Implementation of Long Constraint Length Viterbi Decoders using Programmable Active Memories

R A Keaney, L H C Lee, D J Skellern

School of Mathematics, Physics, Computing and Electronics
Macquarie University, NSW 2109, Australia
Email: richardk@mpce.mq.edu.au

J E Vuillemin, M Shand

Digital Equipment Corporation Paris Research Laboratory
85 Ave Victor Hugo, Rueil–Malmaison 92563 Paris, France

This paper describes the implementation of a 2 kbit/s Viterbi decoder for a rate 1/4 constraint length 15 convolutional code using the DECPeRLe–1 system. The DECPeRLe–1 system comprises a reconfigurable coprocessor board based on a matrix of Xilinx Field Programmable gate arrays and a suite of CAD tools to build designs using C++ as a hardware description language.

1. Introduction

In many of today's digital communication systems the signal to noise ratio (SNR) of the link has become the most severe limitation. Convolutional encoding with Viterbi decoding provides a means to improve the SNR of the link without increasing the power budget, and has become an important technique in satellite and deep space communications systems. The coding gain of a system is primarily determined by the constraint length (K) of the code. The complexity of a Viterbi decoder increases exponentially with the constraint length of the code. Single chip VLSI codecs for K = 7 and K = 8 codes are commercially available, such as the one described in [1]. NASA has included a K = 15 encoder on the Galileo space probe, for which a Viterbi decoder based on large arrays of custom VLSI chips has been developed [2]. This paper will present an alternative implementation of the Viterbi decoder using Programmable Active Memories (PAMs).

A PAM is a uniform array of identical cells which may be programmed to generate any synchronous finite state machine, provided the array is large enough, and the clock speed is slow enough [3]. The DECPeRLe–1™ system developed by staff at the Digital Equipment Corporation Paris Research Laboratory (PRL) is a PAM implementation based on an array of Xilinx™ Field Programmable Gate Arrays (FPGAs). The DECPeRLe–1 system interfaces to a sophisticated CAD system resident on a DECstation™ 5000 workstation. Users implement their system using C++ as a hardware description language. The CAD system flattens the design and converts it to a bitstream to program the hardware configuration. The system offers the advantage of software flexibility, through its reconfigurability, with the speed of a hardware implementation.

Section 2 of this paper provides a short overview of convolutional encoding and Viterbi decoding. Section 3 provides an outline of the DECPeRLe–1 system. Section 4 details the implementation of the Viterbi decoder using the DECPeRLe–1 system and Section 5 concludes with test results obtained for the decoder and compares the performance of the decoder to the performance predicted by simulation.

2. Convolutional encoding/Viterbi decoding

Convolutional codes are generated by the modulo 2 addition of n input bits with m previous bits to produce r output bits. m is called the memory order of the code, the ratio of n/r is called the code rate, and $n + m$ is called the constraint length of the code. Figure 1 outlines the rate 1/4, constraint length 15 encoder used on the Galileo space probe. The coding gain of a code is a function of the code rate, the constraint length, and the generator polynomials. The generator polynomials of a code are the sequence of taps of the shift register used to generate each of the r output bits of the codeword. There is no analytical method to prove that one set of polynomials will provide the best coding gain, and polynomials are usually selected by exhaustive trial. This becomes infeasible for very long constraint length codes as the simulation time required to try all possibilities becomes prohibitive. Performing code searches is an important application for a re–configurable hardware decoder.

The Viterbi algorithm is a maximum likelihood decoding algorithm for convolutional codes [4]. A detailed description of the algorithm is beyond the scope of this paper, rather a description in terms of the major blocks required for a hardware implementation will be given. The 3 major blocks of the decoder are:

