Hardware-Accelerated NIOS-II Implementation of a Turbo Decoder

Andreas Corneliussen, Erik B. Poulsen, Pradeep Silpakar, Troels T. Østeraa
Department of Electronic Systems
Aalborg University
Aalborg, Denmark

Yannick Le Moullec
Department of Electronic Systems
Center for Software Defined Radio
Aalborg University
Aalborg, Denmark
ylm@es.aau.dk

***Abstract*— This paper presents a hardware accelerated NIOS-II implementation of a turbo decoder. Firstly, a MatLAB prototype consisting of a) an encoder made of a parallel concatenation of two RSC encoders and b) a decoder based on two identical SOVA decoders is constructed. Simulations of the prototype show that the BER decreases for every iteration in the decoder (down to 10^{-4}), except for low SNR cases (below -5 dB). Secondly, two FPGA implementations of the decoder are described and compared. The first one consists of software executing on a Nios II/f soft-core processor, while the second one adds hardware acceleration. Computationally demanding parts of the decoder are identified, rescheduled for parallel execution and moved from the software implementation to the hardware accelerator. The decoding process in the hardware accelerated implementation results in approximately the same BER as for the software implementation, but the execution time is decreased by between 34 % and 25 %, when the number of decoding iterations are increased from 1 to 20, respectively. The accelerated implementation increases the number of required resources from 10 to 16%, as compared to the software one.**

Keywords-turbo decoder; FPGA, soft-core processor; NIOS-II; hardware accelerator;parallelism, partitionning

I. INTRODUCTION

The concept of Turbo Coding (TC) is the idea of using two or more codes on the same symbol sequence, and then use the knowledge obtained by decoding one code to improve the decoding of the second code, and vice-versa, in an iterative manner. This means that, given two very different codes wherein all the information from the source data is reserved, TC is potentially able to iteratively decode until no more errors exist in the decoded symbol sequence. The error correcting performance of TC is therefore dependent on the difference between the two codes. A frequently used method to assure high difference between the two codes is to interleave the input sequence to one of the encoders so that it encodes the same bits in a different order. A typical TC scheme therefore consists of four elements; encoder, decoder, interleaver, and de-interleaver.

The uses for TC are many because it approaches the Shannon limit [1]. It is, for instance, used in applications like magnetic/optical data storage systems, ADSL modems and satellite communication [2]. However, because of the iterative behavior of the decoding process, the error correcting performance is bounded by the computational power and precision of the platform on which it is implemented [3]. It is therefore challenging to implement TC on embedded platforms and to use the concept to its full potential because such platforms are often limited in computational power and precision.

The objective of this work is to investigate whether it is possible to improve the embedded software implementation of TC by means of hardware acceleration. This is achieved by 1) synthesizing a soft-core processor on an FPGA, 2) implementing the software of TC for this processor, and 3) designing and implementing a hardware accelerator to reduce the execution time of the software solution.

The remainder of this paper is organized as follows: Section II presents the MatLAB Prototype of the encoding/decoding chain. Section III details both the software and the hardware accelerator implementations. In section IV the results are presented and section V concludes the paper.

II. MATLAB PROTOTYPE

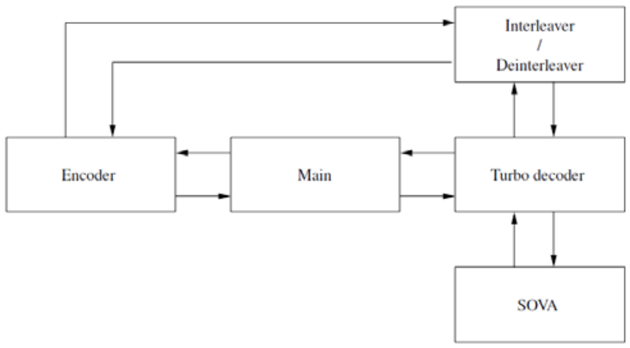

Figure 1. The structure of the MatLAB prototype of a Turbo Coding scheme

The turbo encoder/decoder chain is shown in Figure 1. The encoder is implemented using two Recursive Systematic Convolutional (RSC) [4] encoders with the

