ALGORITHMIC ADVANCES FOR SOFTWARE RADIOS

Matteo Frigo (Vanu Inc., One Porter Sq., Cambridge, MA 02140, athena@vanu.com)

ABSTRACT

We present two algorithms: a novel demodulator for Complementary Code Keying (CCK), and a "lazy" variant of the Viterbi algorithm. These algorithms are more suited to software implementations than existing algorithms for the same problems. The new algorithms are noise-adaptive: their running time is not constant, but instead it depends on the noise conditions. We argue that this property is desirable for software radios.

1. INTRODUCTION

Algorithms that are appropriate for hardware radio implementations are not necessarily the most adequate solution for a software radio. Roughly speaking, hardware offers a large degree of parallelism, but it is constrained by a fixed data flow. On the other hand, software must use a small set of CPU resources, but it can decide at runtime which strategy to process the input signal is best. Because of these differences, many algorithms that were designed with hardware implementations in mind need to be reconsidered from a software-radio perspective.

This paper details some recent progress made by Vanu Inc. in developing algorithms for software radios. Specifically, we discuss two algorithms: a novel demodulator [4] for the Complementary Code Keying (CCK) modulation scheme [12] used in IEEE 802.11b, and the *lazy Viterbi decoder* [3], a maximum-likelihood decoder for convolutional codes that is meant to replace the celebrated Viterbi algorithm [14, 6] in our software-radio systems.

These two algorithms share the following feature: Their running time is not constant, but it depends on the signal-to-noise ratio (SNR) of the input signal. The lower the noise, the faster the algorithm. While this property may be inappropriate for hardware designs, it is advantageous for software. First, under good noise conditions, a noise-adaptive algorithm saves CPU cycles, therefore reducing power consumption and prolonging battery life. Second, even at the minimum tolerable SNR level, a noise-adaptive algorithm may still be substantially faster than an algorithm designed for the worst case. For example, no need exists to use an algorithm that works well at 0 dB if the system is meant to work only at SNR greater than 5 dB.

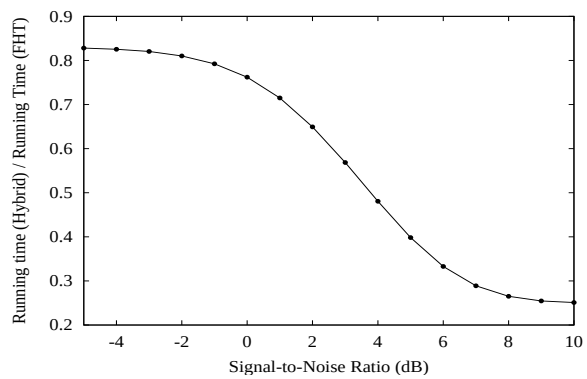

Figure 1: Running time of Hybrid CCK decoding algorithm as a function of SNR for CCK Demodulation. Running time is given as a fraction of the running time for the FHT maximum-likelihood decoder [12].

To illustrate the benefits of noise-adaptivity, consider the Complementary Code Keying (CCK) modulation scheme used in the IEEE 802.11b wireless LAN standard. The maximum-likelihood CCK demodulator from [12] employs an algorithm based on the fast Hadamard transform (FHT). In order to produce 8 bits of output, the demodulator requires about 500 arithmetic operations, which amounts to $687.5 \cdot 10^6$ operations per second at the standard data rate of 11 Mbps. Even without counting the overhead of loops, function call, pipeline stalls, and register spills, this computational load is onerous for a software implementation on current CPUs. Our best implementation of the CCK demodulator requires about $1375 \cdot 10^6$ cycles to process one second of data on a relatively state-of-the-art Athlon XP processor. On the other hand, under good noise conditions, our "Hybrid" demodulation algorithm runs about 4 times faster than the algorithm from [12]. Figure 1 shows the running time of the Hybrid CCK demodulator as a function of the SNR of the input signal. (The Hybrid algorithm is not maximum-likelihood, but the loss of optimality is negligible. See Figure 2.)

In the remainder of this paper, we discuss the new algorithms in detail. Section 2 discusses our new CCK demodulator, and Section 3 discusses our convolutional decoder.