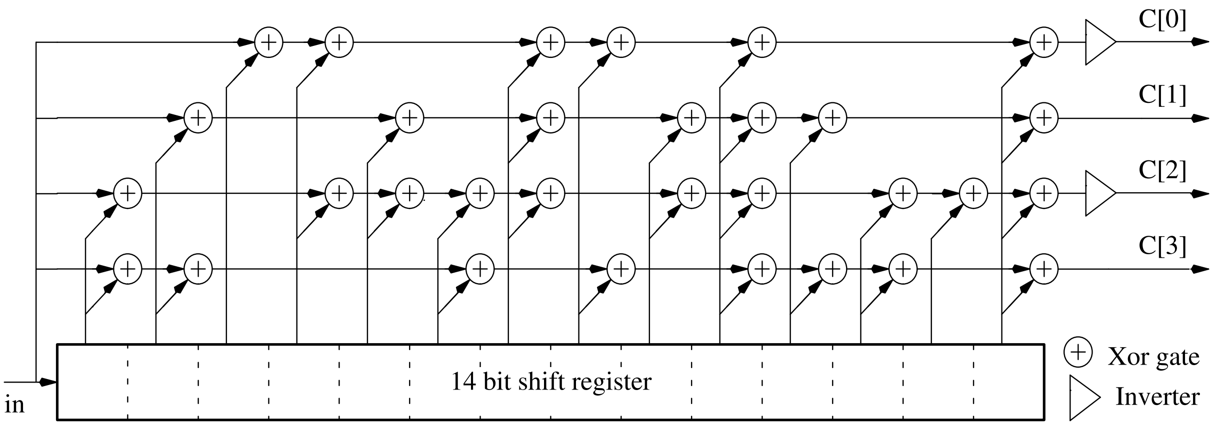

Figure 1. The rate 1/4 K=15 Convolutional Encoder used on the Galileo Space Probe

1. The branch metric (BM) calculator.
2. The state processors.
3. The decision memory.

The decoder accepts 4 inputs from the receiver, R[0..3], which correspond to the 4 transmitted bits of the encoder output codeword C[0..3]. R[0..3] may be single bit inputs but are more often 3–bit or 4–bit quantised values from the receiver. The BM calculator calculates an upper and lower branch metric which assign the probability that the current received codeword corresponds to the output codeword of the encoder with an input bit of 0 or 1 respectively for each of the 2^m (ie 16,384) possible states of the encoder shift register. The simplest algorithm for metric generation is to calculate the linear distance between the received codeword and the transmitted codeword. The BM range for such a scheme is $r(q-1)$ where q is the number of quantisation levels used for the decoder inputs symbols. The BM range strongly affects the number of bits required in the state processors, and is an important parameter to minimise in a VLSI design of a decoder. Using the Euclidean rather than linear distance will yield a BM range of $r^{0.5}(q-1)$, reducing the metric range by 1 bit for a rate1/4 code. It has been shown by simulation that a further bit can be saved from the branch metrics by using a non linear look–up table for the metric calculation, without loss of coding gain.

The state processors accumulate branch metrics over time and select the survivor (ie minimum) metric for each state at each time period. The state is the m bit binary number contained in the encoder shift register, where the most recent (ie leftmost bit) of the register is the least significant bit. As time advances the bits are shifted right through the register and the state advances from j to $(2j)\mathrm{mod}(2^m) + i$, where i is the next input bit. This leads to the trellis structure shown in Figure 2. The example shown is for a 16 state (ie K=5) code.

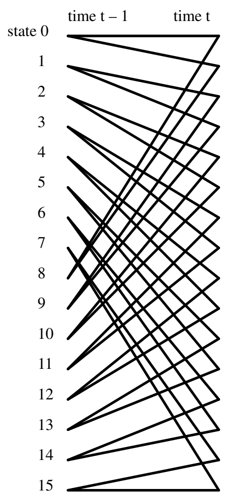

Figure 2. A 16 State Code Trellis

The upper branch from each state at time $t-1$ to the next state at time t represents a transmitted 0 and the lower branch a transmitted 1. The state processor for state j at time t adds the accumulated metric (known as the path metric, PM) from state $j/2$ at time $t-1$ to the upper BM for state j and adds the PM from state $j/2 + 2^{m-1}$ at time $t-1$ to the lower BM for state j and selects the smaller of the 2 sums as the new output PM for state j at time t. This process is known as Add–Compare–Select and the state processors are often referred to ACS processors. For a memory order m code 2^m ACS processors are required. The ACS processor also produces a decision output based on whether the upper or lower branch into the state was the survivor metric, where 0 signifies the upper branch was the survivor and 1 the lower branch was the survivor.

