in the

Cooley-Tukey algorithm, and alternative algorithms.

For the Viterbi decoder, [21] gives the OL description of the standard decoding algorithm. However, there exist other algorithms, amenable to parallelization, e.g. [22], which could provide scaling beyond 2 threads enabled by the pipelined parallelism. In our implementation, Viterbi decoder only has the degree of unrolling (# of stages for the unrolled block) as the degree of freedom.

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

4 Optimized Code Generation

We have extended Spiral to be able to generate the optimized code from the formulas in Fig. 1. This required extensions for dealing with mixed data-type formulas, bit-level and byte-level SIMD vectorization, and mixed vector-length vectorization.

Spiral performs the vectorization and parallelization by rewriting at the OL formula level. We had to add additional rewriting rules for the vectorization of the modulator, and general bit-matrices. Spiral was able to vectorize the code, and parallelize the transmitter. Parallelization of the receiver required a special breakdown rule to expose the pipeline parallelism, to mimic the implementation in [24].

In Spiral, a special OL compiler generates code for OL formulas. OL compiler, briefly explained in [17] is an extension of the SPL compiler, described in detail in [4, 23].

In order, to generate optimized code, the OL compiler, first converts OL into $\sum$-OL, a lower level representation, In this stage constructs like $\otimes$ 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 $A : \mathbb{C}^n \to \mathbb{C}^m$)

$\mathbf{code}(A \circ B, y, x)$	$\mathbf{code}(B, t, x)$; $\mathbf{code}(A, y, t)$;
$\mathbf{code}(\mathrm{I}_k \otimes A, y, x)$	`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. 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.

In addition, complete loop unrolling eliminates temporary array storage, which we call array scalarization, which in some cases achieves the largest speedup. Array scalarization also helps when multiple blocks are combined into a single piece of code.

5 Performance Results

We benchmarked the generated code on three platforms listed below (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.

The performance results are given in Fig. 2. The Viterbi decoding is the most time consuming block, and takes the larger proportion of runtime as data rate increases. At 54 Mbps it is 88% of runtime on the Core platforms and 82% on the Atom; at 6 Mbps, it is 63-64% on both Cores, and 54% on the Atom. The other blocks are still important, the aggressive optimizations and block combining are needed to reduce the runtime to the current level.

The second set of plots compares the achievable data rates. The generated code outperforms both of the hand-coded implementations [24] and [14] we compare against. 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.

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] 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.

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

[6] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicore," *IEEE Signal Processing Magazine*, vol. 26, no. 2, pp. 90–102, Dec. 2009.

[7] 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.

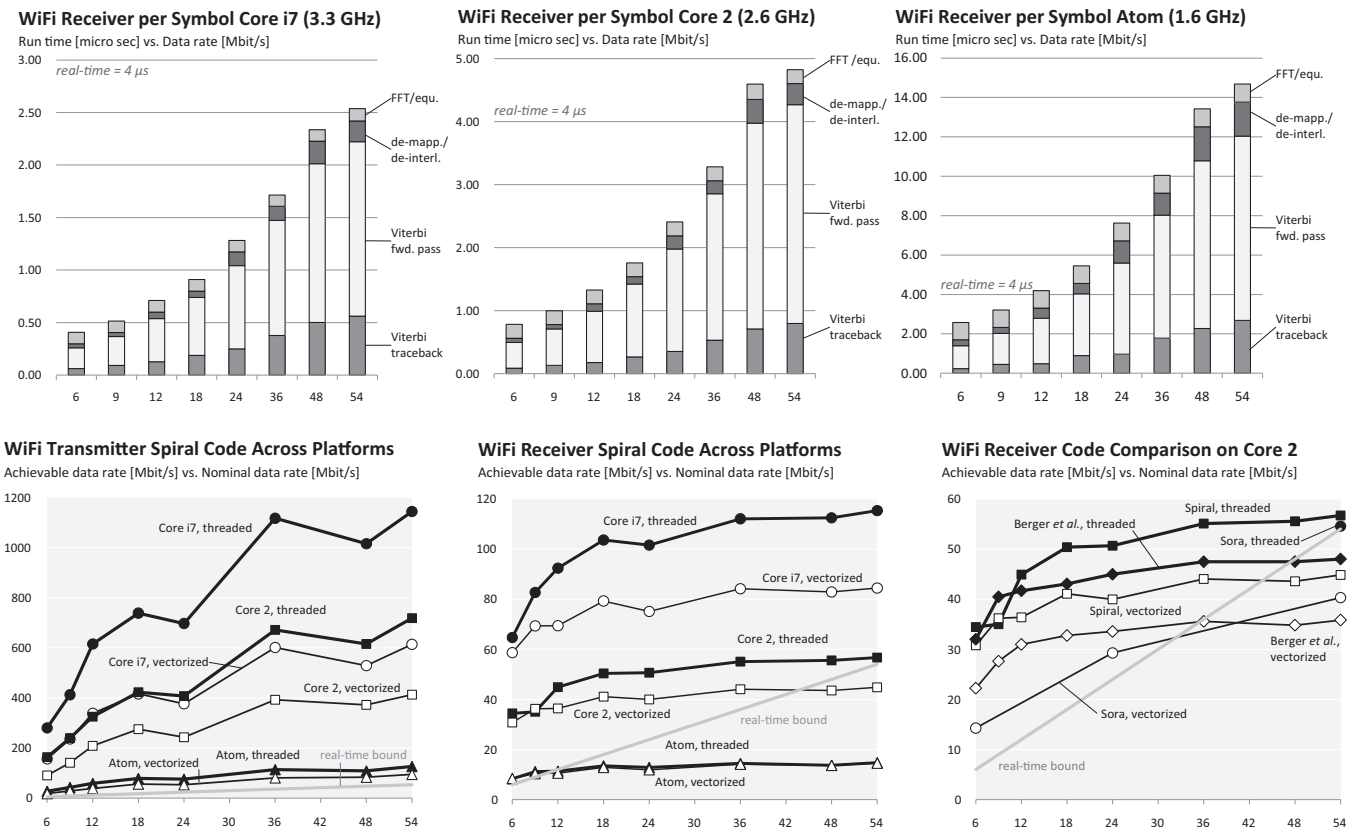

Figure 2: Receiver runtime composition and achievable vs. nominal throughput in the transmitter and receiver. The hand-written implementations are Berger [24] and Sora [14]. Even though Atom only achieves real-time at 9 Mbps, with its TDP of 2.5W, it provides the best performance per Watt.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] E. D. Willink, "The waveform description language: Moving from implementation to specification," in *Proc. MILCOM*, vol. 1, 2001, pp. 208–212.

[17] 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*, LNCS 5658. Springer, 2009, pp. 385–410.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.