This is the authors' version of a paper published in the Proceedings of the International Conference on Computer and Electrical Engineering, 2009.

feedback polynomial g1(D) = $D^2 + D + 1$ and the output polynomials g2(D) = $D^2 + 1$ and g3(D) = D + 1. The interleaver design chosen for this prototype is of the quadratic type [5] and the input block length is chosen to be 2048 bits. The Main function, illustrated in the middle of Figure 1, generates a random bit string to use as source data. This string is forwarded to the encoder to achieve an encoded bit string which requires the use of the interleaver. However, first all bits are converted from a 0/1 representation to -1/1 to be compatible with the decoder, and afterwards Additive White Gaussian Noise (AWGN) is added to emulate the effect of a noisy channel. After the noise is added, the bit string is passed to the decoder which calls the Soft Output Viterbi Algorithm (SOVA) [6] decoder, the interleaver and the deinterleaver functions in the needed order.

A. *Channel Emulation*

The prototype implementation includes an emulation of a channel to be able to test the performance of the system when used with an AWGN channel. Therefore, white Gaussian noise was generated and the power was normalized with respect to the encoded sequence.

Afterwards the noise was multiplied by , where

Signal-to-Noise Ratio (SNR) is the signal-to-noise ratio of the channel emulation in dBs. The noise and the encoded sequence are then added to emulate an AWGN channel.

B. *Test Methodology*

The test consists of encoding long strings of data and then adding noise at different SNRs. The noisy data is then passed to the decoder which performs several iterations before outputting the data. After the decoding is complete, the decoded data is compared with the original strings, and the BER is calculated. It is chosen to test the prototype at SNRs ranging from -8 to 2 dB in steps of 1 dB to show the asymptotical behavior at lower SNRs while still showing the decaying exponential shape for higher SNRs. The test was performed for multiple numbers of iterations ranging from 1 to 8 iterations, since the benefit of performing more iterations is negligible. It is furthermore chosen to repeat the test a 1000 times for each SNR and calculate the mean of the resulting data set to increase the reliability of the results.

C. *MatLAB Simulation Results*

The result of the compliance test is shown in Figure 2. It can be seen from the results that more iterations reduce the BER as expected. It is furthermore visible that the decoder does not improve much on the quality of the received data if the SNR is below -5 dB.

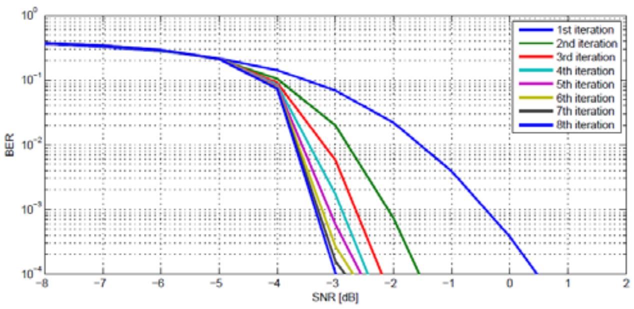

Figure 2. MatLAB simulation results. Each curve shows the result of a certain number of iterations as stated in the legend.

III. FPGA IMPLEMENTATIONS

This section details the software and hardware-accelerated FPGA implementations of the decoder. The selected platform is the Altera DE2 board which includes a Cyclone II EP2C35F672C6 FPGA and the soft-core processor is the NIOS-II.

