REAL-TIME SOFTWARE IMPLEMENTATION OF AN IEEE 802.11A BASEBAND RECEIVER ON INTEL MULTICORE

Christian R. Berger, Volodymyr Arbatov*, Yevgen Voronenko*, Franz Franchetti*, Markus Püschel*[†]

[*]Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA
[†]Dept. of Computer Science, ETH Zurich, Switzerland

ABSTRACT

We present a software-only implementation of an IEEE 802.11a (WiFi) receiver optimized for Intel multicore platforms. The receiver is about 50 times faster than a straightforward C implementation, i.e., an implementation that has the same functionality, but leaves optimization completely to the compiler. Our hand-optimized implementation achieves real-time for all data rates up to the maximum of 54 Mbit/s on a Core i7, clocked at 3.3 GHz, and for up to 12 Mbit/s on an Atom, clocked at 1.6 GHz, using two cores in both cases. To achieve this performance we use up to two threads, up to 16-way vectorization using Intel's SSE, and various other optimizations.

1. INTRODUCTION

Current wireless devices commonly implement all baseband digital signal processing in application specific integrated circuits (ASIC). This allows sufficiently fast processing with low power but the functionality is fixed and cannot be modified or extended. Software-defined radio (SDR) [1, 2] aims to overcome this problem by exchanging the ASIC with a fully programmable platform. This could be a field programmable gate array (FPGA), a digital signal processor (DSP), a full-fledged general purpose processor (GPP), or any combination of those. While FPGAs can be reconfigured to update algorithms or add functionality, DSPs and GPPs can furthermore dynamically assign computational resources, for example between receive and transmit algorithms or between multiple supported wireless standards that do not run concurrently. This is of interest as many devices already support a number of wireless standards, but still are implemented in dedicated hardware.

Available SDR platforms can be roughly divided into two groups: the first consists of low-power stand-alone boards mainly based on DSPs and FPGAs (as used in [3–5]), the second is based on simple radio front-end boards attached to commodity personal computers (PC), therefore performing the actual computing on a GPP [6–8].

Contribution. In this paper we present a software implementation of an IEEE 802.11a (WiFi) receiver optimized for Intel multicore platforms such as Core or Atom. Our baseline is a straightforward C implementation that leaves the optimization completely to the compiler. Hand-optimizing the code by removing unnecessary index computations, avoiding conditional statements, and extensively using single-instruction multiple-data (SIMD) vectorization, we achieve a speedup of about a factor of 30. Parallelization for two cores yields another 20% on Atom and 60% on Core for a total speedup of 35–50 times.

The final code achieves real-time for all data rates up to the maximum of 54 Mbit/s on a Core i7 clocked at 3.3 GHz, and up to 12 Mbit/s on an Atom clocked at 1.6 GHz.

Related work. Software implementations of IEEE 802.11a on stand-alone DSP platforms, like the TI TMS320C64x series, have either considered only the transmitter [3, 4], or are almost 50 times slower than real-time performance [5].

More recently, there has been work on platforms that have higher peak performance due to increased parallelism in the form of multiple processor cores and SIMD vector extensions [9, 10]. Reference [9] presents a software implementation of IEEE 802.16e (WiMAX) on the Sandbridge Sandblaster platform that consists of four DSP cores with SIMD arithmetic units. The code achieves real-time for data rates up to 2.9 Mbit/s in WiMAX, which is functionally similar to WiFi. The signal-processing on-demand (SODA) architecture in [10] consists of four SIMD processing elements, a control unit, and scratchpad memory. Based on hardware simulation the SODA platform should achieve real-time performance for up to 24 Mbit/s using a fully software defined receiver.

The combination of commodity PCs and simple radio front-end boards is especially interesting for academic testbeds (see e.g. [11]) since it enables the testing of new receiver algorithms with limited (software-only) implementation effort. Still, to date there are few implementations of IEEE 802.11a or similar complexity physical layers on academic testbeds.

The popular GNU Radio [6] has only a "miniature version" that runs at a much reduced data rate on a smaller 8 MHz channel, compared to the full 20 MHz bandwidth with up to 54 Mbit/s. This is because an early version of the GNU radio front-end board, the Universal Software Radio Peripheral (USRP), could only transfer data across a USB2 connection to the host PC. But also on the updated version (USRP2), which should theoretically support the necessary throughput via a Gigabit Ethernet connection [6], there is to date no real-time implementation of an IEEE 802.11a receiver.

