WiLIS: Architectural Modeling of Wireless Systems

Kermin Elliott Fleming, Man Cheuk Ng, Samuel Gross,
and Arvind
CSAIL, Massachusetts Institute of Technology
{kfleming,mcn02,sgross,arvind}@csail.mit.edu

Abstract

The performance of a wireless system depends on the wireless channel as well as the algorithms used in the transceiver pipelines. Because physical phenomena affect transceiver pipelines in difficult to predict ways, detailed simulation of the entire transceiver system is needed to evaluate even a single processing block. Further, some protocol validations require simulation of rare events (say, 1 bit error in 10^9 bits), which means the protocol must simulate for a long enough time for such events to materialize. This requirement coupled with the heavy computation typical of most physical-layer processing, rules out pure software solutions. In this paper we describe *WiLIS*, an FPGA-based hybrid hardware-software system designed to facilitate the development of wireless protocols. We then use *WiLIS* to evaluate several microarchitectures for measuring very low bit-error rates (BER). We demonstrate, for the first time, that the recently proposed SoftPHY [16, 30] can be implemented efficiently in hardware.

1 Introduction

In digital wireless communication, a transmitter sends data to a receiver by modulating the carrier signal with a signal that represents the digital data being transmitted. The receiver recovers the data from the on-air signal through a reverse process called demodulation. Unfortunately, the carrier signal observed by the receiver is perturbed by various physical phenomena such as noise, interference, multipath induced fading, and shadow fading. In order to permit reliable transmission, both modulation and demodulation involve applying various types of algorithms in series to minimize the impact of these physical phenomena. Examples include: 1) avoidance of bursty errors by shuffling bits, 2) error correction by adding redundancy, and 3) estimations and corrections of noise and fading by sending pre-defined training sequences.

A wireless protocol is defined by the series of algorithms comprising its modulation and demodulation process. Various protocols are designed for different use cases in which the significance of these physical phenomena differs. As a result, to evaluate or validate a particular algorithm in a wireless protocol, it is necessary conduct experiments with other protocol algorithms in-place and with an accurate channel model capable of simulating the effects of the physical phenomena under expected use cases. For example, recent wireless research [16, 30] proposes to modify the physical layer (PHY) of existing 802.11a/g to provide accurate bit-error rate (BER) estimates and to pass these estimates to the upper layers of the protocol stack, where they may be used to improve overall performance. Figure 1 shows the components required to validate and evaluate their protocol. This system represents most of a fully functional 802.11a/g pipeline, with only synchronization and channel estimation absent, even though their suggested modifications are limited to two components, namely the *Soft-Decision Convolutional Decoder* and the *BER Estimator*. There are three primary challenges to simulate such wireless systems for protocol evaluations.

First, validating a protocol often requires observation of events that occur infrequently. This is because many wireless protocols are intended to be able to recover data even when the signal is severely corrupted. On the other hand, the validation process is most interested in *rare* cases in which the data is corrupted. For example, the aforementioned proposal requires BER estimates that can predict BER as low as 10^{-9}, an operating point at which the vast majority of bits are received correctly. Therefore, to achieve reliable measures for an algorithm that produces BER estimates, one needs to produce a statistically significant number of very uncommon events.

Second, most protocols are going to be implemented in ASIC hardware to meet the power, throughput, and latency requirements necessary for deployment. In order to do so, the hardware implementations of many algorithms in a wireless system can only be approximations of the originals. Let us consider BER estimation again: the proposed algorithm requires the whole packet to be buffered before it can produce BER estimates. While the

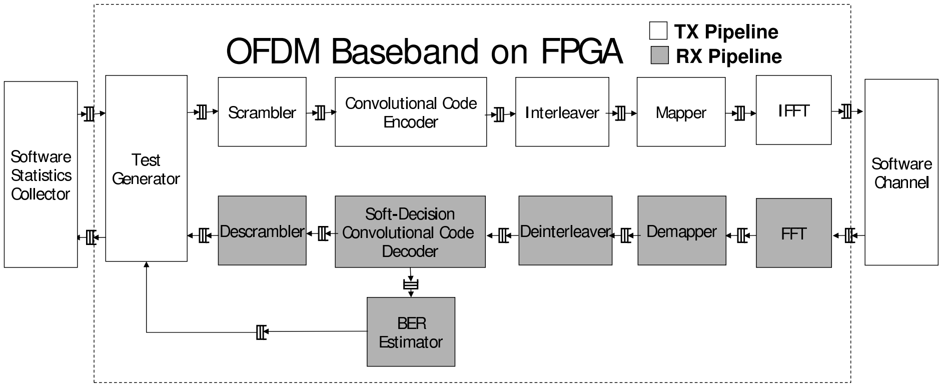


Figure 1: Components required to validate a BER estimator in a co-simulation environment.

original floating point implementation of the algorithm has been proven effective through off-line trace simulation, real-time hardware deployment will require the algorithm to be modified and simplified. Common approximation techniques that might be applied include: 1) using fixed point arithmetic instead of floating point arithmetic; 2) replacing complicated arithmetic with simplified one; 3) replacing full-block processing with a sliding window approach; 4) ignoring less significant terms in the algorithms. In general, these approximations distort the input, and hence the behavior, of downstream modules in ways that are difficult to quantify. Thus to get an accurate characterization of a hardware implementation, even of a single module, we must simulate a large numbers of surrounding *hardware* modules.

Third, it is difficult to obtain realistic traffic to test and debug the wireless systems. Broadcasting data on-air presents difficulty because it is nearly impossible to control the broadcast environment, rendering experiments irreproducible. On the other hand, generating synthetic traffic using set of mathematical channel models usually involves heavy use of high-complexity floating point operations and is best suited for software.

