Turbo Code implementation on the C6x

William J. Ebel
Associate Professor
Alexandria Research Institute
Virginia Polytechnic Institute and State University
email: webel@vt.edu

Keywords: Error Correcting Codes, Turbo-Codes, Fixed-Point Numbers, MAP Decoding, Soft Decision

Abstract: In this paper, we describe some important issues and our progress in implementing a Turbo decoder on the TMS320C6201 programmable DSP. Furthermore, we describe some advancements that might make a Turbo decoder implementation on the C6x more efficient. Benchmarks for evaluating the performance of hardware implementations are featured, as well as performance results for efficient implementations on the Texas Instruments TMS320C6201 fixed point DSP.

I. INTRODUCTION

Turbo codes are being proposed for the 3rd generation wireless standard known as 3GPP [2]. In this paper, we describe an implementation of the Turbo decoder algorithm in a C6x along with important implementation issues. The issues include normalization, a stopping criteria, trellis termination, etc.

The parallel-concatenated Turbo encoder takes the form as shown in Figure 1 [2]. The

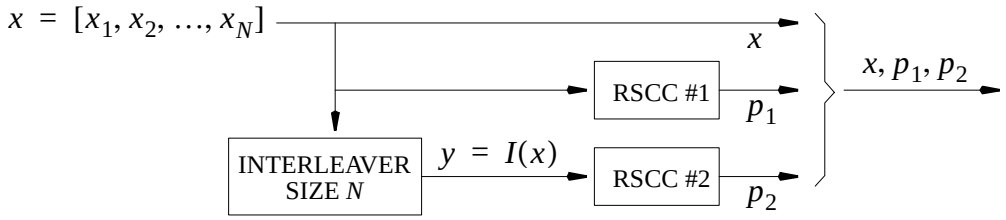

Figure 1. Parallel concatenated Turbo encoder.

information vector x is input into a recursive systematic convolutional code (RSCC) and an interleaved version of it is input into a second RSCC encoder. The Turbo encoder output vectors x, p_1, and p_2 are all binary, i.e. have components drawn from $\{0, 1\}$. We assume that the modulator is binary and implements the mapping $c = 2b - 1$ where $b \in \{0, 1\}$ and $c \in \{1, -1\}$. Furthermore, we assume that the channel noise is AWGN with power σ^2. Then each measured component is given by a Normal distribution with conditional mean ± 1 and

conditional variance σ^2. Let the measured vectors be denoted x', p'_1, and p'_2. These are easily converted into log-likelihood ratios (LLR) by scaling by the factor $2/\sigma^2$. Let $\Lambda(x)$, $\Lambda(p_1)$, and $\Lambda(p_2)$ denote the measured vectors in LLR form.

The standard Turbo decoder algorithm is shown in Figure 2 [2]. The parity LLR vectors are