Other recent academic development kits include Rice University's Wireless Open-Access Research Platform (WARP) [7], and Microsoft Research's Software Radio (Sora) [8]. To our knowledge, the only other real-time software-only implementation of IEEE 802.11a on a GPP is [8], where the authors use hand-coded assembly and large look-up tables to achieve real-time on the Intel server processor Core 2 Quad. In contrast, we achieve real-time both on Intel server processors (Core 2 and Core i7) and mobile processors (Atom), while performing more computation to minimize lookup table sizes to fit Atom's smaller cache size.

2. WIFI RECEIVER

The IEEE 802.11a (WiFi) orthogonal frequency division multiplexing (OFDM) receiver baseband processing consists of seven major steps as shown in Fig. 1: i) The cyclic prefix (CP) is removed and a discrete Fourier transform (DFT) is used to split the orthogonal subcarriers. ii) The channel effect is equalized via a scalar, complex multiplication. iii) The complex symbols are demodulated to render

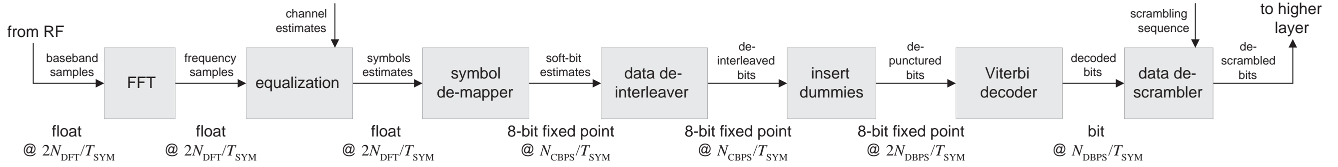


Fig. 1. The OFDM receiver used in IEEE 802.11a (WiFi) has seven blocks, the arrows and variables indicate the data flow; one OFDM symbol is processed every $T_{\mathrm{SYM}} = 4\,\mu$s, the DFT is of size $N_{\mathrm{DFT}} = 64$, coded- and data-bits per symbol are defined in Table 1.

Data rate Mbits/s	Modulation	bit/symb. N_{BPSC}	Code rate R	Coded bit N_{CBPS}	Data bit 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. Supported data rates in IEEE 802.11a with corresponding modulation schemes and coding rates.

bit estimates. iv) The data is de-interleaved; and v) punctured bits are filled with dummy observations. vi) Viterbi decoding determines the most likely transmitted bit sequence based on the applied convolutional code. vii) The data sequence is de-scrambled using a pseudo random sequence.

Next, the blocks in Fig. 1 are described in some detail, with focus on implementation complexity. Performance optimizations for the x86 architecture including vectorization and threading are discussed later.

FFT and equalization. The input at the receiver is typically assumed to consist of complex baseband samples, arriving at the sampling frequency of 20 MHz. During one OFDM symbol duration, $T_{SYM} = 4\,\mu$s, 80 samples arrive, of which the first $N_{\mathrm{CP}} = 16$ contain the cyclic prefix (which is discarded), and the latter $N_{\mathrm{DFT}} = 64$ are fed into a DFT implemented using a fast Fourier transform (FFT). The output contains frequency samples, which are equalized using the channel frequency response (that we assume has been estimated previously) to attain estimates of the transmitted symbols, which are drawn from one of the modulation schemes listed in Table 1.

On DSP boards typically a fixed point format is used to reduce complexity. On an x86 architecture this will not lead to any gains, since efficient floating point arithmetic units are available. Instead the data ordering is relevant for optimal cache usage. An implementation choice here is interleaved vs. split complex format, where in the former each baseband sample's real and imaginary part are stored adjacently, while in the latter all real parts are followed by all imaginary parts.

Symbol de-mapper. The symbol estimates can be found using the minimum-distance receiver (the log-likelihood corresponds to the Euclidean distance). This step is generally of high complexity, since finding the individual bit likelihoods requires combining all symbol likelihoods with matching bits. To reduce complexity a look-up table can be used [8].

