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 functional behavior, but leaves the 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.

I. INTRODUCTION

A. Motivation

Current wireless devices commonly implement all baseband digital signal processing in application specific integrated circuits (ASIC). On the one hand this leads to sufficient computational resources at low power consumption, but on the other hand the functionality is fixed and cannot be modified or extended in any way.

The concept of software-defined radio (SDR) [1], [2] aims to change this approach 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 these. This way, modification or total replacement of functionality becomes considerably easier. While FPGAs can be reconfigured to update algorithms or add functionality, DSPs and GPPs can furthermore dynamically assign computational resources, may that be between receive and transmit algorithms or between multiple supported wireless standards that do not run concurrently. This seems especially interesting as nowadays many devices already support an abundance of wireless standards, but still are implemented in dedicated hardware.

So far, 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]), while 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].

Unfortunately, most low-power stand-alone platforms do not offer the performance needed to satisfy the real-time demands of a software-only implementation, and if they do they use specialized hardware accelerators that limit their flexibility. On PCs the available peak-performance of modern multicore processors should be sufficient to achieve real-time, but to utilize this performance efficiently can be very demanding for all but the most expert programmers.

B. 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. We start with a straightforward C implementation that leaves 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. On Core we achieve another speedup of about 60% using two threads, leading to a total improvement in runtime of about 50 times. On Atom we also use two threads, but the additional improvement is only about 20%, since the so-called “hyper-threading” share functional units. In total, the resulting optimized code runs in 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.

Our software implementation could be directly used in SDRs of the second group mentioned above that run on commodity PCs [6]–[8]. Furthermore the trend in the first category, of stand-alone SDRs, is moving from DSPs and FPGAs to low power, specialized GPPs. These have similar architectural features as the GPPs considered here, i.e., vector- and thread-level parallelism, and the optimizations that we perform will be similar to those needed on these and other future platforms.

C. 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 50x 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. In [9] a software implementation of IEEE 802.16e (WiMAX) is presented using the Sandbridge Sandblaster platform that consists of four DSP cores with SIMD arithmetic units; the platform achieves real-time for WiMAX data rates up to 2.9 Mbit/s, which is functionally largely identical to WiFi. The signal-processing on-demand (SODA) architecture 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 [10]. The work in [11] used a multiprocessor DSP board, but needed hardware acceleration for computational intensive kernels.

The combination of commodity PCs and simple radio front-end boards is especially interesting for academic testbeds (see e.g. [12]) since it allows the testing of new receiver algorithms with limited (software-only) implementation effort. Still, so far 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 early versions 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.

II. WiFi Receiver

A. Overview

The IEEE 802.11a (WiFi) orthogonal frequency division multiplexing (OFDM) receiver baseband processing consists of seven major steps defined in [13] as shown in Fig. 1:

- The cyclic prefix (CP) is removed and a discrete Fourier transform (DFT) is used to split the orthogonal subcarriers; the channel effect is equalized via a scalar, complex multiplication.
- The complex symbols are demodulated to render bit estimates.
- Next, the blocks in Fig. 1 are described in some detail, including a cost analysis and pseudo code to illustrate a straightforward implementation. Performance optimizations for the x86 architecture including vectorization and threading are discussed later.
B. Receiver Blocks

**FFT and equalization.** The transmitted baseband signal is

\[ x_i = \sum_k s_k e^{j2\pi \frac{ik}{N_{DFT}}}, \quad i = -N_{CP}, \ldots, N_{DFT} - 1, \]

where the coded information bits are carried by the complex symbols \( s_k \). The received baseband signal has gone through a dispersive channel that can be modeled as a convolution with the baseband channel \( h_\ell, \ell = 0, \ldots, L, L < N_{CP}, \)

\[ y_i = \sum_{\ell=0}^L x_{i-\ell} h_\ell + n_i, \quad i = -N_{CP}, \ldots, N_{DFT} - 1. \]

Also additive noise \( n_i \) has corrupted the signal. Due to the cyclic prefix, the convolution will appear as cyclic within the \( N_{DFT} \) samples. Next a discrete Fourier transform (DFT) is applied,

\[ z_k = \sum_{i=0}^{N_{DFT}-1} y_i e^{-j2\pi \frac{ik}{N_{DFT}}} = s_k H_k + \tilde{n}_k. \]