The path metrics will grow infinitely large over time, however it can be shown that the minimum PM range (ie the difference between the largest and smallest PM at any time period) is m(BM range). The two most common methods for coping with metric growth are to rescale when the metrics have grown to beyond the minimum range or to add a sign bit and use 2's complement arithmetic in the ACS processors.

The decision memory stores the 2^m decision outputs from the state processors at each time interval. By tracing back through all the decisions for a time period d, known as the decision depth, correct bits can be decoded because all the minimum survivor paths will have merged onto the one (hopefully) correct path. The decision depth has been shown by simulation to be about 5 times the constraint length if the traceback is started from the state with the minimum survivor metric, and about 7 to 8 times the constraint length if the traceback is started from an arbitrary state.

3. The DECPeRLe–1 System

The DECPeRLe–1 system is a programmable hardware accelerator based on a matrix of Xilinx Field Programmable Gate Arrays (FPGAs) attached to a DECStation 5000 Ultrix™ workstation[5]. Figure 3 outlines the

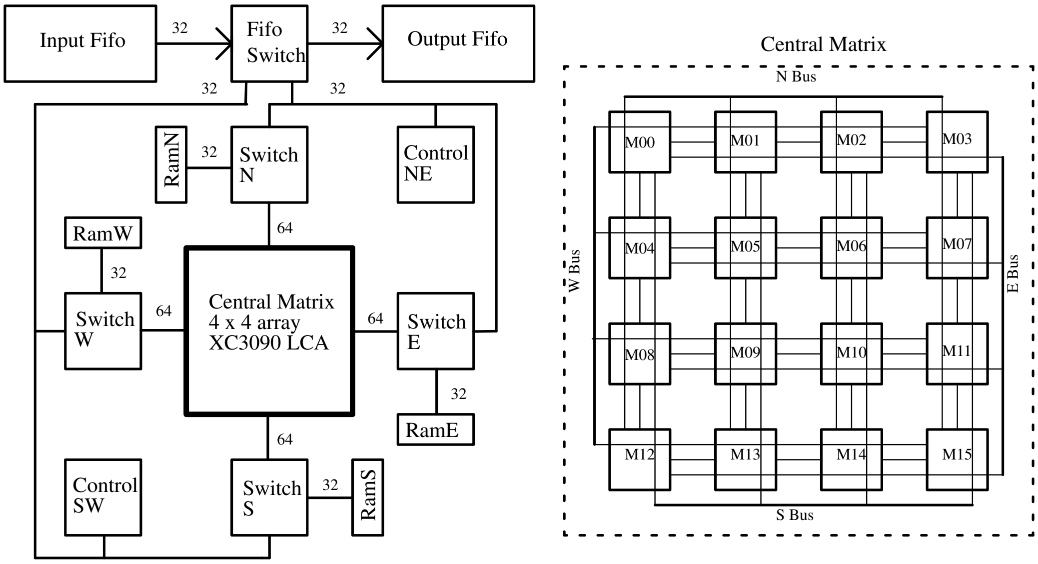

Figure 3. The DECPeRLe–1 System Architecture

architecture of the system. The central matrix consists of 16 Xilinx XC3090 Logic Cell Arrays™ (LCAs), connected to four 64–bit buses, shown as the North (N), South (S), East (E) and West (W) matrix buses. These buses are connected to the N, S, E and W data switches, which are also XC3090 LCAs. Each switch is also connected to a bank of 32–bit by 256k word high speed SRAM. The N and E switches are linked by another 32–bit bus, as are the S and W switches. Each of these buses connects to a 5th switch LCA, known as the Fifo Switch, which communicates to the host workstation via 32–bit FIFOs connected to a TurboChannel™ expansion slot. Finally 2 more LCAs are included to control the N and E memories (Controller NE) and the S and W memories (Controller SW). The XC3090 LCA consists of a 16x20 array of configurable logic blocks (CLBs), where each CLB has 2 flip–flops and a 5 input programmable logic array. The central matrix of LCAs thus forms a 160x64 array of bit programmable logic elements.