The first two challenges imply that pure software simulator is not suitable because software simulation of detailed hardware is extremely slow; nodes in our simulation cluster process only a few kilobits per second. Parallel simulation across dozens of machines is not sufficient to produce enough of the rarer events for accurate characterization. Meanwhile, the last challenge suggests that accelerating the whole testbench on FPGA can also be problematic because the channel model is not amenable to hardware implementation. Therefore, we opt for FPGA co-simulation, which accelerates the simulation of the hardware pipeline using FPGAs but keeps the channel model implementation in software. The communication between the two are handled by a fast bi-directional link between the FPGA platform and the host PC.

In this paper, we introduce the Wireless Latency Insensitive Simulator (WiLIS) an FPGA-based simulator that facilitates evaluations of wireless protocols at speed close to line-rate. WiLIS is implemented as an extension of Airblue [21], an FPGA software-defined radio platform aimed at providing on-air operation. However, Airblue lacks facilities to validate protocols through high-speed simulation, which WiLIS provides. Therefore, WiLIS and Airblue complement each other in the process of protocol development.

We make two novel contributions in this paper: 1) We implement WiLIS, 2) Using WiLIS, we show that the proposed SoftPHY can be implemented efficiently in hardware, providing insight into a practical microarchitecture of this implementation.

Paper organization: §2 discusses the characteristics of WiLIS that allows it to be an efficient, usable for protocol evaluations. Then, we briefly describe the physical implementation of WiLIS in §3. In §5, we explore other potential approaches to protocol modelling. In §4, we demonstrate the effectiveness of WiLIS through by exploring hardware implementations SoftPHY. Finally, we conclude in §6.

2 WiLIS Properties

We developed WiLIS with the intention of providing a configurable system for high-speed architectural modelling of wireless systems. The following properties of WiLIS are useful for rapidly assembling various wireless models for evaluation.

Latency-Insensitivity: We observe that we require only *functional* and not cycle-accurate simulation to characterize our hardware. Taking advantage of this observation, we implement our functional models using the Airblue [21] toolkit. Modules in latency-insensitive designs do not make assumptions about the latencies of the other modules in the pipeline, a property which has been shown to be useful in modular refinement [7].

In WiLIS, the latency insensitive property of these modules permits us to completely decouple the transmitter, receiver, and channel model, allowing each to operate on its input as soon as that input is available. From a practical perspective, this allows us to improve our communication throughput by way of large, pipelined transfers of data between FPGA modules and software modules and obviating the need for precise synchronization between hardware and software. These optimizations increase our throughput by approximately one order of magnitude.

In addition to performance benefits, the latency-insensitive property also gives us the flexibility to refine or swap the design of any module in the system without affecting the correctness of the whole system. This approach was particularly useful for us in developing our case study, because it enabled us not only to interchange major architectural components but also to transition to the FPGA from software simulation *without modifying any source*.

Automatic Multi-Clock Support: In the hardware portion of our functional model, the throughput of each module in the pipeline may not necessarily match if the whole design is running at the same clock frequency. As a result, the peak performance of the whole pipeline can be bottlenecked by a single slow module. In DSP systems, this rate matching issue, if not addressed, can greatly reduce performance. WiLIS solves this problem by providing automated support for multiple clock domains. A user can change the throughput of a module by specifying a desired clock frequency. WiLIS will automatically instantiate an FPGA primitive providing the specified clock frequency and add special cross-domain communication constructs between every pair of connected modules that are in different clock domains. We achieve this service by extending the mechanisms used by the SoftConnections [24] design tool to carry clock information. In practice, multi-clock support improves modularity. In a typical hardware design users must either pollute their module interfaces to supply submodules with clocks, or the submodule must know its parent's clock frequency to synthesize its own clock. Neither of these cases is portable. Because our compilation tools handle multiple clock domains, WiLIS modules gain a degree of portability.

FPGA Virtualization: In principle, WiLIS can be executed on an FPGA platform as long as the hardware design fits into the FPGA and there is a bi-directional link between the FPGA and a host processor. In reality, various FPGA platforms are connected to the PC through different types of links, e.g., PCI-E and USB. Each type of link requires specific RTL and software codes to be run on both the FPGA and the host processor. Users should be insulated from these details. We implement WiLIS on top of LEAP [23], which is a collection of device drivers for specific FPGA platforms. LEAP provides a set of uniform interfaces across devices like memory and off-FPGA I/O and automatic mechanisms for multiplexing access to these devices across multiple user modules.

By porting Airblue modules to LEAP, we gain portability and modularity. WiLIS models can be run automatically and without code modification on any platform supported by LEAP and providing LEAP I/O functionality, including future high-speed platforms. Because LEAP handles multiplexing of these resources automatically, user modules are insulated from one another in sharing common devices, aiding in modular composition.

Plug-n-Play: In general, WiLIS provides multiple implementations of each module. In many cases, users want to experiment with different combinations through mix-and-matching different implementations. Users may also wish to use their own modules in combination with existing ones. While this can be achieved by modifying the source code, this sort of work is usually tedious and, therefore, prone to error. To facilitate this process, WiLIS provides bindings for AWB [9], an open development tool which provides GUI support for plug-n-play designs. AWB users pick the implementation of each module by choosing from a list of available implementations. This plug-n-play approach greatly increases the speed of constructing a working wireless system.

3 WiLIS Implementation

As a base for WiLIS, we implemented a functional model of a 802.11-like Orthogonal Frequency Division Modulation (OFDM) baseband pipeline, based on the Airblue toolkit. Our design language is Bluespec [6], which is a high-level description language that facilitates development of latency-insensitive designs [22, 10].

For software channel, we implement an Additive White Gaussian Noise (AWGN) channel with a variable Signal-to-Noise-Ratio (SNR). To take advantage of the computation power of multi-core processors, our software channel implementation is multi-threaded.