Due to the regular structure of the constellations, the bit likelihoods can be also calculated with few arithmetic operations. First, in-phase and quadrature can be handled separately, meaning that one QPSK symbol can be decomposed into two BPSK symbols, 16-QAM into 4-PAM, and 64-QAM into 8-PAM. Furthermore, approximate bit log-likelihoods can be easily calculated due to symmetries in the constellations.

To reduce complexity, the symbol estimates are quantized. Due to the available data structures on an x86 architecture, we choose 8 bit fixed point format (which incurs negligible distortion).

Data de-interleaver. This stage simply reorders the bit estimates. While in ASIC or FPGA implementations this is achieved by connecting the corresponding wires of block inputs and outputs, in software this corresponds to reading from and writing to memory.

Insert dummies. The convolutional codes of rate 2/3 and 3/4 are implemented by modifying the rate 1/2 code using puncturing, which changes the effective rate of a code while avoiding any modification of the encoding/decoding scheme. In practice this means that some encoded bits are not transmitted, and at the receiver dummy observations (log-likelihood of zero) are inserted into the received data stream to keep the appearance of the original coding scheme. This approach is deemed advantageous for hardware implementations.

Viterbi decoder. The Viterbi decoder is of constraint length 7 with generating polynomials 133 and 171 (in octal representation). The associated 64 states are updated using 32 add-compare-select (ACS) steps. The cost of the forward-path is 14 operations per ACS, which includes calculating the cost metrics, updating the paths, and choosing the minimum.

Data de-scrambler. The decoded bits are XORed with the scrambling sequence. The pseudo-random scrambling bit sequence is periodic with a period of $2^7 - 1 = 127$, and is generated in hardware using a length 7 shift register.

3. PARALLELIZATION

3.1. Overview and Minimizing Operations Count

First, we performed some basic optimizations that reduce the operations count and simplify data accesses:

- index computations are simplified to few additions using pointer arithmetic or loop induction counters,
- conditional statements are avoided,
- stages that only reorder data are merged with the adjacent computation,
- compact data types are chosen, e.g., final bit decisions are packed densely.

As explained next, we optimized for parallelism, which on Intel platforms comes in two forms: SIMD vector extensions and thread level parallelism.

3.2. SIMD Vector Extensions

Current SIMD extensions on x86 platforms are called SSE (streaming SIMD extensions) and offer data types and instructions to operate in parallel on 128 bit data types. These include vectors of four single-precision floating point numbers, or vectors of sixteen bytes. The so-called intrinsics interface enables the programmer to use these instructions directly in C. The challenge is in minimizing loads and stores and in-register vector shuffle operations. In the following, we briefly explain how the receiver is efficiently vectorized.

FFT. For the FFT we use code generated and vectorized by our Spiral program generation system [12, 13]. The code is among the fastest ones available on Intel platforms.

Equalization. In the WiFi standard the data organization assumed is the complex interleaved format, which is not favorable for vectorization, as neighboring values need to be combined. Instead we use the split format (all real followed by all imaginary parts), which enables better vectorization.

Symbol de-mapper. The split format carries through from the previous stage; accordingly after the symbol de-mapper the output bit estimates are shuffled by a stride permutation compared to the normal data flow.

Data de-interleaver. The data de-interleaver cannot be vectorized easily, but the reordering of the data in the previous step can be undone by modifying the interleaver. Also we fuse the puncturing with this operation, reducing the number of passes through the data.

Viterbi decoder. Our implementation follows closely [14], which completely sixteen-way vectorizes the Viterbi forward pass using a byte data type. The path metrics in this case are prone to overflow, which has to be addressed. We solve this by reducing the soft-information to 6-bit (although the data type stays the same) and by rescaling the path metrics every four bit. To minimize overhead, the rescaling is merged into the ACS steps. Besides the regular ACS, there is a modified version that subtracts a constant from all path metrics (ACS-write), and a version that determines the minimum path metric across all states (ACS-read). One cycle is ACS, ACS, ACS-read, ACS-write.

The Viterbi trace-back can only be run after the Viterbi forward pass has finished processing all OFDM symbols to minimize overhead. This makes the final bit decisions only available after all OFDM symbols have been received and takes a significant amount of memory (still easily fits into most L2 caches). Alternatively, the traceback can run on overlapping segments.