2. A NOVEL CCK DEMODULATOR

The IEEE 802.11b standard for wireless local area networks has high data rates in order to operate at speeds comparable to Ethernet. Complementary Code Keying (CCK) was adopted by the IEEE as the modulation scheme to achieve this data rate [12]. In this section, we detail the Hybrid algorithm for CCK decoding, and give experimental results that show a significant improvement in running time with only a negligible loss in error rate. This algorithm is a special case of a more general decoder for first-order Reed-Muller codes that we have published in [4]. The same paper [4] also proves analytical bounds on the error rate of the Hybrid algorithm.

In CCK modulation, an information sequence (c_0, c_1, c_2, c_3) is a block of four symbols, where $c_i \in \{0, 1, 2, 3\}$. These symbols are modulated using QPSK to values $\phi_i = \omega^{c_i}$, where $\omega = e^{\pi j/2} = j = \sqrt{-1}$, and encoded into eight complex numbers $(y_0, \ldots, y_7)$ using the following encoding function:

$$\begin{aligned} y_0 &= \phi_0 & y_1 &= -\phi_0\phi_1 \\ y_2 &= \phi_0\phi_2 & y_3 &= \phi_0\phi_1\phi_2 \\ y_4 &= -\phi_0\phi_3 & y_5 &= \phi_0\phi_1\phi_3 \\ y_6 &= \phi_0\phi_2\phi_3 & y_7 &= \phi_0\phi_1\phi_2\phi_3 \end{aligned} \tag{1}$$

These eight symbols are then subject to a noisy channel. We use $(r_0, \ldots, r_7)$ to denote the noisy symbols received at the other end of the channel. We have $r_i = y_i + N_i$, where N_i denotes the noise. Based on the received vector r, the decoder must output hard estimates $\hat{c}_i$ of the information symbols c_i, where $i \in \{0, 1, 2, 3\}$.

2.1. Majority-logic decoding

Consider first the case where there is no noise in the channel, i.e., $N_i = 0$, so that $r_i = y_i$ for all i. The decoding problem is now easy. For example, consider the expression $-r_1 r_0^*$. If there is no noise in the channel, then $-r_1 r_0^* = -y_1 y_0^* = \phi_1$. Similarly, $-r_4^* r_6 = -y_4 y_6 = \phi_2$, and $r_7 r_3^* = y_7 y_3^* = \phi_3$. Therefore, when there is no noise in channel, we can simply "read off" ϕ_1, ϕ_2 and ϕ_3 using simple arithmetic operations between certain received symbols.

In reality, these computations will be corrupted by noise, and will not always yield the correct answer. For example, we have $-r_1 r_0^* = (-y_1 + N_1)(y_0 + N_0)^*$. In expectation, however, we still have $-r_1 r_0^* = \phi_1$, and if the noise is low, then $-r_1 r_0^*$ is a good approximation to ϕ_1.

The principle behind majority logic decoding is to use simple computations on the received bits to produce "votes" for the value of each information symbol. In hard decision majority logic, the value that receives the most votes becomes the decoded information symbol. In soft decision majority logic, the votes are "soft" values, and they are averaged to form a "soft estimate" for each information symbol. Ideally, these votes should involve as many code bits as possible so that local noise cannot drastically affect our estimate.

In CCK, we use $(\hat{\phi}_1, \hat{\phi}_2, \hat{\phi}_3)$ to denote the soft estimates for ϕ_1, ϕ_2, ϕ_3 (we will cover the special case of ϕ_0 in Section 2.2), and compute each of them based on four votes as follows:

$$\begin{aligned} \hat{\phi}_1 &= (-r_1 r_0^* + r_3 r_2^* - r_4^* r_5 + r_7 r_6^*) \,/\, 4 \\ \hat{\phi}_2 &= (\ \ r_2 r_0^* - r_1^* r_3 - r_4^* r_6 + r_7 r_5^*) \,/\, 4 \\ \hat{\phi}_3 &= (-r_4 r_0^* - r_1^* r_5 + r_6 r_2^* + r_7 r_3^*) \,/\, 4 \end{aligned}$$

Ideally, if $\hat{\phi}_i$ is a good estimate of ϕ_i, then $|\phi_i - \hat{\phi}_i|$ should be small. The majority logic decoders of Paterson and Jones [11], and Van Nee [13] commit to a hard decision $\hat{c}_i$ for each information symbol c_i, based on $\hat{\phi}_i$. By a hard decision based on $\hat{\phi}_i$, we mean that $\hat{c}_i = \arg\min_{c \in \{0,1,2,3\}} |\omega^c - \hat{\phi}_i|$.