Due to the cyclic convolution, in frequency we have a product between our transmitted complex symbols \( s_k \) and the channel frequency response

\[ H_k = \sum_{\ell=0}^L h_\ell e^{-j2\pi \frac{ik}{N_{DFT}}}, \quad k = 0, \ldots, N_{DFT} - 1. \]

We assume here that the channel has been estimated with sufficient accuracy in a previous stage. Since the new noise \( \tilde{n}_k \) and \( n_i \) are related by a unitary transform, they share the same statistics. First, the pilot and null subcarriers are removed from the \( \tilde{z}_k \). Then equalization removes the influence of the channel by dividing through the complex channel frequency response,

\[ s_k = z_k / H_k = s_k + \tilde{n}_k / H_k, \quad k = 0, \ldots, N_{SD} - 1. \]

Note that the noise values will now be scaled by the \( H_k \) (the phase rotation is not important). This means that the reliability of different elements of \( \hat{s}_k \) will vary depending on the level of noise, which has to be taken into account when generating soft inputs for the Viterbi decoder.

The equalization in (5) is implemented as a complex multiplication by including the channel magnitudes into the channel estimates. For each of the \( N_{SD} \) data carrying subcarriers equalization will take six real operations. The pseudo code throughout this section will assume complex interleaved format (alternating real and imaginary parts):

```c
for(k=0;k<N_SD;k++)
    H[2*k] = H[2*k]/mag[k];
    H[2*k+1] = -H[2*k+1]/mag[k];

z = fft(y);
for(k=0;k<N_SD;k++)
    s[2*k] = H[2*k]*z[2*k]+H[2*k+1]*z[2*k+1];
```

Assuming \( 5N\log_2(N_{DFT}) + 2N_{SD} \) floating point operations, the cost for this step is:

\[ 5N_{DFT} \log_2(N_{DFT}) + 6 \cdot N_{SD} \]

\[ 4N_{DFT} \]

\[ 2 \cdot N_{SD} \]

**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.

For BPSK, the baseband samples are proportional to the log-likelihood:

\[ \hat{b}_{2k} = \text{Re}(\hat{s}_k), \quad \hat{b}_{2k+1} = \text{Im}(\hat{s}_k), \]

for \( k = 0, \ldots, N_{SD} - 1 \). For 4-PAM, the first bit likelihood is as in BPSK, while the second requires two operations:

\[ \hat{b}_{4k} = \text{Re}(\hat{s}_k), \quad \hat{b}_{4k+1} = 2 - |\text{Re}(\hat{s}_k)|, \]

\[ \hat{b}_{4k+2} = \text{Im}(\hat{s}_k), \quad \hat{b}_{4k+3} = 2 - |\text{Im}(\hat{s}_k)|. \]

For 8-PAM, two bit likelihoods are similar as in 4-PAM, while the last bit requires two more operations:

\[ \hat{b}_{6k} = \text{Re}(\hat{s}_k), \quad \hat{b}_{6k+1} = 4 - |\text{Re}(\hat{s}_k)|, \]

\[ \hat{b}_{6k+2} = 2 - |\text{Re}(\hat{s}_k)|, \]

\[ \hat{b}_{6k+3} = \text{Im}(\hat{s}_k), \quad \hat{b}_{6k+4} = 4 - |\text{Im}(\hat{s}_k)|, \]

\[ \hat{b}_{6k+5} = 2 - |\text{Im}(\hat{s}_k)|. \]

The approximate log-likelihood values \( \hat{b}_k \) are floating point values. To reduce complexity, they are quantized using a \( q \)-bit quantizer with \( 2^q \) levels. We denote the quantizer as a function \( Q(\cdot) \). Due to the available data structures on an x86 architecture, we choose \( q = 8 \) (which incurs negligible distortion). Before the various log-likelihoods can be combined, we have to apply a weighting to the demodulation outputs to account for their reliability, i.e., the quantizer will work differently on each symbol estimate. We apply a scaling factor \( |H_k|^2 \), where the same scaling factor is applied to all bit log-likelihoods from subcarrier \( k \):

\[ \hat{b}_{Nasc+k+m}^{(d)} = Q \left( |H_k|^2 \cdot \hat{b}_{Nasc+k+m} \right), \]