A. *Software Implementation*

The soft-core processor is designed with arithmetic units only capable of integer calculations. This means that one must either use a fixed point representation, or design a floating point implementation using the arithmetic units available. Such an implementation possibly provides a better numerical stability to the TC implementation and certainly eases the design of it. A floating point implementation, however, increases the number of calculations required for rather simple operations, and therefore increases the execution time of the algorithm significantly. It is therefore chosen to use a fixed point implementation, where all numbers are represented as 32 bit integers with the point placed at 2^{16}. This means that all numbers are multiplied by $2^{16} = 65536$ and rounded to the closest integer lower than the resulting number. The Nios processor, however, is not capable of multiplying or dividing two 32 bit numbers with full precision and range. A consequence is that one has to choose between range or precision in all multiplication operations. Either one must divide each number by 2^8 (or one of them by 2^{16}) and then multiply the numbers, or multiply the numbers and then divide the result by 2^{16}. The first method throws away 8 bits of precision in each number before multiplying the numbers, whereas the second method can cause up to 32 bits of overflow since $(2^{31}-1)^2 = 2^{62}-2^{32}+1$, which does not fit in a 32 bit register. The software for the embedded system has therefore been designed to use the best of the two methods, i.e. divides one operand by 2^8 and also divides the result by 2^8.

The overall system is illustrated in Figure 3.

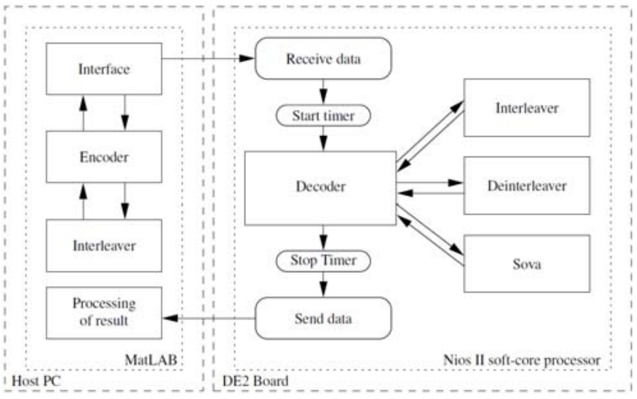

Figure 3. The software implementation of the turbo decoder (right) and its connection to the MatLAB implementation of the turbo encoder.

The software embedded in the Nios II soft-core processor is composed of 5 functions: *main* to receive the data from the PC, place it in the right data structures and call the decode function. When the decode function is finished, *main* transmits the decoded data to the PC. It also measures the execution time of the other functions, and transfers this information to the PC. *decode* is in charge of the iterative part of the decoding process, it calls the functions *interleave*, *deinterleave* and *sova*. *interleave* and *deinterleave* are in charge of interleaving and deinterleaving either the La, sys or LLR sequence. It has been chosen to implement a quadratic interleaver and implement the permutation using a lookup table rather than the matrix-vector multiplication. *sova* performs the actual decoding.

The soft-core processor is configured as follows. Core type: Nios II/f, Hardware multiply: Embedded multipliers, Hardware divide: Yes. The peripherals are as follows. SRAM: 512 KB (actual usage is only 400 Kb),RS-232 UART, JTAG UART, and TIMER. The system is synthesized and programmed from the Quartus II suite, the C program is compiled and sent to the Nios II/f using the Altera Nios II IDE program. After the completion of the synthesis in Quartus II, the summary report states that the total amount of LEs used is 3250 out of 33,216, i.e. roughly 10 % of the available LEs. The execution time measurements of the soft-core implementation are presented in Section IV.

B. Hardware Accelerated Implementation

The next step is to examine the software implementation to identify the most computationally intensive parts of the algorithm by means of profiling. In Nios II IDE, the pprofiler is called niosii-elf-gprof. This profiler samples the program counter at certain intervals which quickly produces results but it might not give the most precise results. The profiler output is generated by the Turbo decoder functions on the Altera DE2 board using the Nios II/f soft-core processor operating at 50 MHz. A summary of the flat profile of the functions is given in Table I.