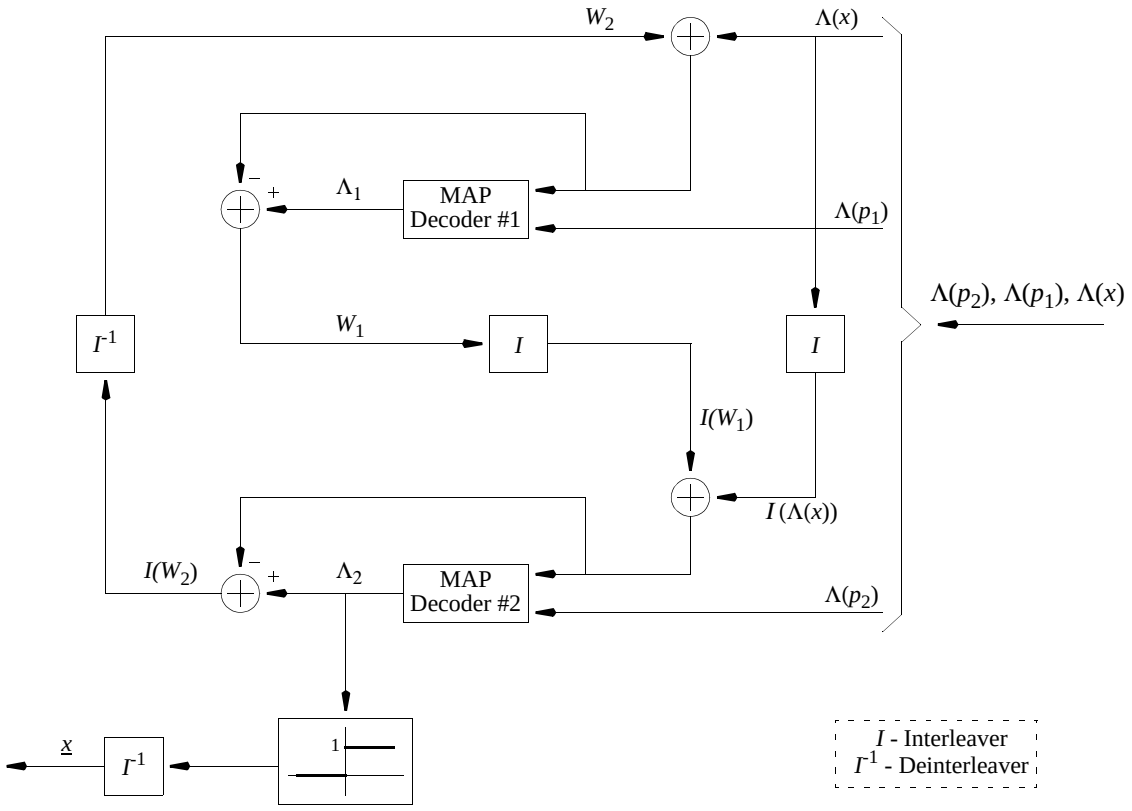

Figure 2. parallel concatenated Turbo decoder.

input into two different MAP decoders and $\Lambda(x)$ is combined with an extrinsic vector prior to each MAP decoder. The result is an algorithm that iterates around a loop which successively refines the estimate of the information vector until convergence is reached.

A simpler form for the Turbo decoder is shown in Figure 3. Here each MAP decoder estimates

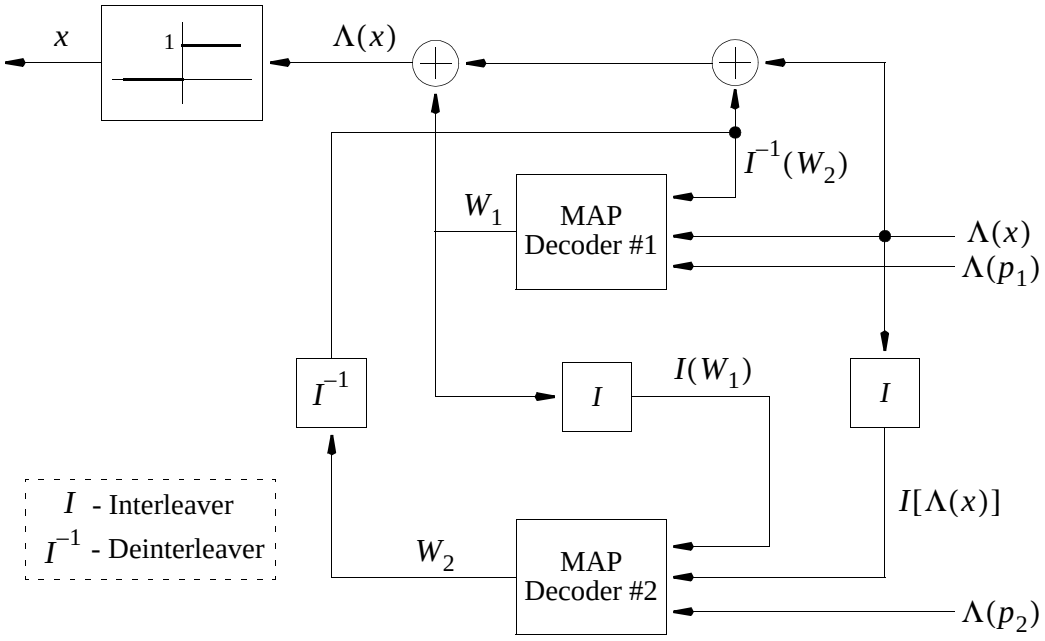

Figure 3. Simplified Turbo decoder.

the extrinsic vectors directly. This direct measure of the extrinsic vectors is achieved with a slight complexity reduction in the MAP decoder and it also eliminates the subtraction at the output of each MAP decoder shown in Figure 2. This version of the decoder is also more inherently stable when using fixed-point numbers.