\( k = 0, \ldots, N_{SD} - 1, \quad m = 0, \ldots, N_{BPSC} - 1. \)

The pseudo code takes the form

```c
#define SOFT_MAX 256 // example q = 8
inline unsigned char Q(float x) { if(fabs(x)<(float)(SOFT_MAX/2)) return (unsigned char)(x+(SOFT_MAX/2)); else return (x<0) ? 0 : (SOFT_MAX-1); }
```
The second step is a stride permutation.

The puncturing sequence is a technique that changes the effective rate of a code while avoiding any modification of the encoding/decoding scheme. This approach is seen in practice this means that some encoded bits are not transmitted, as determined advantageous for hardware implementations.

The cost of this step is approximately:
- $6N_{SD}N_{BPSC}$ floating point operations,
- $3N_{SP}$ floating point loads, $N_{BPSC}N_{SD}$ 9-bit stores.

**Data de-interleaver.** The data de-interleaver consists of two steps, where the first is only used for 16-QAM and 64-QAM:

$$b^{(r)}_{n} = b^{(d)}_{n}, \quad n = 0, \ldots, N_{CBPS} - 1,$$

$$r(n) = \left\{ \begin{array}{ll}
2 \lfloor n/2 \rfloor + (n + \lfloor n/12 \rfloor) \mod 2, & N_{BPSC} \leq 2, \\
3 \lfloor n/3 \rfloor + (n + \lfloor n/18 \rfloor) \mod 3, & N_{BPSC} = 4, \\
& N_{BPSC} = 6.
\end{array} \right.$$

The second step is a stride permutation

$$b^{(s)}_{n} = b^{(r)}_{n}, \quad n = 0, \ldots, N_{CBPS} - 1,$$

$$q(n) = 16 \left( n \mod \frac{N_{CBPS}}{16} \right) + \left\lfloor \frac{16n}{N_{CBPS}} \right\rfloor .$$

The pseudo code takes the form

```
for(n=0;n<N_CBPS;n++) // by definition
  bs[n] = b[n];
```

```
for(n=0;n<N_CBPS;n++) // as matrix transpose
  bs[n] = b[n%N_CBPS/16]*n/N_CBPS/16; // br[n];
```

The convolutional codes of rate 2/3 and 3/4 are implemented by modifying the rate 1/2 code using puncturing. Puncturing is a technique that 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.

The puncturing is implemented as shown below; the dummy observations are initialized before the start of reception, as their values and positions are predetermined. Then the actual bit observations are filled in:

$$p^{(s)}_{n} = b^{(s)}_{n}, \quad n = 0, \ldots, N_{CBPS} - 1.$$  

The puncturing sequence $p(n)$ can take one of three forms,

$$p(n) = \left\{ \begin{array}{ll}
n, & R = 1/2, \\
4 \lfloor n/3 \rfloor + (n \mod 3), & R = 2/3, \\
6 \lfloor (n + 1)/4 \rfloor + ((n + 1) \mod 4) - 1, & R = 3/4.
\end{array} \right.$$

**Viterbi decoder.** The Viterbi decoder is of constraint length $N_K = 7$ and has $2^{3K} - 1 = 64$ states, which are updated using 32 add-compare-select (ACS) steps. The implementation follows largely the approach in [14]. The cost of the forward-path is 14 operations per ACS, which includes calculating the cost metrics, updating the paths, and choosing the minimum:

- $14 \times 2N_{DBPS}$ operations,
- $2N_{DBPS}$ 8-bit loads and $64N_{DBPS}$ 1-bit stores.