TABLE I. FLAT PROFILE SUMMARY

Function name	% Time	Self-Time	Calls	Self s/call	Total s/call
Decoder	1	0.01	1	0.01	1.15
Sova	80	0.92	2	0.46	0.56
Interleave	1	0.01	2	0.00	0.00
Deinterleave	1	0.01	2	0.01	0.01
Jabob_log	17	0.20	98368	0.00	0.00

Since the sova and the Jacob_log function are using 97 % of the execution time spent on decoding, it is decided to analyze them to determine whether they can be accelerated. To initiate the exploration process, the *Sova* and *Jacob_log* functions are analyzed using combined Precedence Graphs (PG) and Data Flow Graphs (DFG). The operations in the computation of the *best forward and backwards path cost* are illustrated in the combined DFG and PG in Figure 4. The combined DFG and PG for the *output computation* of the sova function is seen in Figure 5. The *Jacob_log* function, which is called by sova function, is shown in Figure 6. The inherent parallelism can be seen in the patterns in the combined DFGs/PGs in figures 4-6. Some observations can be made regarding the parallelism: the computation of branch0 and branch1 is not possible before the computations of par1cost, par2cost and logprob have completed. The algorithm in Figure 6 requires knowledge of mu best and mu best back to calculate a temporary cost, right after the calculation of branch0 and branch1. The orders of calculation of a) branch0 and branch1, b) syscost, par1cost and par2cost, and of c) first term and second term are irrelevant. The rescheduling of the algorithm for the computation of forward and backwards path costs in the *sova* function, can be performed as shown in Figure 7.

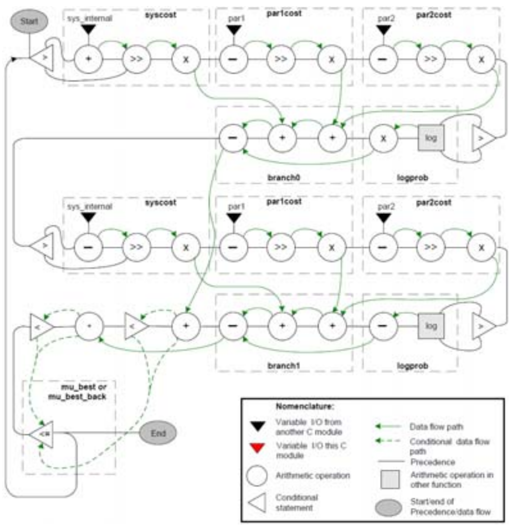

Figure 4. The PG and DFG of the backwards and forward evaluation of the path cost.

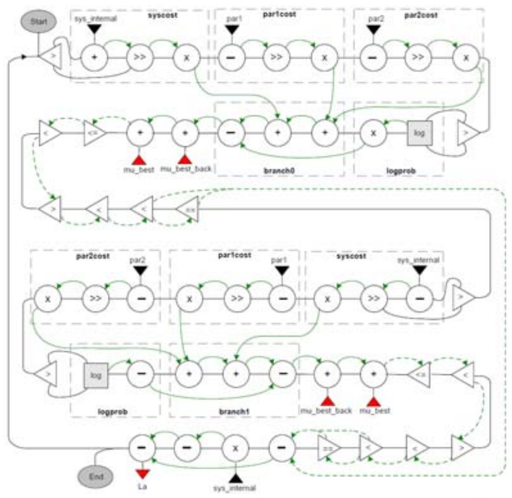
Figure 5. The PG and DFG of the output computation from the branch metrics.

Figure 8 shows the second part, where the branch metrics are re-computed and used with mu best and mu best back from Figure 7 to calculate the extrinsic value, La, and the softoutput, LLR.

The last rescheduled function, Jacob_log, is depicted in Figure 9.

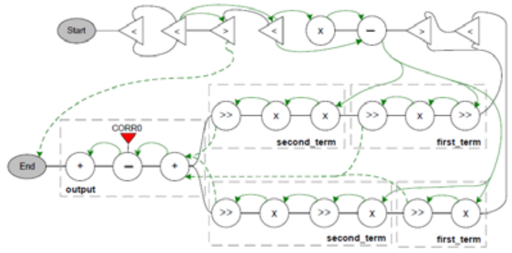
Figure 6. The PG and DFG of the jacob log function which is called from the *sova* function in order to calculate the branch costs, branch0 and branch1.

