Convolutional Decoders for Amateur Packet Radio

Phil Karn, KA9Q

Abstract

This paper describes two freely available convolutional decoders written in the C language. One implements the Fano algorithm for sequentially decoding three rate l/2 K=32 codes while the other implcmcnts the Viterbi algorithm for maximum likelihood decoding of the rate l/2 K=7 NASA Standard code. Both support 8-W soft-decision decoding with arbitrary metric tables and perform according to theory.

IJsing GCC 2.63 under BSDI/OS 2.0 on an AMD 486DX4-100 CPU, the Fano decoder runs up to 375 kb/s while the Vitcrbi decoder runs at a constant speed of 75.1 kb/s. On a 90 MHz Intel Pentium, the speeds are 594kb/s (Fano) and 118 kb/s (Viterbi). This is fast enough to be useful in many amateur packet radio applications.

The code is available over the Internet. See this web page for details: http://www.qualcomm.com/people/pkarn/ham.htm1

Introduction

C'onvolutional coding is one of the two major types of error control coding (the other is block coding, e.g., Golay, R.ecd-Solomon). A convolutional encoder is very simple. It consists of a K-stage shift register into which user data is clocked. The value K is called the constraint length of the code. The shift register is tapped at various points according to the code *polynomials* chosen. Several tap sets are chosen according to the code rate given as a fraction, e.g., l/2. The denominator of the fraction (2 in this case) is the number of code polynomials while the numerator (1) tells how many user data bits arc shifted into the register per encoder cycle.

Each set of tapped data is summed modulo-2 (XORed) and becomes one of the encoded output symbols. For example, in the K.=7 rate l/2 code I use in my Viterbi decoder, user data is clocked into the 7-stage shift register one bit at a time. The first encoded symbol is then formed by modulo-2 summing the outputs of shift register stages 1, 3,4,6 and 7, while the second symbol is formed by modulo-2 summing stages 1,2, 3,4 and 7. Then the next data bit is clocked in and the cycle repeats.

Convolutional encoders are extremely simple; that's one reason they're so popular on deep space probes. Convolutional &co&g takes more work. Basically, all convolutional decoders work by generating local hypotheses about the originally transmitted data, running those hypotheses through a local copy of the encoder, and comparing the encoded results with the (noisy) encoded data that is actually received. A decoder can pursue one hypothesis at a time as in the scquenlial decoder, or it can pursue many in parallel as in the Viterbi decoder. Both decoders compute *metrics* as estimates of the "closeness of fit" between the received and locally regenerated symbols, and they provide cumulative path metrics with the decoded packet as a useful estimate of the decoder's "confidence" in the result.

Unlike block codes, convolutional decoders can easily use *soft decision* information from the demodulator about the quality 01:' each received symbol. They easily handle variable packet sizes such as those found in computer networks, and they're fairly easy implement in software (though they don't necessarily run fast!)

This is not to say that convolutional codes are always superior to block codes. Block codes have some complementary advantages, such as the ability to handle larger data blocks and a greater ability to handle burst errors. The two are sometimes *concatenated* to combine their strengths, as in the *Voyager* and *Galileo* spacecraft.

But one has to start somewhere, so I implemented some convolutional decoders to see if they could run fast enough on personal computers to be useful in amateur packet radio. I believe I have been successful.

Development Environment

Error control decoding can take a lot of CPU cycles. My Fano and Viterbi decoders arc respectively 1 to 2 orders of magnitude slower than my DES encryption code, which is usually thought 01' as very compute-intensive. Careful optimization thus bccomcs extremely important to overall system performance.

I use BSDI/OS, a commercial port of Berkeley (BSD) UNIX to the 32-bit Intel processors (386/486/Pentium) and a close cousin of FrccBSD and NetBSD. BSDI supports a pure protected-mode 32-bit environment that is much more efficient for FJZC than the 16-bit "real mode" provided for DOS backward compatibility.

The performance gains from a 32-bit environment are simply too great to pass up. Given that the 386 is already virtually obsolete and the 486 is now on its way out, this doesn't seem like an unreasonable assumption. My code does not assume BSDI; it should run well in other 32-bit environments, including 32-bit DOS extenders. Since it is in portable C, it should also execute in 16- and &bit environments, but I expect it would be rather slow.

The Fano Decoder

My first coding project was a sequential decoder using the Fano algorithm. Sequential decoders explore one hypothetical data sequence at a time by locally encoding it and comparing it with the noisy encoded version that is actually received. As long as they match reasonably well, the decoder moves forward. When the received and local versions differ by more than a certain amount, the decoder backs up, tries an alternative data hypothesis and moves forward again to see if it works better. The process repeats until the decoder reaches the end of the packet or exceeds some time limit.