Data de-scrambler. The data de-scrambler is easily vectorized, except for calculating the scrambling sequence. To vectorize we therefore choose to precompute eight cyclic repetitions plus eight bits taking up 128 byte storage. The data is stored eight bit packed into one byte, bit-wise operations are used instead of SSE instructions:

3.3. Thread Level Parallelism

Thread level parallelism allows several cores to work on the same task. Current Intel CPUs already have two, four and more cores, and the communication between threads is achieved through a shared memory architecture. The main limiting factor is the communication/synchronization overhead if inter-thread dependencies exist.

The maximal throughput of the receiver is limited by the Viterbi decoder's forward pass, which, for each OFDM symbol needs the final path metric of the previous symbol as input. Parallelizing the forward pass is possible but unpractical due to the small workload. Hence we use a different approach using only two threads that operate as a software pipeline as shown in Fig. 2. While one thread will run the Viterbi decoder, the other will do "Viterbi post-processing" of its previous OFDM symbol (Viterbi traceback and data de-scrambling) and "Viterbi pre-processing" of its next OFDM symbol (FFT, equalization, de-modulation, and de-interleaving).

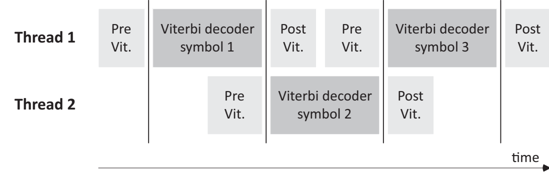


Fig. 2. Threading is done via pipelining using two threads.

4. EXPERIMENTAL RESULTS

We benchmark our implementations on both an Intel Atom N270 with 1.6 GHz and a Core i7 with 3.3 GHz (see [15] for more details on multicore platforms), compiled with the Intel C compiler icc 11. We plot the throughput versus the data rate; i.e., if we measure a time of t micro seconds to receive 25 OFDM symbols then the throughput is $(25N_{\mathrm{DBPS}})/t$ bit/s and the data rate is $N_{\mathrm{DBPS}}/T_{\mathrm{SYM}}$.

We consider the following four code versions in our benchmarks; the difference is in the degree of optimization:

1. *pretty C*: a straightforward implementation in C that neither optimizes index computations nor minimizes passes through the data. The code is easy to read and the optimization is left completely to the compiler.
2. *optimized scalar*: ANSI C code that minimizes the operations count, merges as many parts of the interleaver and puncturing with neighboring stages, and uses a highly optimized scalar FFT function generated by the SPIRAL tool [12].
3. *vectorized*: SIMD vectorized code using intrinsics as outlined in Section 3. In particular the FFT, equalizer, and demodulator use four-way floating point SIMD instructions, while the Viterbi forward pass uses 16-way 8-bit SIMD instructions.
4. *threaded*: This code uses in addition two threads to parallelize the Viterbi forward pass with the other blocks as explained in Section 3.3.

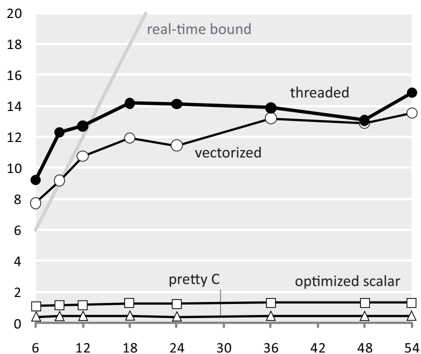
Fig. 3 shows the performance of these implementations for different data rates. Higher is better in these plots, and all implementation above the shown real-time bound achieve real-time.

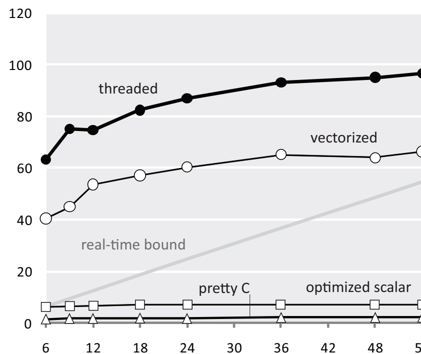
We find that the optimized scalar code is about 4x faster than the naive implementation, while vectorization yields an additional 10x on both Core and Atom.