MAP Decoder:

The MAP decoder requires that two sets of state metrics be computed, one using a forward recursion through the trellis, the $A_i(j)$ metrics, and one using a backward recursion through the trellis, the $B_i(j)$ metrics.

Specifically, the $A_i(j)$ metrics are computed using the forward recursion

$$A_i(k) = \ln\{\exp[A_{i-1}(j_0) + \Gamma_i(j_0, k)] + \exp[A_{i-1}(j_1) + \Gamma_i(j_1, k)]\} \quad (1)$$

where j_0 and j_1 are the states at stage $i-1$ that join to state *k* at trellis stage *i*. This recursion is initialized by

$$A_0(i) = \begin{cases} 0, i = 0 \\ -\infty, \text{else} \end{cases}$$

where an index of $i = 0$ refers to the all zero state. Similarly the $B_i(j)$ metrics can be computed by the backward recursion

$$B_{i-1}(j) = \ln\{\exp[B_i(k_0) + \Gamma_i(j, k_0)] + \exp[B_i(k_1) + \Gamma_i(j, k_1)]\} \quad (2)$$

where k_0 and k_1 are the states at stage i that join to state j at trellis stage $i-1$. The initial conditions are

$$.B_n(j) = \begin{cases} 0, j = 0 \\ -\infty, \text{else} \end{cases}$$

For a rate 1/2 RSCC, the $\Gamma_i(j, k)$ metrics are given by

$$\Gamma_i(j, k) = x_i\Lambda(x_i) + p_i\Lambda(p_i)$$

where $x_i, p_i \in \{0, 1\}$ and where p_i refers to a component of p_1 or p_2. Finally, the extrinsic output is given by

$$W_i = \ln\left\{\sum_{j,\, k\ni x_i = 1} \exp[A_{i-1}(j) + p_i\Lambda(p_i) + B_i(k)]\right\} - \ln\left\{\sum_{j,\, k\ni x_i = 0} \exp[A_{i-1}(j) + p_i\Lambda(p_i) + B_i(k)]\right\} \quad (3)$$

where W_i and p_i refer to the extrinsic and parity associated with the same MAP decoder. We note that there are always the same number of branches associated with each summation. Therefore, any constant offset associated with either the alpha or beta metrics for a give set of trellis states will subtract out.

II. TURBO DECODER ISSUES

There are two issues associated with a practical implementation that we briefly discuss: (1) state metric normalization, and (2) a stopping criteria.

A. Normalization

As the state metrics are successively computed for the forward recursion, a positive bias becomes apparent. This is a problem if a number representation is used that limits the dynamic range, such as fixed-point numbers. Since the calculation for W_i is not affected by a constant offset for the alpha metrics at a given set of state metrics, the alpha metrics can be normalized by subtraction by a constant for that state. A similar observation is made for the beta metrics. For the

i^{th} stage, then, the normalizing constant is

$$C = \text{Max}\{A_i(k) \; ; \; k = 0, 1, \ldots, S-1\}$$

where S is the number of states in the trellis per stage.

B. SNR Stopping Criteria

A second issue deals with the iterative nature of the decoder. The bit-error rate (BER) associated with the output of the Turbo decoder after each complete iteration reaches a point of diminishing returns (convergence). Moreover, the BER at a given iteration varies widely for each received codeword. Most of the received codewords will converge after just 2 or 3 iterations, while a small percentage of codewords require 8 or more iterations to converge. Since the number of decoder iterations directly relates to latency (and complexity if it is defined in terms of total operations), then it is desirable to reduce the *average* number of iterations required for Turbo decoding. We introduce a simple method for terminating the iterative process in the Turbo decoder that involves measuring the signal-to-noise ratio (SNR) of the extrinsic information at the output of each MAP decoder. When the SNR exceeds some predetermined threshold, the iterations are stopped.