I chose a code with a constraint length K of 32. Sequential decoders can easily handle such large constraint lengths, and 32 was a natural choice given the CPU word size. I originally chose the NASA Standard polynomials used by the Pioneer 10 and 11 probes. I then found two other polynomials in the literature with slightly better performance, so I added these as compile-time options.

Sequential decoders have been heavily analyzed, and their behavior is well understood. As might be expected, decoding time is a random variable that depends strongly on the quality of the received signal. When the Eb/NO is several dB more than the decoder threshold (2.5 dB for this code), decoding quickly proceeds at a nearly constant rate. (See fig 1). Near the threshold, the mean decoding time and its variance increase. (See fig 2.) Below the threshold the variance becomes infinite and many packets "time out" the decoder. (See fig 3).

An import;lnt figure of merit for any Fano decoder is its "computational rate", the speed at which it can examine a branch in the code tree and make a decision to move forward, backward or sideways in the tree. On a "clean" packet the decoder rapidly moves forward, but a noisy packet causes many backward and sideways moves that slow down the process.

Optimizing the Fano decoder

Optimizing the Fano decoder involves a tradeoff between "clean" and noisy packets. I decided to optimize for noisy packets, as long as the resulting hit for clean packets wasn't too great. After all, clean packets already decode quickly. Noisy packets need the most help. Some optimizations required additional memory, but on modern systems these were no-brainers.

The main part of the Fano decoder is a loop that executes on every forward motion. Imbedded in this loop is a smaller loop that does backward motion when needed. Optimizing this code was fairly straightforward. It consisted mainly of "factoring out" operations from the loop whenever possible so they're executed only once per packet no matter how many moves the decoder makes. For example, all possible metrics for each branch (four for this code) are computed before going into the loop, even though only two are actually examined on any particular forward decoder motion. Although this represents a little extra work on a clean packet, this helps substantially on a noisy packet by avoiding the repeated recomputation of branch metrics.

Another optimization is to stash quite a bit of information into the data structure that represents a node in the decoding tree. Each node element includes the decoder state and cumulative metric along with the list of all possible branch metrics just mentioned. This lets the decoder move rapidly backward by simply decrementing a pointer, without having to recompute anything. This improves performance on noisy packets at the cost of extra memory and a slight reduction of decoding speed on clean packets.

The Viterhi Decoder

In contrast to the sequential decoder, a Viterbi decoder lv@cully explores in parallel every possible user data sequence. It encodes and compares each one against the received sequence and picks the closest match. For this reason, it is also known as the *maximum likelihood* decoder.

The decoder can't actually do this, of course, since the number of possible data sequences doubles with each additional data bit. But the Vitcrbi decoder recognizes at each point that certain sequences cannot possibly belong to the maximum lilkclihood path, and it discards them. This happens because the encoder memory is limited to K bits; a Viterbi decoder in steady-state operation keeps only 2*(K- 1) paths.

The complexity of a Viterbi decoder therefore increases exponentially with the constraint length K. Larger K's do perform better against noise, but a Viterbi decoder for a K=32 code is just not practical. For my Viterbi dccodcr I chose the rate l/2 K=7 NASA standard convolutional code used on Voyager and in many commercial satellite systems.

Tlhis shorter code has less coding gain than the K=32 sequentially decoded code, but Viterbi decoding has the practical advantage of executing at a constant speed regardless of the received signal-to-noise ratio. This makes it attractive for real time, delay-limited applications that can tolerate some uncorrected errors, e.g., digital speech. Viterbi decoding is also more tolerant than sequential decoding to metric table and receiver AGC errors. And the inherent parallelism of the Viterbi decoder makes it easy to implement in hardware, an important consideration when contemplating very high speeds. It is not yet clear, though, that these advantages outweigh the slower average execution speed when implemented in software in a packet radio system. Experimentation with both is warranted.

The structure of the Viterbi decoder is very different from the Fano decoder. The Fano decoder consists of several pages of fairly "random" C code. But the Viterbi decoder consists almost entirely of a simple Add/Compare/Select (ACS) operation that is executed repeatedly for each decoded data bit -- 64 times per bit for a K=7 code.

As the name implies, an ACS operation adds a branch metric to each of two cumulative path metrics that convcrgc into a node, compares them and selects the "surviving" (best) path. This implements the "discard paths that can't possibly be right" feature at the heart of Viterbi decoding.