**Data de-scrambler.** To undo the scrambling, the scrambling sequence $b^{(s)}_{n}$ is added to the decoded bit sequence $b^{(d)}_{n}$,

$$b^{(d)}_{n} = b^{(w)}_{n} + b^{(s)}_{n}, \quad n = 0, \ldots, N_{DBPS} - 1 \quad (14)$$

$$b^{(s)}_{n} = \begin{cases} b^{(s)}_{n-4} + b^{(s)}_{n-7}, & n \geq 0 \\
1, & n < 0 \end{cases} \quad (15)$$

The pseudo-random bit sequence $b^{(s)}_{n}$ is periodic with a period of $2^7 - 1 = 127$. The pseudo code keeps a memory of length seven of the pseudo-random sequence:

```
for(n=0;n<N_DBPS;n++) // calculate random sequence
  bs[n%7] = bs[(n%7)*bs[(n+3)%7]];
```

```
for(n=0;n<N_DBPS;n++) // undo scrambling
  bd[n] = bw[n] - bs[n%7];
```

**III. PARALLELIZATION**

**A. Overview and Minimizing Operations Count**

As we show later, a straightforward C implementation of the receiver based on the pseudo code shown previously underperforms by about two orders of magnitude compared to our best implementation. Two achieve this speed-up we first performed the following basic optimizations:

- index computations are simplified to few additions using pointer arithmetic or loop induction counters,
- avoid conditional statements,
- stages that only reorder data are merged with neighboring stages to reduce overhead,
- data types are chosen as compact as possible, e.g., final bit decisions are packed densely.

As explained next, we then exploited the parallelism offered by Intel’s current multicore platforms, which comes in two forms: single-instruction multiple-data (SIMD) and thread level parallelism.

**B. SIMD Parallelism**

SIMD parallelism is provided in the form of short vector instructions called streaming SIMD extensions (SSE) on x86 architectures. SSE offers data types and instructions to operate by Intel’s current multicore platforms, which comes in two forms: single-instruction multiple-data (SIMD) and thread level parallelism.

**C. SIMD Parallelism**

The convolutional code is in minimizing loads and stores and in-register vector shuffle operations.

In the following, we briefly explain how the receiver is efficiently vectorized. For the pseudo code we assume a vector length of four and use a Matlab like indexing notation to designate vector types.
FFT. For the FFT we use code generated and vectorized by our Spiral program generation system [15], [16]. The code is among the fastest ones available on Intel platforms.

Equalization. The data organization assumed previously a 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:

```c
for(k=0;k<N_SD;k+=4)
  // complex split format
  mag[k:k+3] = H[k:k+3]*H[k:k+3] + H[(k:k+3)+N_SD]*H[(k:k+3)+N_SD];
  H[(k:k+3)+N_SD] = -H[(k:k+3)+N_SD]/mag[k:k+3];
```

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 \( N_{\text{DBPS}} \) stride permutation compared to the normal data flow:

```c
#define SOFT_MAX 256 // example q = 8
inline unsigned char Q(vec x) {
  x += SOFT_MAX/2;
  x = max(x,0);
  x = min(x,SOFT_MAX-1);
  return (unsigned char)x;
}
```

```c
// 64-QAM example
for(k=0;k<N_SD;k+=4)
  // complex split format
  m_vec = mag[k:k+3];
  b_vec = 4-fabs(b_vec);
  bd[(k:k+3)+N_SD] = Q(m_vec*b_vec);
  bd[(k:k+3)+2*N_SD] = Q(m_vec*b_vec);
```

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 the puncturing can be fused into this operation, reducing the number of passes through the data. As an example we show the de-interleaver for 54 Mbit/s, which combines two stride permutations with the \( R = 3/4 \) puncturing:

```c
#define SOFT_MAX 256 // example q = 8
inline unsigned char Q(vec x) {
  x += SOFT_MAX/2;
  x = max(x,0);
  x = min(x,SOFT_MAX-1);
  return (unsigned char)x;
}
```

```c
for(k=0;k<N_SD;k+=4)
  // complex split format
  m_vec = mag[k:k+3];
  b_vec = 4-fabs(b_vec);
  bd[(k:k+3)+N_SD] = Q(m_vec*b_vec);
  bd[(k:k+3)+2*N_SD] = Q(m_vec*b_vec);
```

Viterbi decoder. Our implementation follows closely [14], which completely sixteen-way vectorizes the Viterbi forward pass. 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 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 as defined in (15). To vectorize we therefore choose to precompute eight cyclic repetitions plus eight bits taking up 128 byte storage. This can be used to easily scramble also much longer sequences and the needed modulo operation is simply a power of two (can be implemented as bit-wise AND). The data is stored eight bit packed into one byte, bit-wise operations are used instead of SSE instructions:

```c
for(n=0;n<N_DBPS/8;n++)
  // undo scrambling
  bw[n] = bd[n] "bs[n/n/128]"128;
```

C. Thread Level Parallelism

Thread level parallelism allows several cores to work on the same task. Intel CPUs currently already have 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 many inter-thread dependencies exist.