It is well known [2] that the extrinsic components become conditionally Gaussian as the Turbo decoder iterates. This suggests that the BER associated with the extrinsic vector can be determined using the standard Q-function. In fact, the argument of the Q-function can be related to an extrinsic SNR. The standard relationship between SNR and the BER of a BPSK modulation scheme is

$$P_W(\varepsilon) = Q(\sqrt{\text{SNR}_W})$$

where $P_W(\varepsilon)$ is the error probability associated with the extrinsic vector W and SNR_W is the corresponding extrinsic SNR.

The SNR_W can be empirically determined from the components by

$$\text{SNR}_W = \frac{m_W}{\sigma_W^2}$$

where

$$m_W = \frac{1}{N} \sum_{i=0}^{N-1} |W_i|$$

and

$$\sigma_W^2 = \frac{1}{N-1} \sum_{i=0}^{N-1} W_i^2 - m_W^2.$$

Although the extrinsic components are not very Gaussian prior to convergence, this is exactly the situation where SNR_W is small and, therefore, is not of concern. From a complexity standpoint, the mean and variance calculation given above can be done as the extrinsic components are being calculated and do not constitute a significant amount of additional complexity.

To illustrate the virtue of this approach, the histogram of the extrinsic components were plotted for a Turbo code using 4-state RSCC and a blocklength of 120. The result, given in Figure 4, shows that the extrinsic vector diverges rapidly at some particular iteration. This divergence also corresponds to the iteration where no errors in the decoded vector occur. The following table illustrates the typical trend for a specific received codeword which resulted in no decoded errors

after iterating 5 times.

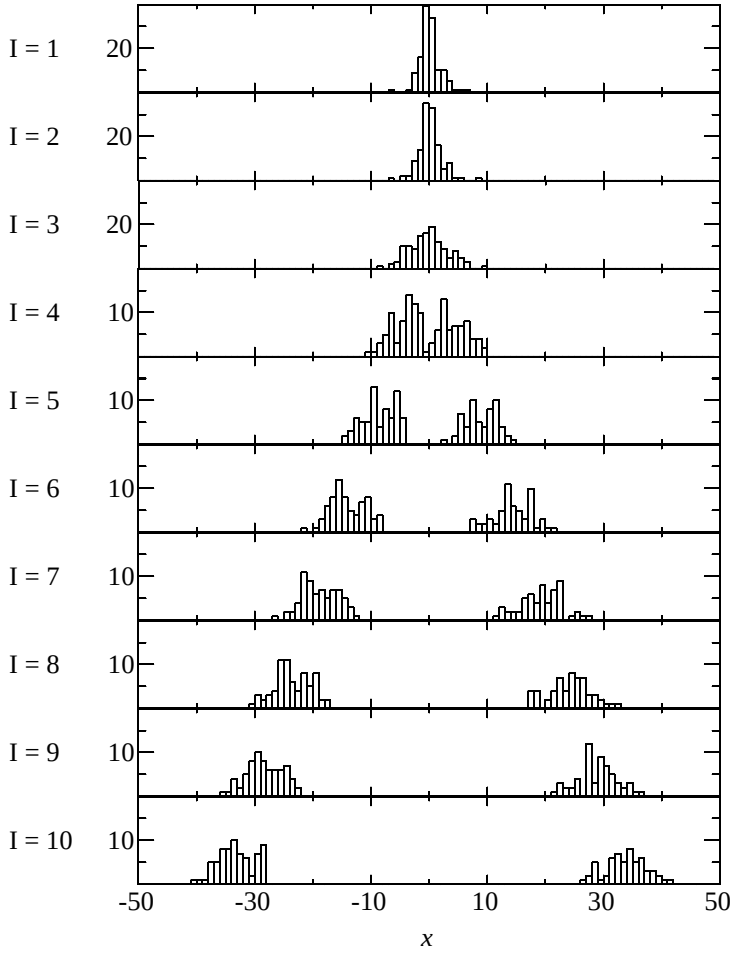

Figure 4. Historgram of the an extrinsic vector

The following table gives actual performance results for the same Turbo code with 4-state constituent RSCC and for 3 different thresholds. The table clearly shows that there is a performance trade-off of # iterations vs. the decoded BER that is a function of the SNR threshold.