Bly analogy, consider a highway routing problem. If you determine that the route from Baltimore to New York via Philadelphia is shorter (better) than the route that goes through Pittsburgh, then you don't really have to cons&r the Pittsburgh route at all when extending your trip beyond New York to Boston. The same thing happens in distance-vector routing algorithms such as RIP and NET/ROM: each node only propagates its best routes to its neighbors.

An important design parameter for a Viterbi decoder is the path memory length. An ideal decoder would keep every possible path in memory and delay a final decision about the first bits of the packet until the very end. But this isn't really necessary. Analysis and simulation have shown that several constraint lengths back, the paths usually merge into a single maximum-likelihood candidate. So a Viterbi decoder that retains 4-5 constraint lengths of decoded data will achieve nearly the same uncorrected error rate as an ideal decoder.

For a K=7 code, a 32-bit path memory is a nice match to a 32-bit word size. The ACS operation can record the surviving path at each node by simply copying the single machine word containing it. This is fast and efficient since 32-bit operations on a 32.-bit machine are no more costly than smaller operations.

Optimizing the Viterbi Decoder

ACS operations are often paired into "butterflies", so called because of their appearance on a trellis diagram. There are 32 butterflies per data bit for a K=7 code. As you might guess, the butterfly macro was the focus of almost all of our optimization efforts.

Optimization focused on two goals: minimizing the number of CPU cycles per butterfly and maximizing the Ll (on-chip) calche hit ratio. This latter goal is especially important on the 486DX2 and DX4 since external bus cycles arc proportionately even more expensive in a clock-multiplied CPU.

Originally I executed the butterflies in a tight loop, 32 loops per data bit, giving the expected poor pcrformancc. The most dramatic speedup came from unrolling this loop, but not for the reason you might suspect. The usual reason to unroll a loop is to amortize the loop control overhead over more instructions. This contributed a little, but the big win came from something clsc.

The ACS operations work on an array of node elements. There are two such arrays, one for the decoder's current state and another for the new state after the current symbols have been processed. After each reccivcd symbol pair is processed, the two arrays are swapped. (Actually, pointers to these arrays Lare swapped). Every 8 bits, the path with the best metric is chosen

and the high order 8 bits are extracted and placed in the output buffer.

Each butterfly works on a fixed set of old and new state array elements and a fixed pair of hypothesized branch symbols. The original code computed the addresses of each array element and the branch symbols as functions of the loop index, which meant doing it repeatedly at execution time. But when the loop was unrolled, constants replaced the loop index in these expressions. This allowed them to be completely evaluated at compile time and replaced with immediate constants in the generated code, giving the dramatic speedup.

Around this time I upgraded my CPU from the clock-doubled 486DX2-66 to a clock-tripled 486DX4-100. (The latter really should have been named the 486DX3-99...) On many CPU-intensive operations this new chip gives nearly the 50% improvement you'd expect from the ratio of internal clock speeds. This was true for my DES code and for the Fano decoder, but the Vi terbi decoder only sped up by about 15%.

A little analysis revealed why. The 486's on-chip cache operates in write-through mode. For programs that only occasionally write to memory, this isn't a problem because the CPU also has a 4-deep memory write queue. As long as there's room, the CPU can schedule a write and continue execution. But if the queue is full, the CPU blocks until space opens up.

Each Viterbi ACS reads five memory words: two node metrics, two branch metrics and the surviving path. It also writes two words to the new node: the survivor's path and metric. This is an unusually high ratio of memory writes to memory reads. The reads generally hit in the cache, but the writes go through to main memory. If there's a bottleneck in writing to main memory, this would explain why the DX4-100 isn't much faster than the DX2-66: the external memory bus is no faster for the faster CPU.

Unfortunately there is no easy way to disable write-through caching on the 486. Fortunately, it's easy to do on the Pcntium; most systems app'arently support write-back caching. This is a big win in Viterbi decoding because the data written to cache in one cycle is immediately read in the next and overwritten in the one beyond that. There's no reason to go to main memory at all.

Not having a Pentium, I stopped further optimization efforts. Franklin Antonio N6NKF, who does have a Pentium, picked up the code and tried a few more tricks. He examined the assembler code generated by his compiler (Watcom) and discovered that by reordering the butterflies to place together those operating on the same pair of branch symbols, the compiler did a better job of allocating registers and keeping common subexpression results involving the branch metrics. This produced a noticeable speedup that to my surprise carried over somewhat to the 486.

Acknowledgements

I would like to thank Franklin Antonio, N6NKF, for the considerable time he spent optimizing the Viterbi dccodcr code on his Pcntium, and for pointing out some bugs.

Appendix - sample decoder performance

This section presents performance results for the two decoders. In the Fano section I present information about the decoder's variable speed while in the Viterbi section I show the uncorrected error performance. (Since Fano dccodcrs on long constraint length codes produce essentially no errors and Viterbi decoders execute at a constant rate, there's little point in the reverse set of statistics.)