Our simulation environment consists of a Virtex-5 based ACP FPGA module [15] attached to a 1066 MHz front-side bus (FSB) and a quad core Xeon processor mounted to the same bus. This configuration provides a fast FIFO communication with bandwidth in excess of 700MB/s between the FPGA and the processor. Currently, we configure the FPGA to run the baseband pipeline at 35 MHz with the exception of the BER prediction unit, which runs at 60 MHz since it operates at per-bit granularity. This configuration allows our baseband pipeline to be capable of achieving the throughput of the fastest rate in 802.11g at 54 Mbps.

Figure 2 shows the simulation speed of different rates achieved by our baseline 802.11 system with the software channel. We are able to achieve simulation speeds which are between 32.8% and 41.3% of the line-rate speeds of the corresponding 802.11g rates. At the highest rate, we are able to achieve simulation speeds in excess of 20 Mbps. Experimental results show that all the simulation runs constantly use up only about 55 MB/s of the

Modulation	**Simulation Speed (Mb/s)**
BPSK 1/2 (6 Mbps)	2.033 (33.9%)
BPSK 3/4 (9 Mbps)	2.953 (32.8%)
QPSK 1/2 (12 Mbps)	4.040 (33.7%)
QPSK 3/4 (18 Mbps)	6.036 (35.3%)
QAM-16 1/2 (24 Mbps)	8.483 (35.3%)
QAM-16 3/4 (36 Mbps)	12.725 (35.2%)
QAM-64 2/3 (48 Mbps)	15.960 (33.2%)
QAM-64 3/4 (54 Mbps)	22.244 (41.3%)

Figure 2: Simulation speeds of different rates. Numbers in parentheses are the ratios of the simulation speeds to the line-rate speeds of corresponding 802.11g rates

700 MB/s available communication bandwidth between the FPGA and the processor, indicating that our software modules are the bottleneck of our system. Program analysis shows that computing noise values for the AWGN channel dominates our software time, even though the software is already multi-threaded to take advantage of the four available cores. Since noise generation alone was sufficient to saturate a quad core system, our choice of co-simulation was sound.

4 Case Study: Estimating BER

As mentioned earlier, it would be useful for the physical layer (PHY) of a wireless system to provide accurate bit-error rate (BER) estimates to the upper layers of the protocol stack. For example, Partial Packet Recovery (PPR) [17] uses per-bit BER estimates, the probability that the given bit is in error, to determine the bits to be retransmitted, improving the efficiency of the conventional Link Layer's Automatic Repeat-reQuest (ARQ) mechanism. Conventional ARQ requires the retransmission of the entire packet in the event of any bit error. Another example is SoftRate [31], which uses per-packet BER estimates, i.e., the expected number of bits in the packet that are in error divided by the size of the packet, to dynamically choose the optimal rate for each packet transmission.

It is difficult to estimate the BER of a channel accurately, because the receiver normally does not know in advance the content of the transmitted data. Furthermore, the channel behavior itself may be highly variable, even across a single packet, and must be measured frequently to obtain accurate BER estimates. The recently proposed SoftPHY abstraction [16, 30] offers a solution to the problem of fine-grained BER estimations. SoftPHY makes use of a soft-decision convolutional-code decoder to export a confidence metric, the log-likelihood ratio (LLR) of a bit being one or zero, up the networking stack. While this work has shown that SoftPHY is able to produce high quality BER estimates, it has been evaluated only in software and does not meet the throughput (54-150 Mbps) or the latency (25 μs) requirements of high-speed wireless standards such as 802.11a/g/n. For SoftPHY to be useful, it must be implemented efficiently in hardware while meeting these performance targets.

Soft-decision convolutional-code decoders are commonly used as a kernel for decoding turbo codes [5], and numerous hardware implementations [19, 3, 4, 1, 18] have been optimized for this purpose. These implementations are based on either the BCJR algorithm [2] or the SOVA algorithm [12]. The former usually provides better decoding performance but involves more computation and more complex hardware. To reduce hardware complexity, all these implementations ignore the signal-to-noise ratio (SNR) during the calculation of LLR. While this optimization does affect the performance of turbo codes because they require the LLR outputs only to maintain their relative ordering, it is unclear that the same optimization will be as effective to SoftPHY BER estimations which need to also take into account the magnitude of these values.

To study whether these implementations can be used for BER estimations, we implemented SoftPHY based on both BCJR [3] and SOVA [4] in WiLIS. Then, we empirically evaluated and characterized our designs through simulations of SoftPHY in the context of an 802.11-like OFDM baseband processor shown in Figure 1.

4.1 Convolutional Code Processing

The accuracy of our BER estimation is, in part, determined by the performance of the baseband processor in which it operates. We will briefly describe the baseband components most relevant to BER estimation in the following. [22] contains a more complete description of OFDM baseband processing.

Convolutional encoder: A convolutional encoder is a shift register of $k - m$ bits where k and m are the constraint length and input symbol bit-length respectively. At each time step, an encoder with coding rate of m/n ($n > m$) generates an n-bit output according to n generator polynomials, each specifying the bits in the shift register to be "XORed" to generate an output bit. In our experiment, we use the convolutional code of 802.11a which has constraint length of 7 and code rate of 1/2.

Soft-Decision Convolutional Code Decoder: A soft convolutional decoder produces at its output a decision bit $\hat{b_i}$ and an log-likelihood ratio (LLR) denoting the confidence that the decision is correct with the following definition:

$$LLR_{dec}(i) = log\frac{P[\hat{b_i} = b_i|y]}{P[\hat{b_i} \neq b_i|y]} \quad (1)$$

which is the ratio of the probability that the bit is correctly decoded ($\hat{b_i} = b_i$) to the probability that the bit is incorrectly decoded ($\hat{b_i} \neq b_i$). Given y, the decoder determines the most likely state sequence of the shift register from the encoder that would generate y. There are two common algorithms to decode convolutional code and output LLRs: the SOVA algorithm [12] and the BCJR algorithm [2]. We implemented both, which are discussed in details in §4.3, for a proper hardware evaluation. Next, we discuss the demapper that provides the inputs of the decoder and its hardware implementation.