2.2. Switching to an optimal algorithm

Our Hybrid algorithm first computes the values $\hat{c}_i$, $i \in \{1, 2, 3\}$, as in majority logic. However, before committing to the hard estimates $\hat{c}_i$, the Hybrid algorithm checks how close the hard estimates are to their soft counterparts. To this end, we establish a global "sensitivity" parameter θ. If $|\arg(\hat{\phi}_i) - \arg(\omega^{\hat{c}_i})| > \theta$, for some $i \in \{1, 2, 3\}$, we discard all the estimates $\hat{c}_i$, and revert to the optimal FHT decoder for the entire block; otherwise, we commit to the hard estimates $\hat{c}_i$. Since in practice the channel amplifies the signal by some unknown gain, we choose to use the difference in phase as a reliability measure instead of the difference in magnitude.

We now address how compute the estimate $\hat{c}_0$, once we have committed to $(\hat{c}_1, \hat{c}_2, \hat{c}_3)$. We set $\phi_i = \omega^{\hat{c}_i}$, for all $i \in \{1, 2, 3\}$, and then use the equations in (1) to compute eight votes for ϕ_0. Specifically, we set $\hat{\phi}_0 = \frac{1}{8}(r_0 - r_1\phi_1^* + r_2\phi_2^* + r_3\phi_1^*\phi_2^* - r_4\phi_3^* + r_5\phi_1^*\phi_3^* + r_6\phi_2^*\phi_3^* + r_7\phi_1^*\phi_2^*\phi_3^*)$, and make a hard decision $\hat{c}_0$ based on $\hat{\phi}_0$. Otherwise, if we are not confident in the estimates $\hat{c}_i$, we throw them out and revert to the optimal FHT decoder for this block.

2.3. Experimental results

We ran our Hybrid algorithm against an optimized version of the FHT decoder for SNR from -5 to 10, and measured the running time and block error rate of both at each SNR. By block error rate, we mean the number of blocks in which at least one of the four ϕ's is decoded incorrectly over the total number of blocks. We used a value of θ such that $\tan\theta = 2/3$. This offered the best trade-off between running time and error correcting ability, since comparing to the ratio $2/3$

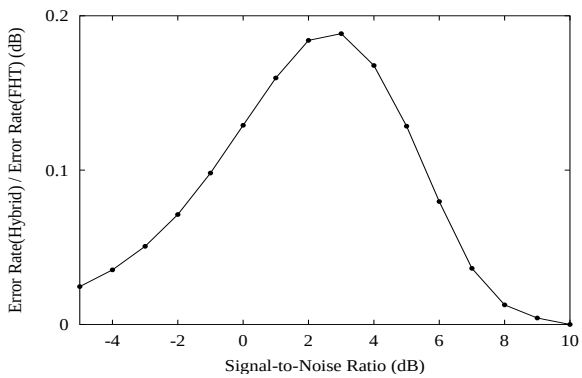

Figure 2: Loss in block error rate of the majority logic and Hybrid algorithms vs. optimal algorithm as a function of SNR. The y-axis is E_a/E_o in dB, where E_a is the block error rate of the plotted algorithm, and E_o is the block error rate of the optimal FHT algorithm.

is computationally simple. All experiments were performed on 1 billion encoded blocks of data subject to a simulated AWGN channel with varying SNR. All algorithms ran on a Pentium III 1 GHz processor.

When the SNR is high, the Hybrid algorithm runs approximately four times faster than the FHT decoder. As the SNR decreases, the frequency with which the Hybrid algorithm switches to the slower FHT decoder increases, and thus the running time increases. Figure 1 shows this relationship in detail. We remark that the Hybrid algorithm is always faster than the FHT decoder, regardless of the noise level.

The error performance of the Hybrid algorithm is near-optimal. Figure 2 shows the block error rate as a function of SNR of the Hybrid algorithm and the majority logic algorithm (without switching), as compared to the optimal FHT algorithm. Here we see that the majority logic algorithm can perform a full 2.4 dB worse than FHT, whereas the Hybrid algorithm is never more than $.2$ dB worse, making it quite close to an optimal decoder.

3. A FAST MAXIMUM-LIKELIHOOD DECODER FOR CONVOLUTIONAL CODES

In this section, we describe the *lazy Viterbi decoder* for convolutional codes.