The system is programmed using C++ as a hardware description language. A C++ library is provided which contains low level primitives for describing designs [6]. Logical nets are described using Boolean variables. Boolean equations are described using the standard C syntax for bitwise operators. The library provides the *reg* and *mux* primitives to describe a D–type flip flop and a 2 input multiplexer, which together with the C language operators allow any CLB configuration to be described. Additional primitives are supplied to assign specific CLB pins to specific nets and declare the usage of tri–state buffers and clocks. A special operator to assign the placement of nets to CLBs is also provided. Hierarchical designs are constructed using C functions and C++ class declarations, and a standard template file is provided to map a design into the board architecture.

4. Implementation of the Decoder

A constraint length 15 convolutional code has 2^{14} states, requiring 16,384 ACS operations to be performed for each input codeword. The decoder architecture time multiplexes the use of a small number of fast, pipelined, bit parallel ACS processors. The main constraint on the architecture is the memory bandwidth required to retrieve the PMs from the previous codeword and store the new PMs for the current codeword. The absolute minimum PM range required for a rate 1/4 constraint length 15 code using 4–bit quantised inputs is 8 bits, however this requires metric rescaling. The hardware implementation is greatly simplified if an extra sign bit is used and 2's complement arithmetic employed in the ACS processors. This requires a minimum of 9 bits for the PMs.

Inspection of Figure 2 shows that adjacent states of the trellis share the same input PMs and that each state requires an input PM from the top half and an input PM from the bottom half of the trellis. Hence it is logical to store the PMs for states 0 to 8,191 in one memory (the West RAM) and the PMs for states 8,192 to 16,383 in another memory (the East RAM). This allows both input PMs for a state to be accessed simultaneously. The memory word size is 32 bits, allowing four 8–bit or two 16–bit PMs to be stored in a word. As the minimum PM size required was 9 bits, it was decided to use 16–bit PMs, and hence 16–bit arithmetic in the ACS processors. This allows plenty of headroom to implement higher rate and/or longer constraint length codes in the future. Accessing 2 input PMs from the top half of the trellis and 2 from the bottom allows 4 states to be calculated in parallel, requiring 4 ACS processors and 4 sets of branch metrics. However the 4 ACS processors produce 4 new output PMs which must be stored in memory. As the output PMs are from adjacent states they must be written to the same memory bank, which will require 2 write accesses to store the four 16–bit output PMs in adjacent 32–bit words. From this the basic calculation cycle is 1 read access to retrieve the input PMs for all 4 states, followed by the ACS calculation, followed by 2 write accesses to store the output PMs. The system is heavily pipelined so the ACS calculation for 1 group of 4 states is carried out whilst the input PMs are being retrieved for another group of 4 states and the output PMs are being stored for a previous group of 4 states. The memory access time is 40ns yielding a basic cycle time of 120ns to compute 4 states in parallel. This yields a throughput of 1/(16,384/4 x 120ns) $\approx$ 2kbit/s.

Figure 4 shows the outline of a pair of ACS processors. Pairs of ACS processors are grouped together as they