Demapper: The convolutional code demapper maps each subcarrier's phase and amplitude to a particular set of bits, based on the transmitter modulation scheme. Since these values maybe distorted by the channel, the demapper also assigns a LLR to each demapped bit with the following definition.

$$LLR_{demap}(i) = log\frac{P[b_i = 1|r[k]]}{P[b_i = 0|r[k]]} \quad (2)$$

which is the ratio of the probability that the i-th decoded bit is 1 to the probability that the decoded bit is 0 given the received symbol $r[k]$ at time k that contains bit i.

A good approximation [27] of this LLR under a flat-fading Additive White Gaussian Noise (AWGN) channel can be obtained with the following equation.

$$LLR_{demap}(i) = \frac{E_s}{N_0} \times S_{modulation} \times R_{dist}(i) \quad (3)$$

which is the ratio ($R_{dist}(i)$) between the Euclidean distance of the received symbol to the closest 1 and the distance to the closest 0, multiplied by the signal-to-noise-ratio ($\frac{E_s}{N_0}$) and a constant depending on the modulation scheme ($S_{modulation}$).

We base our demapper on Tosato et al. [29], who further optimize the calculation of $R_{dist}(i)$ by eliminating multiplications and divisions. If $\frac{E_s}{N_0}$ remains roughly the same for all data subcarriers and across the packet transmission, further optimization can be made by ignoring $\frac{E_s}{N_0}$ and $S_{modulation}$ due to the fact that the bit-decoding decisions are determined by the relative ordering of the terms in the convolutional decoding computation instead of their magnitudes. This optimization allows the decoder to achieve the same decode performance with reduced bit-width (23-28 bits → 3-8 bits), which helps significantly reduce the area of the decoder. Unfortunately, the magnitude of the computation is important when estimating the BER.

4.2 BER Estimation

Our BER estimator takes per-bit LLR estimates from the soft decision decoder and translates them into per-bit BER estimates. These estimates may be processed before they are passed up to higher levels, for example by calculating the packet BER.

From equation 1, the LLR estimate can be converted to a per-bit BER with the following equation.

$$BER_{bit} = \frac{1}{1 + e^{LLR_{dec}}} \quad (4)$$

Unfortunately, the LLR estimates produced by either BCJR or SOVA are only approximations of the true LLR. This imprecision has two causes: first, the SNR and the modulation factors (as shown in equation §3) are ignored when the hardware demapper generates the inputs for the decoder; second, the input values are interpreted using different scales by the hardware BCJR and SOVA. We study the impact of these input scalings to a LLR estimate output from both algorithms [2, 12] and find this estimate can be converted to to the true LLR ($L\hat{L}R_{dec}$) with the following equation.

$$L\hat{L}R_{dec} = \frac{E_s}{N_0} \times S_{modulation} \times S_{dec} \times LLR_{dec} \quad (5)$$

where $\frac{E_s}{N_0}$ is the SNR, $S_{modulation}$ is a constant scaling factor determined by the modulation scheme and S_{dec} is another scaling factor determined by the decoder.

One way to implement the per-bit BER estimator is to mathematically calculate the precise value for each scaling factor and then adjust the LLR according to equation 5. After that, the per-bit BER can be obtained by using a lookup table generated following equation 4. While the last two factors can be computed statically, the SNR needs to be estimated at run-time.

Instead of implementing an SNR estimator, we believe that a pre-computed constant for SNR is sufficient. We observe: 1) we only need the BER prediction to be accurate up to the order of 10^{-7} because a maximum size of a packet is usually in the order of 10^4 bits. While the order of 10^{-5} is sufficient for checking packet errors, extra margin can help rate adaptation protocols like SoftRate [31] to identify potential of sending packets at higher rate; 2) the range of SNR over which a modulation's BER drops from 10^{-1} to 10^{-7} is only a few dB [8]. Therefore, we can pick an appropriate SNR constant, i.e., a value in the middle of the SNR range mentioned above for each modulation and still get reasonably accurate BER estimates. This proposal will slightly underestimate the BER if the actual SNR is lower than the chosen middle value and overestimate the BER if the SNR is higher. With this simplification, we can implement a BER estimator as a two-level lookup. Given an LLR output from the decoder and the modulation scheme, we look up the right table and obtain the BER.

4.3 Soft Decision Decoder Architecture

A convolutional encoder is implemented with a shift register. At each time step, it shifts in an input bit, transits to the next state, and produces multiple bits as an output based on the transition. By observing only these outputs, as determined by the demapper, a decoder attempts to determine the most likely state transitions of the encoder. In contrast to hard decision decoders, which output a single decision bit, soft decision decoders produce at their output a decision bit and an LLR denoting the confidence that the decision is correct. Both SOVA and BCJR require minor augmentation to calculate these ratios.

Theoretical work [20] has shown that BCJR and SOVA are deeply related: both SOVA and BCJR decode the data by constructing one or more *trellises*, directed graphs comprised of all the state transitions across all time steps. Each column in a trellis represents all the possible state of the shift register in a particular time step. For example, there will be 2^n nodes in a column if the size of the shift register is n bits. Two nodes are connected with a directed edge if it is possible for the encoder to reach one from the other by way of a single input. Each node is associated with a value called the path metric. Although path metrics have different meanings in BCJR and SOVA, they generally track how likely it is that the encoder was in a particular state at a particular time.

Two kernels are used to calculate path metrics: the branch metric unit (BMU) and the path metric unit (PMU). At each time step, the BMU produces a branch metric for each possible transition by calculating the distance between the observed received output and the expected output of that transition. This distance constitutes an error term: if it is large, then the output associated with the distance is not likely. Then, the PMU calculates the new path metric for each transition by combining the corresponding branch metric with the path metric of the source node from the previous timestep. As both SOVA and BCJR use BMU and PMU, the designs of these two components are shared. The PMU is parameterized in terms of path permutation, which differs between the forward and backward trellis paths of BCJR, and the Add-Compare-Select (ACS) units, which can be different between SOVA and BCJR. The BMU is identical in SOVA and BCJR.