Maximum-likelihood (ML) decoding of convolutional codes is often implemented by means of the Viterbi algorithm [14, 6, 5]. The main drawback of the Viterbi decoder is execution time. To decode a single binary information symbol, the decoder performs $O(2^k)$ operations, where k is the size of the internal memory of the encoder ($k+1$ is often referred to as the *constraint length* of the code). This exponential dependence on k makes a software implementation of

Algorithm	Best case	Worst case
Viterbi	$\Theta(2^k)$	$\Theta(2^k)$
A^*	$\Theta(\log L)$	$\Theta(2^k \log(L2^k))$
Lazy Viterbi	$\Theta(1)$	$\Theta(2^k)$

Figure 3: Asymptotic running time of three decoders in the best and worst cases. In the formulas, $k+1$ is the constraint length of the code, and L is the length of the block.

the algorithm inefficient for many codes of interest, such as the one used in the IS-95 CDMA standard for which $k = 8$. To overcome this problem, other decoder structures, namely sequential decoders [2] and A^* search [1], have been investigated in the literature. Under good Signal-to-Noise Ratio (SNR) conditions, sequential decoders are more efficient than the Viterbi algorithm, but, in addition to being suboptimal, they become prohibitively slow at low SNR [2]. The A^* decoder combines the reliability and performance of the Viterbi algorithm while running as efficiently as a sequential decoder when the SNR is high. However, previous descriptions of A^* decoders do not apply to continuous streams of data, and they do not address certain implementation problems that are critical to the practicality of the algorithm. Specifically, under high noise conditions the implementations detailed in the literature lead to a running time asymptotically even worse than Viterbi's.

In our research, we have extended the A^* approach to apply to continuous streams of data, and we have solved the implementation problems. Specifically, our *lazy Viterbi decoder* offers (i) maximum-likelihood decoding, (ii) best-case running time much better than the Viterbi algorithm, (iii) worst-case asymptotic running time no worse than the Viterbi algorithm, and (iv) simple data structures that allow for an efficient software implementation. Figure 3 summarizes the asymptotic complexity of the best and worst cases of the three algorithms.

Maximum-likelihood decoding of convolutional codes is equivalent to the computation of a shortest path on a particular directed graph called a *trellis*. A trellis node is labeled with a pair (s,t), where s represents the state of the encoder at time t. An edge $(s,t) \rightarrow (s',t+1)$ in the trellis represents the transition of the encoder at time t from state (s,t) to state $(s',t+1)$. Each edge $(s,t) \rightarrow (s',t+1)$ in the trellis is labeled with a nonnegative *branch metric* d, which measures the likelihood that the encoder moves into state s' at time $t+1$ given that the encoder is in state s at time t and given the received symbol at time t. The branch metrics can be defined in such a way that the sum of the branch metrics on a path is a measure of the likelihood of that path.

A trellis contains a distinguished *start node* at time 0. The *accumulated metric* of a node is the distance of the node from

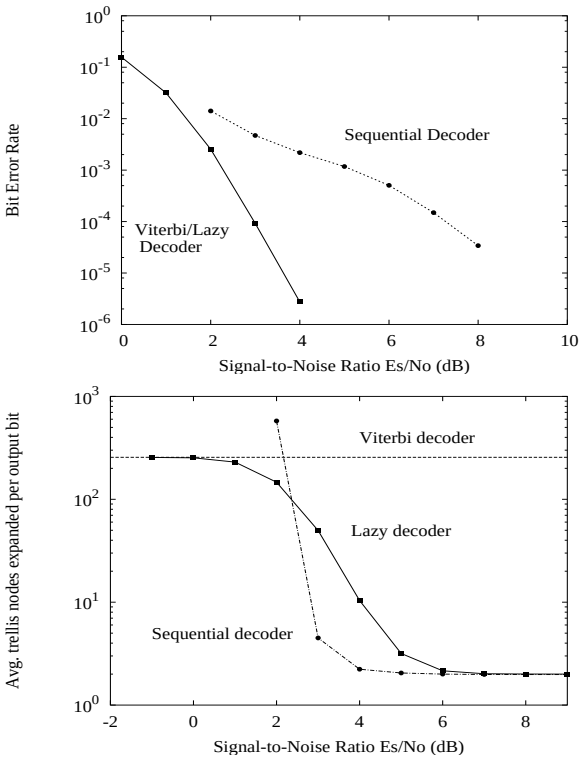

Figure 4: **(Top)** *Bit error rate;* **(Bottom)** *Average number of explored nodes per information symbol. Both are given as a function of the SNR, for the Lethargic Viterbi, Viterbi and Sequential decoders, under AWGN. The code is a rate-*$1/2$*, constraint length* 9 *code used in CDMA, generator polynomials (753,541) (octal). For the sequential decoder, experiments were performed on blocks of 100 encoded information bits.*

the start node. The goal of the decoder is to identify, for each time step t, the node at time t with the smallest accumulated metric.