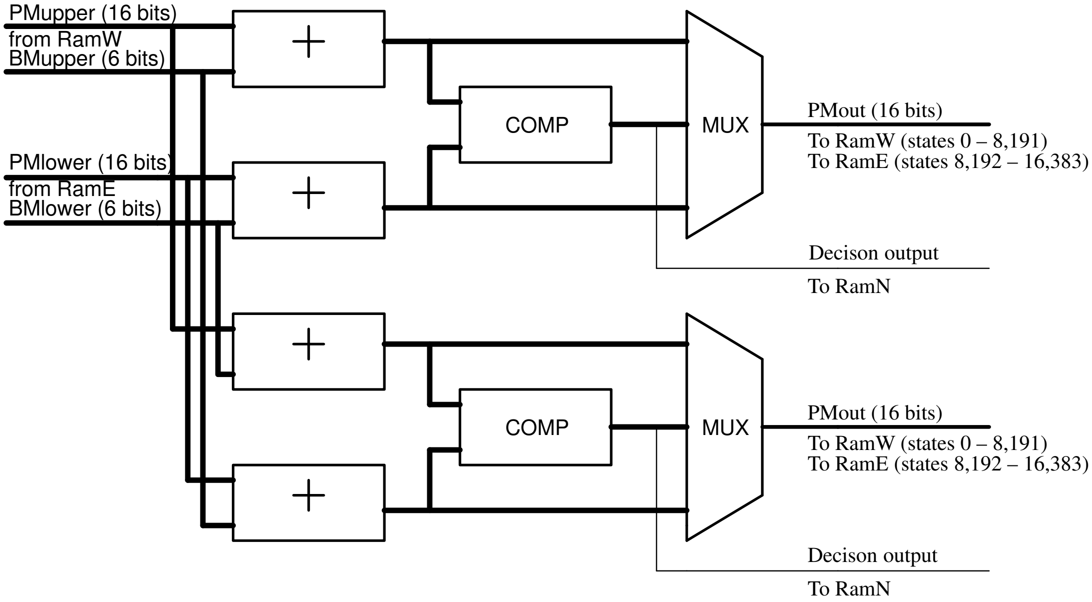

Figure 4. A Pair of ACS Processors

share input PMs. Most convolutional codes, including the one used on the Galileo space probe, have generator polynomials which are a function of the MSB of the shift register for *every* bit of the output codeword. This means the output codeword for state j is the inverse of the output codeword for state $j + 2^{m-1}$. This leads to the conclusion that adjacent states can share branch metrics as well, with the lower BM for state j being equal to the upper BM for state $j + 1$ and vice versa. One pair of ACS processors is implemented bit interleaved in LCA M01 and LCA M05 and calculates states $4j$ and $4j + 1$, $0 \leq j < 4{,}096$. The other pair is implemented in M09 and M13 and calculates states $4j + 2$ and $4j + 3$. The upper PMs are stored in Ram W and fed to the ACS processors via Switch W. The lower PMs are stored in Ram E and fed to the ACS processors via Switch E.

The design was implemented with a maximum combinatorial logic chain of 4 CLBs between pipeline registers. This ensures that the calculation time remains below the memory access time of 40ns. The adders employ a 4 stage pipeline. The comparator is identical to the adders except the upper input is inverted and the initial carry into the LSB is set to one. The MSB of the of the comparison is used as the select input to the 16–bit 2 way multiplexer, and also forms the decision output from the processor.

Separate branch metric calculators are required for ACS processors 0 and 1 (BM01) and ACS processors 2 and 3 (BM23). BM01 is implemented in LCA M00 and BM23 in LCA M08. Each BM calculator incorporates a configurable rate 1/4 constraint length 15 encoder which consists of a 14–bit state register and four 15–bit mask registers. The mask registers define which taps of the state register are used to generate the four output bits of the codeword, with the 15th bit of each register defining the inversion of the output bit. The mask registers are statically defined for the Galileo code, but can be configured to any arbitrary code required. The state register is connected to a 12–bit counter which cycles through each of the 4,096 states for which the BM calculator is generating metrics.

The upper BM is calculated by adding the 4 received codewords. The received codewords are first inverted if the corresponding output bit of the encoder for that state is set. The lower branch metric calculation is identical except the complement of the encoder outputs are used. For example, at state 0 the encoder output C[0..3] is 1, 0, 1, 0. If the received codeword R[0..3] is 14, 6, 9, 2 then the upper BM is equal to 1 + 6 + 6 + 2 = 15 and the lower BM is equal to 14 + 9 + 9 + 13 = 45. Figure 5 outlines the circuit for one of the BM calculators.