SOVA and BCJR differ in the way they use path metrics to determine the directed edges in the trellis. SOVA attempts to determines the most likely state sequence along a period of time. SOVA requires the PMU to provide the path metrics and their corresponding previous states, i.e., survivor states, at each time step. Using this information, it constructs a sliding traceback window that stores columns of survivor states it received most recently. For each window, SOVA performs a traceback which starts from the node with the smallest path metric for the current time step, and then iteratively follows the survivor state at each earlier time step until it reaches a node belong to the earliest time step in the window. This node is then used to determine the original input to the encoder at that time step.

On the other hand, BCJR seeks to compute the most likely state of the convolutional encoder at each timestep. Given the complete set of encoder outputs, BCJR first calculates the path metric for each state at each timestep moving in a forward direction (α_i) and then computes the path metric for each state in each timestep in the reverse direction (β_i), determining the new path metrics by summing the branch metric - path metric product of incoming trellis edges. Finally, the forward and reverse probabilities for each timestep are combined with the branch transition metric (γ_i) to produce a likelihood for each state at each timestep. The most likely state at each timestep determines the most likely bit input into the convolutional encoder at that timestep.

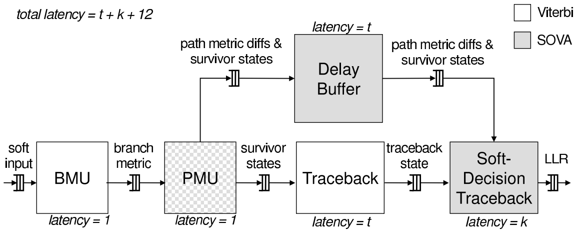


Figure 3: SOVA pipeline: Blocks in white exist in hard-output Viterbi while blocks in grey are SOVA exclusive. Text in italic describes the latency of each block.

In the remaining of the section, we discuss the architectures and the implementation challenges of SOVA and BCJR respectively.

4.3.1 SOVA

Figure 3 shows the structure of our hardware SOVA pipeline, which is based on the one shown in [4]. The pipeline consists of a BMU, a PMU, a delay buffer and two traceback units, all connected by FIFOs.

The two traceback units construct two traceback windows to find the most likely state at each timestep. The results from the first are used as better initial estimates for the second. The second traceback unit also outputs the LLRs. It does so by also keeping track of soft decisions, one for each timestep. Each soft decision represents the confidence of the decoded bit at that timestep. The second traceback unit performs two simultaneous tracebacks, tracking the best and the second best paths, starting with the output state received from the first traceback unit. At each step of the traceback, the states from the two paths are compared. If the two states output different hard decode decisions and the difference of the two path metrics is smaller than the corresponding soft decision, this decision is updated with this smaller value.

The total latency of our SOVA implementation is $l + k + 12$ cycles. l and k are the traceback lengths of the first traceback unit and the second traceback unit respectively. Each BMU and PMU adds an extra cycle of latency. Each FIFO has 2 elements and thus adds at most 2 cycles to the total latency. Therefore, 5 FIFOs add another 10 cycles. If the l and k are both 64, the total latency will be 140 cycles. As our design runs at 60 MHz at least, the latency is no more than 2.3 μs, which implies it can be used in protocols with tight latency bound (25 μs for 802.11a/g).

4.3.2 BCJR

The major difficulty in implementing BCJR lies in the calculation of the backward path metrics. Waiting for an

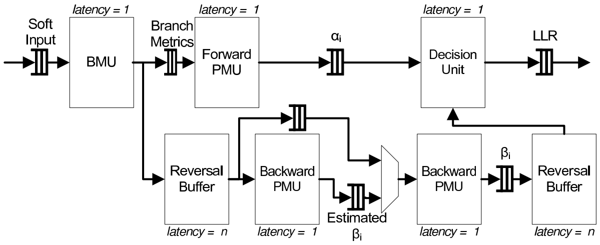


Figure 4: BCJR pipeline.

entire frame of data before beginning computation is unacceptable, both in terms of the latency of processing and in terms of storage requirements. To avoid these issues, we approximate BCJR by operating on sliding blocks of reversed data, the SW-BCJR [3]. Thus, we reverse each block of n data, and determine the backward path metrics of that block in isolation. By making n small, we reduce the latency of the algorithm and reduce storage requirements, at the cost of some accuracy.

However, blocking alone is not enough. In order to process the backwards path for a block p, BCJR must know the final path metric for the succeeding block $p+1$. Unfortunately, this information can only be determined by calculating the reverse path metrics on the remainder of the packet, which we have not yet received. To provide an estimated final path metric for block $p+1$ when calculating block p, we perform a provisional path metric calculation on block $p+1$. Of course, this computation also has uncertain start state, but in this case we use a default "uncertain" state as the initial metric. This configuration shows reasonable performance if block size n is sufficiently large (larger than 32).

Figure 4 shows our streaming BCJR pipeline. Our implementation consists of three major streaming kernels, PMU, BMU, and decision unit, which selects the most likely input bit. In addition to the extra PMU needed for calculating provisional backward path metrics, the backward path also needs a pair of memories to reverse and unreverse blocks. The reversal buffers that we use to re-orient the data frames in the backwards path are based on dual-ported SRAMs. They are streaming, with a throughput of one data per cycle and a latency equal to their size. The pair of reversal buffers and the large FIFO required to cover the latency of the provisional PMU represent a substantial overhead in our architecture. Adding SoftPHY functionality to the architecture is simple: we modify the decision unit to choose both the most like '1' state and the most likely '0' state, subtracting the path metrics of the two states to obtain the LLR. This approach adds only a single subtracter to the pipeline and has no impact on timing.