Both the Viterbi and the A^* algorithm maintain an upper bound to the accumulated metric of all nodes. The basic operation is the *expansion* of a node: Once the accumulated metric of a node u is known, the upper bound of all its successors is updated. The Viterbi algorithm expands nodes breadth-first, and it expands the whole trellis no matter what the noise conditions are. The A^* algorithm always greedily expands the node with the lowest accumulated metric.

Figure 4 shows the number of expansions performed by both strategies as a function of the SNR. At high SNR, the A^* algorithm performs far fewer expansions than the Viterbi algorithm. However, it is wrong to conclude that A^* is unconditionally better than Viterbi, because in practice, expansion is much cheaper computationally for the Viterbi algorithm than it is for the A^* algorithm. The Viterbi algorithm expands every node of the trellis, and consequently it does not incur the overhead of keeping track of which nodes to expand. Moreover, for Viterbi the order of expansion is known at compile time, which allows for optimizations such as constant folding of memory addresses, efficient pipelining, and elimination of most conditional branches. In contrast, the A^* algorithm maintains a priority queue of nodes, keyed by accumulated metric. Such a priority queue slows down practical implementations of the A^* algorithm because of two reasons. First, for a trellis with n nodes, insertion, deletion and update in a general priority queue requires $\Theta(\log n)$ operations, which is asymptotically worse than the $\Theta(1)$ time per expansion of Viterbi. Second, a general priority queue using heaps or some kind of pointer-based data structure is not amenable to the compile-time optimizations that apply to Viterbi.

Our goal with the lazy Viterbi decoder is to make the A^* approach useful in practice. By exploiting the structural properties of the trellis, we can perform all priority-queue operations in constant time, thereby eliminating the $\Theta(\log n)$ slowdown. A careful design of the data structures maintained by the lazy Viterbi decoder allows us to implement the whole expansion operation in constant time, and furthermore, as a short sequence of straight-line code, which is important for efficient pipelining on present-day processors.

3.1. Speed of the lazy Viterbi decoder

In this section, we report on the running time of the lazy decoder on four different processors, and we compare our decoder with optimized implementations of the Viterbi algorithm.

Figure 5 supports our claim that the lazy Viterbi decoder is a practical algorithm. We compared the lazy decoder with the Viterbi decoder written by Phil Karn [8] and with our own optimized implementation of Viterbi. The "unoptimized Karn" decoder works for all constraint lengths and for all polynomials. Karn also provides an optimized decoder which is specialized for constraint length 7 and for the "NASA" polynomials `0x6d`, `0x4f`. This optimized code unrolls the inner loop completely, and precomputes most memory addresses at compile time. Because Karn's optimized decoder only works for constraint length 7, we programmed our own optimized Viterbi decoder that works for constraint lengths up to 6. This program is labeled "optimized Viterbi" in the figure.

Karn [9] also has an implementation that uses SSE instructions on the IA32 architecture. These instructions operate on eight array elements at the same time. Karn's SSE implementation is a great hack, as it expands one node in slightly more than one machine cycle, but it only works for constraint lengths 7 and 9. As can be seen in the table, even the eightfold gain in processing power is not sufficient to beat the lazy decoder for constraint length 9. Moreover, SSE instructions do not apply to the PowerPC processor or the StrongARM. (The PowerPC 7400 processor implements instructions simi-

Decoder	Constraint length	Athlon XP cycles/bit	Pentium III cycles/bit	PowerPC 7400 cycles/bit	StrongARM cycles/bit
Lazy	6	193	201	200	226
Viterbi Optimized	6	275	316	239	310
Karn Unoptimized	6	1041	1143	626	892
Lazy	7	198	205	203	232
Karn Optimized	7	530	558	486	641
Karn Unoptimized	7	1806	2108	1094	1535
Karn SSE	7	107	108	N/A	N/A
Lazy	9	217	235	225	343
Karn Unoptimized	9	6300	8026	3930	5561
Karn SSE	9	307	310	N/A	N/A