These results assume nonfading additive white gaussian noise (AWGN) channels and ideal BPSK or QPSK modems. I used the classic "rejection method" (see *NumericaLRecipes in* C) to produce normally distributed (gaussian) noise samples from a conventional unifoml random number generator.

Keep in mind that without coding, ideal BPSK and QPSK both require an Eb/NO of about 9.6dB for a bit error rate of lc-5.

Sample Fano decoder performance

Statistics from several runs of the Fano decoder follow. To test the effects of metric table inaccuracies, the driver program allows for different values of Eb/NO to be used in the computation of the metric table than for the actual generation of the noisy test signal. Here the same values are used for each. The threshold adjustment parameter delta is an internal tuning

parameter of the dccodcr that depends on the scaling of the metric tables; delta.=17 was found to work well by cxpcrimcnt.

The value N represents the number of forward decoder motions per data bit required to decode a particular packet; for each run, both an average and a cumulative histogram are given. For example, in the Eb/NO = 3dB run the average number of fir>rward decoder motions per user data bit was 2.46, while 2.2% of the packets took 6 or more motions per bit. To estimate tlhe throughput of the decoder at a given Eb/NO, simply divide the rate for "clean" packets (375 Kb/s on the 486DX4-100) by tlhe average N for that Eb/NO. This ignores the cost of backward motions, but they're relatively cheap.

The decoder imposes a limit on N for each packet; if this limit (10000 for these runs) is exceeded, the packet is erased (discarded). In Fig 3 note that even though 33% of the packets were erase.d, none were decoded in error. This is a nice side bcncfit of long constraint length codes: separate error correction mechanisms like CRCs are not really required.

Figure 1 - Fano decoder performance 2.5 dB above threshold

```
Seed 806575651 Amplitude 100 units, Eb/NO = 5 dB metric table Eb/NO = 5 dB
F'rame length = 1152 bits, delta = 17, cycle limit = 10000, #frames = 1000
decoding errors: 0
Average N: 1.181747
  N >=  count  fraction
     1  1000  1
     2    0   0
```

Figure 2 - Fano decoder performance 0.5 dB above threshold

```
Seed 806575707 Amplitude 100 units, Eb/NO = 3 dB metric table Eb/NO = 3 dB
Frame length = 1152 bits, delta = 17, cycle limit = 10000, #frames = 1000
decoding errors: 0
Average N: 2.464074
  N >=  count  fraction
     1  1000  1
     2   540  0.54
     4    71  0.071
     6    22  0.022
     8     9  0.009
    10     6  0.006
    20     3  0.003
    40     0  0
```

Figure 3 - Fano decoder performance 1.5 dB below threshold

```
Seed 806575769 Amplitude 100 units, Eb/NO = 1 dB metric table Eb/NO = 1 dB
Frame length = 1152 bits, delta = 17, cycle limit = 10000, #frames = 1000
decoding errors: 0
Average N: 561.711378
  N >=  count  fraction
     1  1000  1
     2  1000  1
     4  1000  1
     6  1000  1
     8  1000  1
    10   997  1
    20   977  0.98
    40   929  0.93
    60   890  0.89
    80   859  0.86
   100   837  0.84
   200   762  0.76
   400   677  0.68
   600   620  0.62
   800   576  0.58
  1000   556  0.56
```

```
  2000     489  0.49
  4000     413  0.41
  6000     378  0.38
  8000     351  0.35
 10000     326  0.33
```

Sample Viterbi decoder performance

This section shows the results from a typical run of the Viterbi decoder. The Eb/NO here is 3.5 dB; this value was chosen more to produce a reasonable amount of output than anything else. The encoded data was all zeroes, so errors show up as “1” bits in the decoded data. (Decoded data is not shown for the 97 frames that decoded correctly). Note the ch‘aracteristic bursts in which the errors occur, even though the channel is AWGN and stationary. Burst errors are well suited to Reed-Solomon block codes, which is why Reed-Solomon and Viterbi-decoded convolutional codes are sometimes concatenated.

```
decoded  data:
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000330000000000000000
00000000000000000000000000000000
decoded  data:
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
002a240000000000000000000000000
decoded  data:
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000010000000000000000
Seed 806709882 Amplitude 100 units, Eb/NO = 3.5 dB
Amplitude 100 units, Eb/NO = 3.5 dB
Frame length = 1152 bits, #frames = 100, bits decoded = 115200
frame errors:  3 (0.03)
bit errors:  10  (8.68056e-05)
```