The latency of BCJR is dominated by the latency of the reversal buffer units, which must buffer an entire block before emitting data. With a reversal buffer of size n the latency of BCJR is $2n+7$, with pipeline and FIFO latency causing the extra constant term. At 60 MHz with a block size of 64 this corresponds to a latency of 135 cycles, or 2.2μs, which is comparable to the latency of SOVA with traceback length set to 64.

4.4 Evaluation

Using WiLIS, we evaluate different aspects of our SoftPHY implementations. First, we study the relationship between the LLR values produced by our hardware decoders and the actual BERs. Then, we evaluate the accuracy of our per-packet BER estimator in the context of SoftRate. Finally, we compare the hardware complexities of our SOVA and BCJR decoders.

4.4.1 Relationship between LLRs and Per-Bit BERs

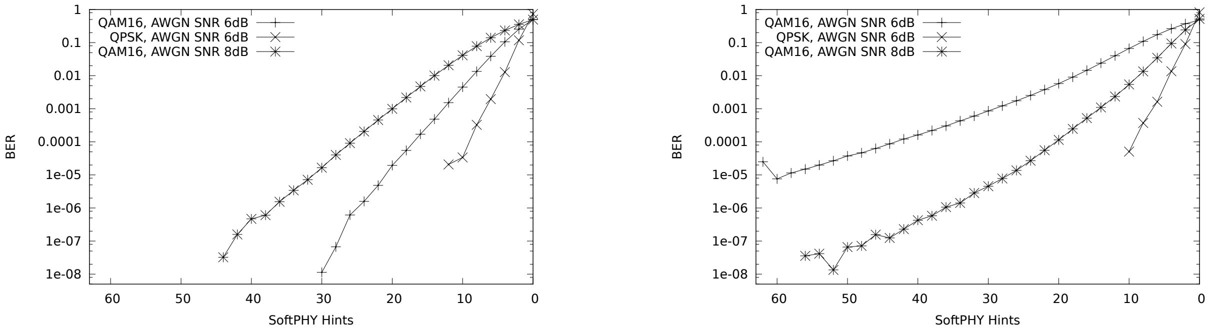
As our WiLIS implementations are approximate algorithms, we must show that the LLR values produced by the hardware decoders correspond well to the LLR suggested by theory. To determine the relationship between these LLR values and the BERs, we simulated the transmission of trillions (10^{12}) of bits on the FPGA. Several resulting curves are shown in 5. Both BCJR and SOVA are able to produce LLRs showing the log-linear relationship with BERs as suggested by the equation 4 in §4.2. As expected, the slopes of the curves vary with SNR, modulation, and decoding algorithm, validating the 3 scaling factors we proposed in equation 5. As a result, we can use these curves to determine the values of these scaling factors and to generate lookup tables for our per-bit BER estimator.

It is important that our implementations are able to produce LLRs that cover a wide range of BERs (i.e., 10^{-7} to 10^{-1}). High per-bit BERs (10^{-2} or above) can predict which bits in the packet are erroneous while low BERs (10^{-7} to 10^{-5}) can predict how likely the whole packet has no error. Although both SOVA and BCJR can produce LLRs that can predict BERs lower than 10^{-7} for some SNRs, BCJR can produce them at a wider range of SNRs than SOVA.

4.4.2 Accuracy of Per-Packet BER Estimates

Per-packet BER (PBER) can be obtained simply by calculating the arithmetic mean of the per-bit BER estimates in a packet. This measure is useful as means of condensing the per-bit BER for communication with higher level protocols. Figure 6 shows the graph plotting the actual PBERs against the predicted PBERs. The predicted PBERs are reasonably clustered around the ideal line, except for high BERs (10^{-1} or above), where there is slight underestimation. These underestimations are a result of the constant SNR adjustment we apply to the decoder's LLR outputs, as discussed in §4.2.

To further test the accuracy of our PBER calculations, we implement SoftRate [31] in WiLIS. SoftRate is a recently proposed MAC protocol which makes use of PBERs to better decide rates at which packets can be transmitted. If the calculated PBER at the current rate is outside of a pre-computed range (for the ARQ link layer protocol, the range is between 10^{-7} and 10^{-5}), then Sof-



(a) BCJR

(b) SOVA

Figure 5: BER v. LLR Hints, across different modulation schemes and noise levels

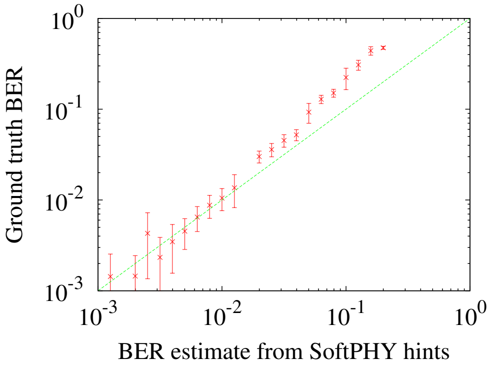


Figure 6: Actual PBER v. Predicted PBER (Rate = QAM16 1/2, Channel = AWGN with varying SNR, Packet Size = 1704 bits). The line represents the ideal case when Actual PBER = Predicted PBER. Each cross with the error bar represents the average of the actual PBERs for that particular predicted PBER value with a standard deviation of uncertainty.

tRate will immediately adjust the future transmission rate up or down accordingly.