Our approach to threading assigns different OFDM symbols to different threads. This potentially allows linear speed-up in the number of threads if there are no data dependencies between symbols (and assuming there are enough OFDM symbols to process). In practice, this parallelization approach
Viterbi decoder
symbol 1
Pre Vit.
Viterbi decoder symbol 2
Pre Vit.
Viterbi decoder symbol 3
Post Vit.
Thread 1
Pre Vit.
Viterbi decoder
Post Vit.
Thread 2
Pre Vit.
Viterbi decoder
Post Vit.

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

is impeded here since the Viterbi forward pass of each OFDM symbol needs the final path metrics of the previous OFDM symbol as its initial values. The theoretical maximum throughput is therefore limited by the Viterbi decoders’s forward path (see Fig. 2).

Even in its optimized form, the forward path accounts for more than half of the total processing time for one OFDM symbol (see Fig. 4 in Section IV). We therefore limit the thread level parallelism to 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).

IV. Experimental Results

We benchmark our implementations on both an Intel Atom N270 with 1.6 GHz and a Core i7 with 3.3 GHz clock (see [17] 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 $\frac{25 \times N_{\text{DBPS}}}{t} \text{ bit/s}$ and the data rate is $\frac{N_{\text{DBPS}}}{T_{\text{SYM}}}$.

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 computation nor minimize passes through the data. The code is easy to read but slow, 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 [15].

3) vectorized: starts from optimized scalar, and applies SIMD vectorization as outlined in Section III. In particular the FFT, equalizer, and demodulator use four-way floating point SIMD instructions, while the Viterbi forward pass used 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 III-C.

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 the vectorization yields another 10x, and threading with two threads up to another 1.6x on Core.

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 (as will be shown later in Fig. 4). 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.

The real-time bounds in the Fig. 3 show that our implementation runs in real-time for all data rates on the Core i7 using a single thread, while on the lower power Atom the real-time is achieved for up to 12 Mbit/s using both threads.

We now focus on the data rate of 54 Mbit/s, to further analyze the improvement of the single-threaded code versions on a block level in Tables III and IV. Although all blocks show significant gains due to optimization, it is obvious that the complexity of the Viterbi forward pass dominates the runtime. A four-fold 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, where we do not achieve the full 16x due to the necessary rescaling overhead as discussed in Section III-C.

A clear example of naive (and inefficient) code is the data de-interleaver; implemented following the definition in the standard, as explained in Section II-B, the de-interleaver is the second largest entry, even though it encompasses no actual arithmetic operations.

Finally we provide a performance profile of the single-threaded vectorized implementation across data rates in Fig. 4. 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 $N_{DBPS}$.

In [8], the 54 Mbit/s mode needed 134% of a 2.66 GHz Intel Core 2, and no performance data for the lower-performance processor Atom is available. Our code run on the 3.3 GHz Core i7 outperforms [8] run on a 2.66 GHz Core 2 on a cycle-by-cycle basis (i.e., the performance is scaled by 2.66/3.3) by 30%. We caution that this may not be a fair comparison, since Core i7 and Core 2 have different microarchitectures and [8] does not fully specify the test platform (it is unclear whether the 65 nm or 45 nm Core 2 Quad was used). We double-checked our performance advantage by re-running our code on a 3 GHz Core 2 Quad processor, and obtain a similar result, which let us conclude that our code is indeed faster than [8].

V. CONCLUSION

A real-time WiFi baseband receiver requires considerable computational performance that is finally within reach on modern general purpose processors. The reason is in both SIMD parallelism and multiple cores. Leveraging this parallelism, however, requires a very careful restructuring and 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.

REFERENCES