Figure 5: Running time of various convolutional stream decoders under high SNR conditions. Times are expressed in cycles per decoded bit. Code for constraint length 6: TIA/EIA-136 code, polynomials `0x2b`, `0x3d`. Constraint length 7: "NASA" code `0x6d`, `0x4f`. Constraint length 9: IS-95 code `0x1af`, `0x11d`. Processors: 1466 MHz Athlon XP 1700+, 600 MHz Intel Pentium III, 533 MHz PowerPC 7400, 200 MHz StrongARM 110. All programs compiled with `gcc-2.95 -O2 -fomit-frame-pointer` and the most appropriate CPU flags.

lar to SSE, but no implementation was available that exploits them.)

The running times in the figure refer to the case of high SNR, where the lazy decoder performs a minimum number of node expansions. This is the most favorable case for the lazy decoder. Our focus on the best case is legitimate because, as can be seen in Figure 4, the lazy decoder operates in the best-case scenario as long as the SNR is at least 5–6 dB, which is a reasonable assumption in practice.

3.2. Description of the lazy Viterbi decoder

The lazy decoder maintains two main data structures, called the trellis and the priority queue. The trellis data structure contains the nodes of the trellis graph whose shortest path from the start node has been computed. Each node u in the trellis data structure holds a pointer $Prev(u)$ to its predecessor on the shortest path. We maintain the invariant that every node in the trellis has been expanded.

The priority queue contains a set of *shadow nodes*. A shadow node $\hat{u}$ is a proposal to extend a path in the trellis data structure by one step to a new node u. Each shadow node $\hat{u}$ in the priority queue holds an accumulated metric $acc(\hat{u})$ equal to the length of the proposed path extension, and a pointer $Prev(\hat{u})$ to the predecessor of u on that path. Nodes $\hat{u}$ in the queue are keyed by $acc(\hat{u})$.

We note that $acc(\hat{u})$ is not stored explicitly at $\hat{u}$, but rather is implicitly stored by the data structure, a detail we will cover later. The predecessor $Prev(\hat{u})$ of a shadow node is always a "real" node in the trellis data structure. All nodes in both the priority queue and the trellis also hold their time and state value.

Initially, the trellis is empty and the queue consists of a shadow $\hat{s}$ of the start node s with $acc(\hat{s}) = 0$. After initialization, the algorithm repeatedly extracts a shadow node $\hat{u}$ of minimum metric m from the priority queue. Such a shadow node thus represents the best proposed extension of the trellis. If u, the "real" version of $\hat{u}$ with the same time and state, is already in the trellis, then $\hat{u}$ is discarded, since a better proposal for u was already accepted. Otherwise, the algorithm inserts a new node u into the trellis with $Prev(u) = Prev(\hat{u})$, and, for each successor v of u, $\hat{v}$ is inserted in the priority queue with metric $acc(\hat{v}) = m + d(u, v)$. This process is repeated until the trellis contains a node at time T.

Unlike the A^* algorithm, in our decoder a node can be both in the trellis and as a shadow in the priority queue; in fact, more than one shadow of the same node can be in the priority queue at the same time. This is one of the "lazy" features of the algorithm: Instead of demanding that all nodes be uniquely stored in the system, we trade a test for priority-queue membership for a delayed test for trellis membership. This choice is advantageous because the check can be avoided altogether if a shadow node is still in the priority queue when the algorithm terminates. Moreover, trellis membership is easier to test than priority-queue membership, as will be clear after we detail the implementation of both data structures below.

Implementation of the trellis. The trellis data structure is a sparse matrix. It is sparse because in practice only a small fraction of the trellis nodes are actually expanded (see Figure 4). It is a matrix because the two indices s and t belong to an interval of integers. Many sparse-matrix representations (including a dense matrix) could be used to represent the trellis. We found it convenient to implement the trellis as a hash table, where the pair (s, t) is the hash key. Using standard

techniques, trellis lookup and insertion can be implemented in expected constant time. In alternative, the "sparse array trick" [10, Section 2.2.6, Problem 24] could be employed for a deterministic $O(1)$ implementation of the trellis.

Implementation of the priority queue The priority queue supports two main operations: insertion of a node, and extraction of a node of minimum metric. In this section, we give a careful examination of the range of accumulated metric values taken on by shadow nodes in the priority queue. Our insights lead to an implementation that allows both insertion and extraction in constant time.