In this experiment, the transmitter MAC observes the predicted PBERs emitted by the receiver estimator and adjusts the rate of the future packets, approximating a full transceiver implementation, in which the packet BER estimate would be attached to an ARQ acknowledgement message. We use a pseudo-random noise model which allows us to test multiple packet transmissions at various rates with the same noise and fading across time. We consider the optimal rate to be the highest rate at which a packet would be successfully received with no errors: a rate picked by SoftRate is overselected (underselected) if this rate is higher (lower) than the optimal rate. Figure 7 shows the performance of our SoftRate implementations with BCJR and SOVA under a 20 Hz fading channel with 10 dB AWGN. Both implementations are able to pick the optimal rate over 80% of the time, suggesting that both produce sufficiently accurate PBERs. As expected, SOVA picks the optimal rate less frequently than BCJR by a small margin: SOVA underselects the rate 4% more often than BCJR while both overselect 2% of the time.

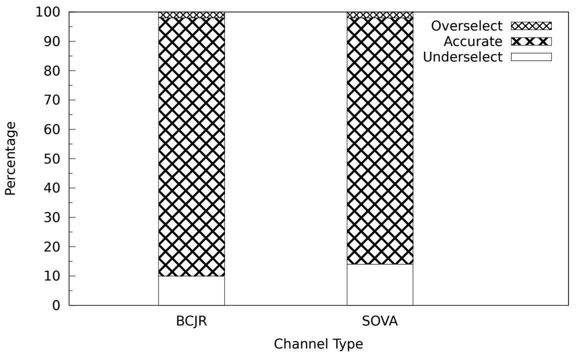


Figure 7: Performance of SoftRate MAC under 20 Hz fading channel with 10 dB AWGN.

4.4.3 Implementation Complexity

Experimental evaluation suggests that BCJR produces superior BER estimates, both per-bit and per-packet. However, this production comes at a high implementation cost. Figure 8 compares the synthesis results of the BCJR and the SOVA decoders using Synplify Pro 2010.09 targeting the Virtex 5 LX330T at 60 MHz. As a baseline, we also show the synthesis results for a Viterbi decoder implementation, as is typically used in commodity 802.11a/g baseband pipelines. We target a processing speed of 60 Mbps, since the maximum line rate of 802.11a/g is rate of 54 Mbps and our decoders are capable of emitting one bit per cycle. Although our designs are optimized to use FPGA primitives like Block RAM, for the purpose of comparison we force the tools to synthesize all storage elements to register.

BCJR is about twice the size of SOVA, primarily due to the three path metric units used by BCJR and its larger buffering requirements. Although BCJR uses fewer registers, this is because it uses large amounts of BRAM. Meanwhile, SOVA itself is about twice the size of Viterbi. The area of both SOVA and BCJR can be reduced by shrinking the length of the backward analy-

Module	LUTs	Registers
BCJR	**32936**	**38420**
Soft Decision Unit	6561	822
Initial Rev. Buf.	804	2608
Final Rev. Buf.	8651	30048
Path Metric Unit	4672	0
Branch Metric Unit	63	41
SOVA	**15114**	**15168**
Soft TU	13456	13402
Soft Path Detect	7362	4706
Viterbi	**7569**	**4538**
Traceback Unit	5144	3927

Figure 8: Synthesis Results of BCJR, SOVA and Viterbi. SOVA is about half the size of BCJR.

sis. In our current implementation, we use a backward path length of 64 for SOVA and a block length of 64 for BCJR. We find that increasing these values provides no performance improvement.

4.4.4 Accuracy of WiLIS Modelling

All models, including those constructed using WiLIS, lose some fidelity as compared to a real implementation. In the case of our WiLIS experiments, our model of the wireless baseband is extremely detailed and accurate: it has been used to build high quality radio transceivers in Airblue. However, the channel models used by WiLIS are certainly approximations of a real wireless channel, and the on-air capabilities of the modules that we have introduced in this study are unknown.

Because we do not have an on-air implementation of SoftPHY or SoftRate, the best comparison that we can make is against previously published [31]. Although the original SoftPHY results were trace-driven, they were based on on-air data collection and provide at least some basis for comparison. WiLIS based-simulation suggests an accuracy rate of 85% for the SoftRate protocol, while the original paper achieved only a 75% accuracy, a differential of 16%. Our hardware model, being an approximation, should have intuitively *underperformed* the ideal software implementation originally proposed. There are likely three contributing error terms in WiLIS simulation. First, our channel model is relatively simple. Second, we took steps to compensate for SNR variability in our SoftPHY implementation, while the original implementation ignored these issues. Third, we did not model channel estimation or synchronization in the receiver. These three factors would serve to increase the apparent performance of SoftRate. Ultimately, we view the discrepancy between the two experiments as acceptable: the offered performance gain of SoftRate is high, around 2x to 4x depending on the base of comparison.

5 Related Work

While WiLIS achieves high simulation speed by the technique of decoupled latency-insensitive design, there are other simulation alternatives which could provide equally high performance. On one hand, we could have built software functional simulation using existing software radios. On the other hand, we could have used a latency-sensitive hardware design, such as one created in Simulink [14] and an emulation technique like the Standard Co-Emulation Modelling Interface (SCE-MI) [13].

Software radios like GNU Radio [11] can be used for architectural exploration. GNU Radio in particular offers a well developed and freely available library of wireless components written in C++. However, GNU Radio, like other software radios, suffers from a performance of only a few Kbps [26]. Even recent, highly optimized software radios [28] find that Viterbi's algorithm alone requires a full processor core to maintain performance of several Mbps. In WiLIS, however, we seek to model algorithms that are known to be 3-4 times more complex than Viterbi [25]. We believe that, in the best case, a well-tuned software radio will be able to achieve a few tens to hundreds of Kbps performance for these algorithms, whereas WiLIS has no problem achieving performance near the line rate.

Emulation of latency-sensitive hardware focuses on the maintenance of cycle accuracy. In the case of SCE-MI, cycle accuracy is maintained by appropriately gating the clock to the latency-sensitive hardware, with the clock ticking only when all the inputs for a given module in a give cycle have been collected. This permits slower modules, for example those implemented in software, to appear to operate at the same speed as the gated modules. The speed at which a design may be emulated is determined by the combination of the speed at which the slower modules can source data to or sink data and the overhead incurred by SCE-MI control model. Although WiLIS is also constrained by its slowest module, SCE-MI emulation will likely incur a larger overhead because the time that the slow module uses for processing typically cannot be used by other modules to perform their own internal work.