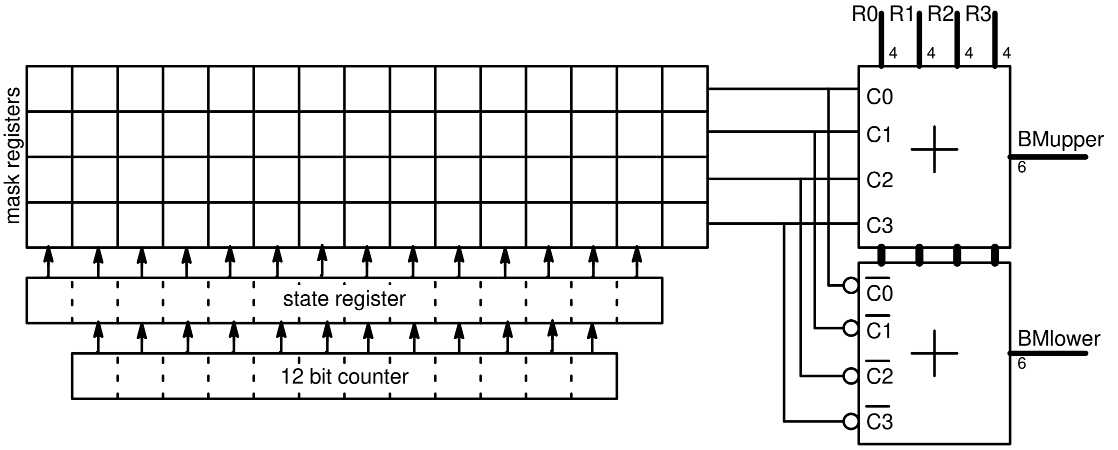

Figure 5. Branch Metric Calculator

The decision outputs from the 4 ACS processors are grouped into 32–bit words in the North Switch and written to the decision memory located in Ram N. The decision memory is logically arranged as a 256 column by 16,384 row memory as shown in Figure 6, where each column contains 512 words of 32 bits each. The write

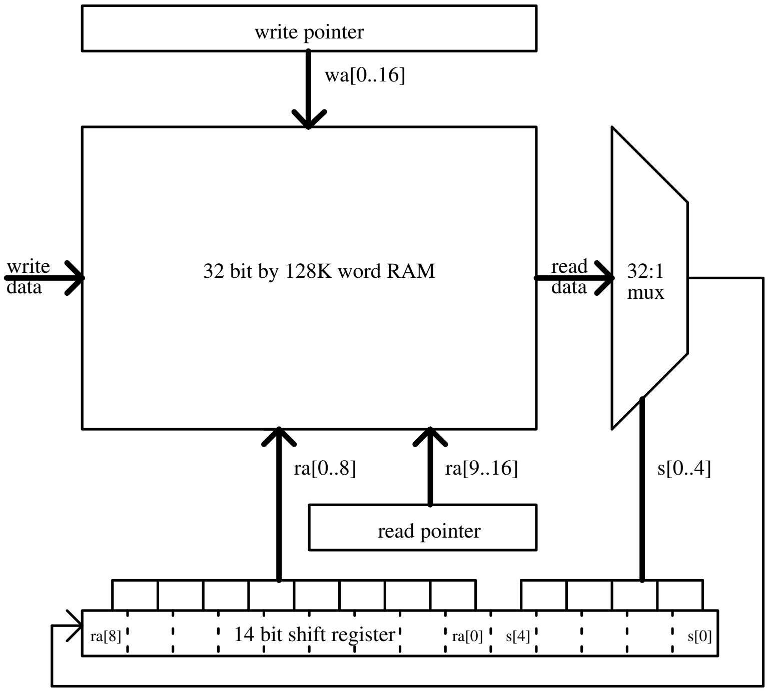

Figure 6. Traceback Decision Memory Architecture

address counter generates the 17–bit write pointer, wa[0..16], starting at column 0 word 0. After 512 writes a complete set of 16,384 decisions has been written to the first column and the decisions for the next codeword are written starting at column 1 word 0. The read address counter generates an 8–bit read pointer which forms the 8 most significant bits of the read address, ie ra[9..16]. This pointer is indexing the column address of the memory. The 9 least significant bits of the read address, ra[0..8], form the row address. This and the 5 select inputs to the 32:1 mux, s[0..4], are generated by the 14–bit shift register connected to the multiplexer output. This arrangement effectively implements a 16,384:1 multiplexer.

The read pointer counter is initialised to 256 and the shift register is initialised to 0. The traceback begins after each new column has been written and traces back through the last 255 codewords by decrementing the read pointer 255 times. This means the read pointer will always begin the traceback on the column that has just been written (ie the most recently received codeword). The shift register shifts right during every read to select the next state (ie the row address and output bit from the 32:1 mux) from the new column. This architecture has a natural speed advantage [7] of 512:1, as it takes 512 write accesses to store a complete column of 16,384 decisions for

1 received codeword, whereas it only takes 1 read to retrieve the required decision from each column. For this reason a 255 bit traceback has been implemented, which is over twice the minimum decision depth required for a constraint length 15 code.