We begin by making the following assumption: *Branch metrics are integers in the range* $[0..M]$, for some integer M independent of the constraint length. This assumption holds for hard-decision decoders, where the branch metric is the Hamming distance between the received symbol and the symbol that should have been transmitted. For soft-decision decoding, this assumption requires quantization of the branch metrics. It is known [7] that quantization to 8 levels is usually sufficient to achieve most of the coding gains, and therefore this assumption is not restrictive.

This assumption implies the following property, proven in [3]: *At any time during the execution of the lazy decoder, all metrics in the priority queue are in the range* $[m..(m+M)]$, *where* m *is the minimum metric in the queue.* Because of this property, we implement the priority queue as an array $[m..m+M]$ of linked lists of nodes. The metric of a node is not stored in the node, but it is implicitly given by which list the node belongs to. The array can be implemented as a circular buffer of $M+1$ pointers. Alternatively, one can maintain the invariant that $m = 0$ by periodically adjusting all metrics when the invariant is violated. (This is a simple $O(M)$ operation that only involves a shift of the array.) In either implementation, insertion of a new node and extraction of a minimal-metric node are constant-time operations.

Stream decoding The lazy Viterbi decoder is easily adapted to process an infinite stream of data. Fix a *traceback length* $L \approx 5.8k$ as in [5]. At time T, the decoder processes a new symbol, and expands until the trellis data structure contains a node u at time T. It then outputs its best estimate of the information bit at time $T - L$ by means of a traceback process [15, Section 12.4.6]. The traceback starts at the node u, and follows a path back (using the $Pred()$ pointers) until it reaches a node at time $T - L$. It then outputs the information bit(s) associated with the first transition on the path.

After this procedure, all nodes at time $T - L$ are no longer needed, and the memory that they occupy must be reclaimed. Specifically, we must delete all the nodes from the trellis, and all the shadow nodes from the priority queue, whose time is equal to $T - L$. To this extent, we maintain a linked list of all nodes and shadow nodes at time t. We maintain an array of pointers into such *time lists* indexed by time. Since only lists in the range $t \in [T - L, T]$ are nonempty, this array can be managed as a circular buffer of length $L + 1$. After the traceback, we walk down the time list for $T - L$, deleting every node (or shadow node) along the way.

REFERENCES

[1] L. Ekroot and S. Dolinar. A^* decoding of block codes. *IEEE Trans. Comm.*, 44(9):1052–1056, 1996.

[2] R. M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Transactions on Information Theory*, IT-9:64–73, 1963.

[3] Jon Feldman, Matteo Frigo, and Ibrahim Abou-Faycal. A fast maximum-likelihood decoder for convolutional codes. In *Proceedings of the IEEE Semiannual Vehicular Technology Conference*, Fall 2002. To appear.

[4] Jon Feldman, Matteo Frigo, and Ibrahim Abou-Faycal. A noise-adaptive strategy for first-order Reed-Muller decoding. In *Proceedings of the IEEE Semiannual Vehicular Technology Conference*, Fall 2002. To appear.

[5] G. Forney. Convolutional codes II: Maximum likelihood decoding. *Inform. Control*, 25:222–266, 1974.

[6] G. D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61:268–278, 1973.

[7] J. A. Heller and I. M. Jacobs. Viterbi decoding for satellite and space communication. *IEEE Transactions on Communications Technology*, pages 835–848, October 1971.

[8] Phil Karn. KA9Q Viterbi decoder V3.0.2, `viterbi-3.0.2.tar.gz`. `http://people.qualcomm.com/karn/code/fec/`, October 1999.

[9] Phil Karn. SIMD-assisted convolutional (Viterbi) decoders, `simd-viterbi-2.0.3.tar.gz`. `http://people.qualcomm.com/karn/code/fec/`, February 2002.

[10] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 2nd edition, 1973.

[11] K. Paterson and A. Jones. Efficient decoding algorithms for generalised Reed-Muller codes. Technical report, Hewlett-Packard Labs, November 1998.

[12] Bob Pearson. Complementary code keying made simple, application note 9850, May 2000. `http://www.intersil.com/data/an/an9/an9850/an9850.pdf`.

[13] R. van Nee. OFDM codes for peak-to-average power reduction and error correction. In *Proc. IEEE Globecom '96, London, England*, pages 740–744, November 1996.

[14] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory*, IT-13:260–269, April 1967.

[15] S. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice-Hall, Englewood Cliffs, NJ, 1995.