Another difficulty in using latency-sensitive hardware for modelling is modification. If a component module is modified, for example, switching Viterbi to BCJR, an architecture that has a very different latency, many modules up and down the pipeline may need to be modified. For a simulator to be useful, particularly to a domain specific audience not completely familiar with hardware design, this level of modification is unacceptable.

6 Conclusion

We believe that FPGAs represent an ideal platform for the development of new wireless protocols. First, a satisfactory FPGA implementation generally implies that a satisfactory ASIC implementation exists. Second, because of the infrequency of many events associated with wireless transmission, high-speed simulation is needed to validate and characterize the implementation. To this end, we developed WiLIS, a flexible and detailed co-simulation platform capable of detailed modelling of an OFDM baseband at speed near the line rate.

WiLIS achieves high-performance yet detailed simulation by accelerating computationally intensive portions of the wireless simulator on the FPGA. WiLIS achieves flexibility through the ability to substitute modules into an existing pipeline without having to modify the remainder of the pipeline and by permitting non-performance critical modules to be implemented in software. The techniques that enable this flexibility are latency-insensitive design, a plug-and-play module substitution, and FPGA virtualization.

We used WiLIS to evaluate two hardware implementations of SoftPHY, a recently proposed protocol. Although the BCJR implementation of SoftPHY outperformed the SOVA implementation, the latter performed acceptably well, and at less than 50% of the area of the former. Generally speaking, the hardware implementations were quite successful at predicting BER with what we believe is an acceptable hardware cost (around 10% increase in the size of a transceiver), indicating that SoftPHY is a competitive augmentation to future wireless chips and protocols. Without a flexible, high-speed simulator like WiLIS, the rapid evaluation of these designs would not have been possible.

References

[1] E. Y. S. Augsburger, W. R. Davis, and B. Nikolic. 500 Mb/s Soft Output Viterbi Decoder. In *ESSCIRC'02*.

[2] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE TIT*, 20(2), 1974.

[3] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara. Soft-Output Decoding Algorithms for Continuous Decoding of Parallel Concatenated Convolutional Codes. In *ICC'96*.

[4] C. Berrou, P. Adde, E. Angui, and S. Faudeil. A Low Complexity Soft-Output Viterbi Decoder Architecture. In *ICC'93*.

[5] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding. In *ICC'93*.

[6] Bluespec Inc. http://www.bluespec.com.

[7] N. Dave, M. C. Ng, M. Pellauer, and Arvind. Modular Refinement and Unit Testing. In *MEMOCODE'10*.

[8] A. Doufexi, S. Armour, P. Karlsson, A. Nix, and D. Bull. A Comparison of the HIPERLAN/2 and IEEE 802.11a Wireless LAN Standards. *IEEE Commun. Mag.*, 40, 2002.

[9] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 2002.

[10] K. Fleming, C.-C. Lin, N. Dave, J. Hicks, G. Raghavan, and Arvind. H.264 Decoding: A Case Study in Late Design-Cycle Changes. In *MEMOCODE'08*, Anaheim, CA, 2008.

[11] GNU Radio. `http://www.gnu.org/software/gnuradio/`.

[12] J. Hagenauer and P. Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *GLOBECOM'89*.

[13] http://www.eda.org/itc/scemi.pdf. Standard co-emulation modelling interface (sce-mi): Reference manual.

[14] http://www.mathworks.com/products/simulink/. Mathworks simulink.

[15] http://www.nallatech.com. Nallatech acp module.

[16] K. Jamieson. *The SoftPHY Abstraction: from Packets to Symbols in Wireless Network Design*. PhD thesis, MIT, Cambridge, MA, 2008.

[17] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *SIGCOMM'07*.

[18] L. Lin and R. S. Cheng. Improvements in SOVA-Based Decoding For Turbo Codes. In *ICC'97*.

[19] G. Masera, G. Piccinini, M. Roch, and M. Zamboni. VLSI Architectures for Turbo Codes. *IEEE Trans. on VLSI Systems*, 1999.

[20] R. J. Mceliece. On the bcjr trellis for linear block codes. *IEEE Trans. Inform. Theory*, 1996.

[21] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, and A. H. Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ANCS'10*, San Diego, CA, 2010.

[22] M. C. Ng, M. Vijayaraghavan, G. Raghavan, N. Dave, J. Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *MEMOCODE'07*.

[23] A. Parashar, M. Adler, M. Pellauer, K. Fleming, and J. Emer. Leap: An operating system for fpgas. 2010.

[24] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft Connections: Addressing the Hardware-Design Modularity Problem. In *DAC'09*, San Francisco, CA, 2009.

[25] P. Robertson, E. Villebrun, and P. Hoeher. A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain. In *ICC'95*.

[26] T. Schmid, O. Sekkat, and M. B. Srivastava. An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios. In *2nd ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, Montreal, Quebec, Canada, 2007.

[27] M. Speth, A. Senst, and H. Meyr. Low Complexity Space-Frequency MLSE for Multi-User COFDM. In *GLOBECOM'99*.

[28] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: High Performance Software Radio Using General Purpose Multi-core Processors. In *NSDI'09*, Boston, MA, 2009.

[29] F. Tosato and P. Bisaglia. Simplified Soft-Output Demapper for Binary Interleaved COFDM with Application to HIPERLAN/2. In *ICC'02*.

[30] M. Vutukuru. *Physical Layer-Aware Wireless Link Layer Protocols*. PhD thesis, MIT, Cambridge, MA, 2010.

[31] M. Vutukuru, H. Balakrishnan, and K. Jamieson. Cross-Layer Wireless Bit Rate Adaptation. In *SIGCOMM'09*.