Table I. SNR stopping criteria results.

	Threshold = 25		**Threshold = 10**		**Threshold = 5**	
E_b/N_o	**# Decoder Errors**	**Ave. Iterations per Block**	**# Decoded Errors**	**Ave. Iterations per Block**	**# Decoded Errors**	**Ave. Iterations per Block**
0	7865	4.902	7861	4.697	7873	4.538
1	1336	4.372	1336	3.609	1361	3.161
1.5	396	3.858	398	2.942	442	2.460
2	94	3.348	94	2.444	141	2.002
2.5	30	2.865	30	2.087	104	1.595
3	0	2.506	4	1.806	74	1.288

The threshold of 25 gave rise to performance that was nearly identical to the performance with a full number of iterations but resulted in about half the number of average iterations for $E_b/N_0 = 3$ dB. We also observe that for low SNR, the decoder iterates the full number of times regardless of the threshold. Clearly, the threshold should be set according to the designed SNR.

III. TMS320C6201 IMPLEMENTATION

This section describes our C6x implementation as it currently stands. This is an ongoing effort and our future interests will be focused on implementing the stopping criteria and in streamlining the implementation to maximize the data throughput rate.

A. Development and Test Environment

The hardware and software environment used to implement the Turbo Decoder consisted of an evaluation module provided by Texas Instruments and a software development environment provided by GO DSP. All hardware used in the development was on the TMS320C62001 EVM (evaluation module). The primary hardware resources that were used in the decoder implementation are:

- TMS32062001 fixed-point digital signal processor with a maximum clock frequency of 200MHz.
- 64k bytes of internal program memory running at system clock speed of 200MHz.
- 64k bytes of internal data memory running at system clock speed.
- 8M bytes of SDRAM located on the EVM running at 100MHz maximum clock frequency (systemclockspeed/2). The SDRAM was used to store the sample data and post processing of the decoded data to gather error statistics.

The limited memory resources of the hardware and the desire for high throughput demand the judicious use of memory and computational resources. Fortunately, careful storage and memory re-use allowed for maximum throughput for block lengths up to 2000 information bits using only the DSP's internal data memory for storage of all computation. For applications requiring more memory, the external SDRAM could be used to allow block sizes as long as 512,000 information bits but result in a throughput reduction.

B. Memory Organization

All memory resources are accessed via the TMS320's on-board DMA controller. The TMS320C6x compiler allows flexible mapping of the memory resources. A memory model is first selected in order to divide the memory up into regions that characterize the size and speed of the memory. The memory model used for the Turbo Decoder implementation is shown in Table II. The fastest memory regions are the internal program memory (IPM), and internal data memory (IDM). The IPM stores the actual DSP executable code. The IDM stores the stack, local variables,

and any variables that require high-performance memory. The remaining memory regions are the slower external memory. Table III shows how the various program variables are assigned to memory regions. Most of the region assignments are fairly general. However, the decoder working memory, sample data, and error statistics section assignments were made due to the program's requirement for performance during certain variable accesses. The Turbo decoder's working memory is assigned to the IDM for high performance, while all post-processing memory is assigned to the slower memory regions.