Two-way threading yields about another 1.6x speed-up on the Core, which comes close to the maximum predicted speed-up considering the fraction of runtime taken up by the Viterbi forward-pass, see Fig. 3 (right). While for Atom the relative improvements of the optimized scalar and vectorized implementations are the same (although on a 5x lower level) as on Core, the threading gain is significantly less, especially at high data rates. This is likely related to the Atom N270's special microarchitecture in which the so-called hyper-threads share functional units.

Our implementation runs in real-time for all data rates on the Core i7 using a single thread, while on the lower power Atom real-time is achieved only up to 12 Mbit/s using both threads.

We now focus on the data rate of 54 Mbit/s and further analyze the block-by-block improvement achieved by single-threaded code variants (Table 2). Although all blocks benefit significantly from the





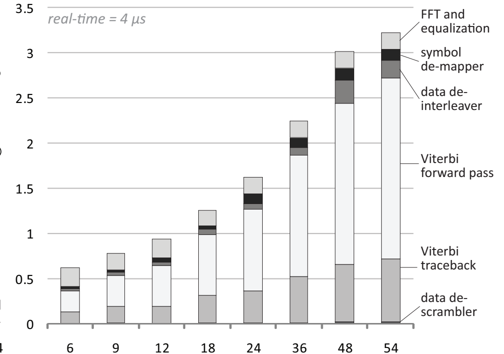


Fig. 3. Throughput measurements on Intel Atom and Core (left and middle respectively), and receiver computation profile for vectorized single-threaded code on Core i7 (right).

Core i7: 54 Mbit/s	pretty C	optimized scalar	vectorized
FFT and equalization	1.66 μs	0.37 μs	0.18 μs
symbol de-mapper	1.35 μs	1.73 μs	0.13 μs
data de-interleaver	3.54 μs	0.16 μs	0.19 μs
Viterbi forward pass	86.04 μs	23.74 μs	2.01 μs
Viterbi traceback	2.96 μs	1.92 μs	0.70 μs
data de-scrambler	1.35 μs	0.03 μs	0.02 μs
total	97.09 μs	28.04 μs	3.29 μs

Table 2. Per symbol timing of individual blocks of 54 Mbit/s mode for three code versions on a 3.3 GHz Core i7.

optimizations, it is obvious that the complexity of the Viterbi forward pass dominates the runtime. A 4x improvement in the Viterbi forward pass is achieved in the optimized scalar code, mainly by formulating the ACS step without any conditional statements. The 16-way vectorization leads to an additional speedup of more than 10x. We do not achieve the full 16x due to the necessary rescaling overhead as discussed in Section 3.2.

Finally we provide a performance profile of the single-threaded vectorized implementation across data rates in Fig. 3 (right). While the runtime per OFDM symbol of the FFT stays constant (there is always one FFT per symbol), we see that the runtime of all other blocks increases (approximately) linear with the data rate, as the operations count is mostly linear in N_{DBPS}.

5. CONCLUSION

A real-time WiFi baseband receiver requires considerable computational performance that is finally within reach on modern general purpose processors due to both SIMD parallelism and multiple cores. Leveraging this parallelism, however, requires a very careful restructuring of the computation. The corresponding implementation leaves standard C and requires vector intrinsics and threading directives. Thus, equally important as the results shown are the techniques used, which will apply across a wide range of modern parallel processors.

6. REFERENCES

[1] W. Wolf, "Building the software radio," *Computer*, vol. 38, no. 3, pp. 87–89, Mar. 2005.

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

[3] 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)*, Austin, TX, Oct. 2004.

[4] 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. of Global Telecommunications Conf.*, St. Louis, MO, Nov. 2005.

[5] 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*, Niagara Falls, Canada, May 2004.

[6] GNU Radio, [online] http://gnuradio.org/.

[7] WARP: Wireless Open-Access Research Platform, [online] http://warp.rice.edu/.

[8] 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*, Boston, MA, Apr. 2009.

[9] 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.

[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] 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.

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

[13] 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.

[14] F. de Mesmay, S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of efficient software Viterbi decoders," in *Proc. Intl. Conf. High Performance Embedded Architectures and Compilers (HiPEAC)*, Pisa, Italy, Jan. 2010.

[15] 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.