5. Results and Conclusions

The decoder was tested by decoding noisy data generated by software. The test program generates a pseudo random binary sequence (PRBS) which is encoded and corrupted by additive white Gaussian noise with zero mean and variance $\sigma^2 = N_0/2$, where N_0 is the two–sided noise spectral density. The corrupted input data for various ratios of the average bit energy (E_b) to N_0 is fed to the decoder and the decoded output bit stream compared to the original PRBS to determine the bit error rate. Figure 7 shows the measured bit error rate performance of the decoder. The coding gain is about 0.5dB below the curve published in [8] for the Galileo space probe polynomials. Some degradation in coding gain is expected in a hardware implementation which must use quantised inputs and integer arithmetic.

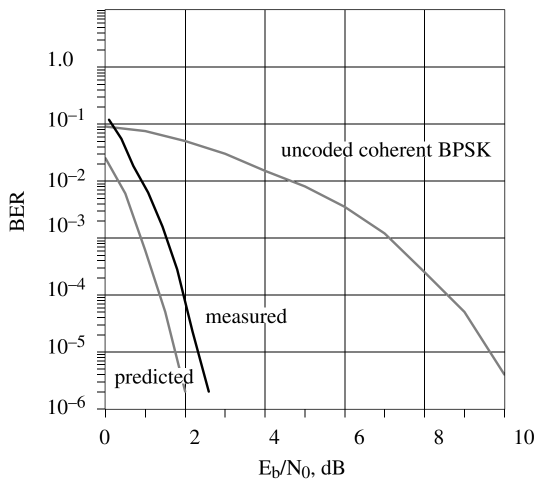

Figure 7. BER performance of the K=15 codec

Further work is being carried out on different metric calculation algorithms to improve the coding gain. The system will also be used to perform a partial code search of all the rate 1/4 constraint length 15 codes to determine if any codes with significantly better error performance than the code used on the Galileo space probe exist. Further refinements to the architecture to utilise more of the DECPeRLe–1 hardware resources are also being studied. By multiplexing the use of ACS processors in both space and time and employing the *state relabelling* technique described in [9], 4 states of 2 input symbols can be calculated at once resulting in twice the throughput.

6. Acknowledgment

The bulk of this work was carried out at PRL in May 1993. The authors gratefully acknowledge the support of Digital Equipment Corporation Australia and Digital Equipment Corporation Paris Research Laboratory.

7. References

[1]. J E Hann, R A Keaney, D J Coggins, D J Jonas, C Potter, D J Skellern, "Forward error correcting Viterbi Codec'", *Proceedings of the 8th Australian Conference on Microelectronics,* Brisbane 1989.

[2]. O M Collins, "The Subtleties and Intricacies of Building a Constraint Length 15 Convolutional Decoder", *IEEE Transactions on Communications,* vol. 40, no. 12.

[3]. P Bertin, D Roncin, J Vuillemin, "Introduction to Programmable Active Memories", *DEC PRL Research Report,* June 1989.

[4]. G C Clark, J B Cain, "Error–Correction Coding for Digital Communications", New York: Plenum, 1981.

[5]. P Bertin, P Boucard, "DECPeRLe–1 Hardware Programmers Manual", *DEC PRL,* Dec 1992.

[6]. H Touati, "PeRLe1DC: A C++ Library for the Simulation and Generation of DECPeRLe–1 Designs", *DEC PRL,* Dec 1992.

[7]. D J Coggins, "A Design Methodology for High Performance VLSI Systems", *PHd Thesis,* Sydney University, 1991

[8]. J H Yuen, M K Simon, W Miller, F Pollara, C R Ryan, D Divsalar, J Morakis, "Modulation and Coding for Satellite and Space Communications", *Proceedings of the IEEE,* vol. 78, no. 7, July 1990.

[9]. D J Coggins, B Vucetic, D J Skellern, "Trellis partitioning scheme based on state relabelling for Viterbi decoding in VLSI", *Proceedings of the International Symposium on Information Theory,* IEEE 1988.