Table II. Memory Model Used for the Turbo Decoder Implementation

Memory Type	Origin (Hex)	Length (Hex Bytes)	Length (Hex Bytes)	Type
INTPROG	0x00000000	0x010000	64k	IPM
INTDATA	0x80000000	0x010000	64k	IDM
EXTMEM0	0x00400000	0x040000	256k	SBSRAM
EXTMEM1	0x02000000	0x400000	4M	SDRAM
EXTMEM2	0x03000000	0x400000	4M	SDRAM

Table III. Decoder Memory Section Assignments

Region	Variable	Description	Type	Size
INTDATA	IMAP[BS]	Interleaver map	ushort	2 BS
INTDATA	IMAPU[BS]	Deinterleaver map	ushort	2 BS
INTDATA	LxAD[BS]	received x sample	short	2 BS
INTDATA	Lp1AD[BS]	received parity 1 sample	short	2 BS
INTDATA	Lp2AD[BS]	received parity 2 sample	short	2 BS
INTDATA	Lext1[BS]	MAP dec. 1 extrinsic data	short	2 BS
INTDATA	Lext2[BS]	MAP dec. 2 extrinsic data	short	2 BS
INTDATA	A[BS][NS]	Alpha calculations	short	8 BS
INTDATA	B[BS][NS]	Beta calculations	short	8 BS
INTDATA	numErrors	Post-processing data	unsigned	4
			Total:	30 BS + 4
EXTMEM1	xSamples[BS NBL]	all received x samples	short	2 BS NBL
EXTMEM1	p1Samples[BS NBL]	all received parity 1 samples	short	2 BS NBL
EXTMEM2	p2Samples[BS NBL]	all received parity 2 samples	short	2 BS NBL
EXTMEM2	xOut[BS NBL / 16]	binary x estimate from decoder	short	BS NBL / 8
EXTMEM2	goldData[BS NBL / 16]	source x data from encoder	short	BS NBL / 8
			Total:	(11/4) NBL BS

BS = interleaver size
NS = Number of encoder states
NB = Number of branches from a state
NBL = Number of blocks

C. Computation

The computational complexity of the Turbo Decoder is dominated by the MAP decoder implementation. Specifically, the alpha, beta, and gamma metrics must be computed for every stage in the block. Therefore, the performance of these three computations limit the speed of the decoder.

1. Gamma Computation

The gamma metric stores the result of the computation. For the $(1 + D^2)/(1 + D + D^2)$ encoder that was chosen for this implementation, there are 4 possible received symbol sequences. These sequences are shown in Table IV. Because the TMS320C6201 has a 4 cycle memory fetch,

Table IV. Branch metrics.

Sequence		
X	***P***	**Branch Metric**
0	0	0
0	1	$\Lambda(p_i)$
1	0	$\Lambda(x_i)$
1	1	$\Lambda(x_i) + \Lambda(p_i)$

and to save valuable internal data memory resources, it was decided that incorporating the gamma calculations into the alpha and beta computation routines was the most efficient implementation. Table IV shows that the gamma calculation actually only requires one computation. Redundantly calculating the one computation turns out to be more efficient than storing the data to memory.

2. Alpha/Beta Computation

The alpha and beta computation routines implement the computation given in (1) and (2), respectively. The summation that is given in (1) represents the addition of the branch metrics for multiple branches entering a given state. Taking the natural log of this sum is computationally expensive. The simple approximation is made to circumvent the natural log computation. If we assume that in most cases $A \gg B$, then we can approximate the exponential adder as just a magnitude comparison:

$$\ln(e^A + e^B) \approx \mathrm{MAX}(A, B) \tag{4}$$

This approximation yields good performance, with a slight coding loss of about 1/2 dB. Implementing the LOG MAP decoder, which does not use the approximation shown above would require a look-up table. The following shows the forward recursive calculation of one alpha metrics at a given stage, taking advantage of the TMS3206X's **_sadd** intrinsic function which computes a saturated add. The beta calculations are similar, except the recursion is backwards,

which changes the gamma metrics' assignment to the state metrics.

```
/** convert operands to (int) and saturated add - more efficient*/
/** this is our one gamma calculation */
g = _sadd(Lest[iStage-1]<<16, Lp[iStage-1]<<16);
/** i1 is one branch, i2 is the second branch entering a state */
i1 = *(Aptr+jStageAlpha-4+0)<<16;
i2 = _sadd(*(Aptr+jStageAlpha-4+2)<<16, g); /* g21 */
/** our approximated ln[exp(i1) + exp(i)] */
*(Aptr+j+0) = (short)(_VMAX(i1, i2)>>16); /*~ ln[exp(i1) + exp(i)] */
```

3. Information LLR Calculation

Calculation of the log-likelihood ratios involves the implementation of (3). The approximation for the log-add given by (4) is again utilized here. The following code illustrates the calculation of the $x = 1$ LLR from the alpha's and beta's (Aptr and Bptr, respectively).

```
/** Our one gamma calculation */
g = Lp[i]<<16; /*g21, g01*/
/* calc 1's LL */
LL1 = _VMAX(_sadd(_sadd(*(Aptr+j+0)<<16,g),*(Bptr+j+1)<<16),
_sadd(*(Aptr+j+1)<<16,*(Bptr+j+2)<<16));
LL1 = _VMAX(LL1, _sadd(_sadd(*(Aptr+j+2)<<16,g),*(Bptr+j+0)<<16));
LL1 = _VMAX(LL1, _sadd(*(Aptr+j+3)<<16,*(Bptr+j+3)<<16));
```

IV. PRELIMINARY PERFORMANCE RESULTS

The performance of the decoder was measured in bit-rate and coding gain. The coding gain of the decoder matches the statistics discussed in [3] and is not discussed here. In this section we give the test results of the bit-rate for the implementation.

A. Bit-Rate

Several strategies were employed to increase the overall bit-rate of the decoder. A few of these strategies were particularly successful in increasing the overall throughput. Table 4 gives bit-rate performance for several strategies.

Table V. Bit-rate performance results for different implementations.

Method	clocks/MAP decode	bps/iteration
Dedicated gamma calculation	136,700	88,000
Integrated gamma calculation	61,268	196,000
Integrated gamma/flat arrays	42,000	286,000

1. Integrated gamma calculation

A method was presented that involved the calculation of the branch probabilities, also referred

to as the gamma calculations. In the dedicated gamma calculation method, functions independently calculate all of the gamma calculations for the entire trellis, and store them to memory. As shown in Table IV, the gamma calculation actually only requires one computation, the rest of the operation is a simple reassignment of variables. Due to the high cost of memory accesses a separate gamma calculation is very inefficient. Alternatively, the gamma calculation can be pulled into each of the alpha and beta calculation functions. This saves the processor numerous memory accesses per iteration, at the cost of only two redundant computation (the branch computation in which the information and parity bit are both one must be done in the alpha calculation, the beta calculation, and information LLR calculation independently). As shown in Table V, the integration of these calculations leads to a considerable performance improvement of over 100.

2. Integrated gamma/flat arrays

The variables that store the alpha, beta, and final information estimate are each multi-dimensional arrays. It was discovered that the compiler is not very efficient in computing the pointer addresses for the array accesses. Since the array accesses are sequential in nature, the compiler can be helped along if the programmer flattens the arrays into one-dimension and manually increments the indexes rather than relying on the compiler to make efficient computations. this enhancement further improves the bit-rate by about 33%.

V. CONCLUSION

The purpose of this paper is to present the results of investigations in the implementation of Turbo-Decoding algorithms on hardware architectures. We show that on the TMS320C6201 fixed-point DSP, a common DSP architecture, that decoders for codes of block sizes on the order of 2000 information bits can be implemented just using the DSP alone. Much larger block sizes can be implemented using peripheral memory. In fact, decoders with block sizes on the order of 512,000 information bits can be implemented on the EVM architecture. For decoders implemented on the DSP's internal memory space, bit-rate performance as high as 286,000 bits/ second/iteration can be achieved. Though the C-code for this specific implementation takes advantage of the TMS320C6x's specific architecture, implementations that are written in processor-specific assembly code should be able to achieve bit-rates that are significantly higher.

VI. BIBLIOGRAPHY

[1] see http://www.3gpp.org

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes”, in *Proceedings of ICC ‘93*, Geneva Switzerland, May 1993, pp. 1064-1070.

[3] D.E. Cress, and W.J. Ebel, “Turbo Code Implementation Issues for Low Latency, Low Power

Applications", *1998 Symposium on Wireless Personal Communications*, MPRG, Virginia Tech, June 10-12, 1998.