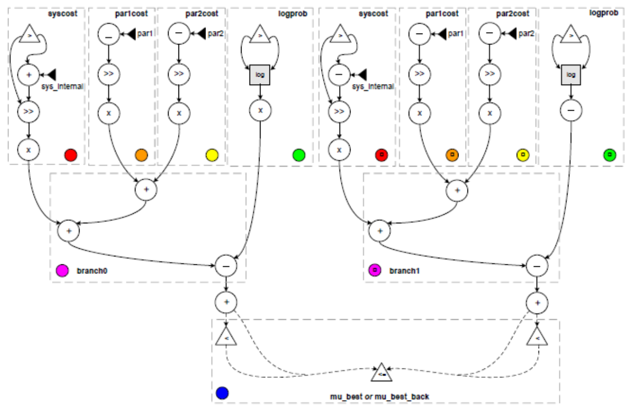
Figure 7. Structure of re-scheduling the sequential code to extract the inherent parallelism.

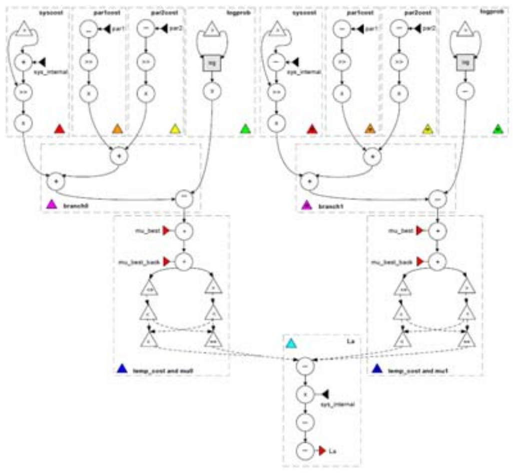
Figure 8. Rescheduling the second part of the SOVA algorithm, where the extrinsic values, La, and the soft-outputs, LLR, are calculated.

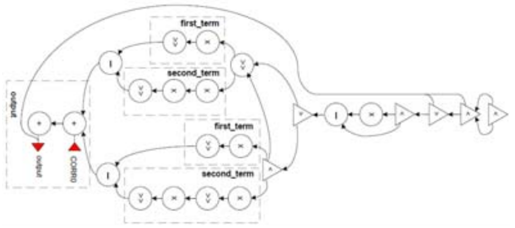
Figure 9. Structure of re-scheduling the sequential code for the jacob log function to extract the inherent parallelism.

It is selected to implement a hardware accelerator with the structure shown in Figure 10. It can compute the branch costs for all three loops, and it can therefore function as a generic block. This is one solution trading off area and execution time, but it can easily be scaled as several accelerators could be implemented and offer branch cost computation. As stated in III.B the soft-core implementation took up 3,250 LEs in the FPGA, which corresponds to roughly 10 % of the total number of LEs. When adding the hardware accelerator, the area consumption increases to 5,909 LEs, which is 16 % of the total area consumption.

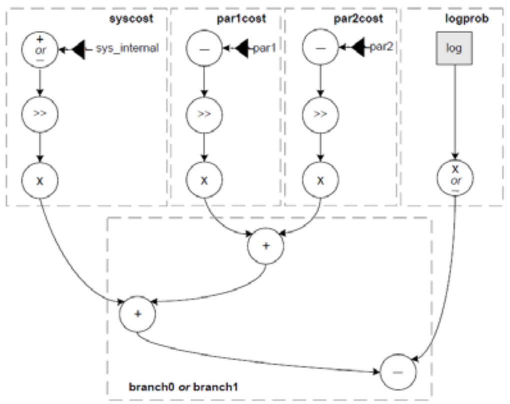

Figure 10. The selected parts of the algorithms for the hardware implementation.

This is the authors' version of a paper published in the Proceedings of the International Conference on Computer and Electrical Engineering, 2009.

IV. Experimental Results

An experimental test is specified in order to provide the required data to be able to compare the two different implementations of Turbo Coding (TC), with regards to execution time, while also verifying that the turbo coding principle is preserved. This test is specified as follows: 1. Generate a random bit sequence of length 2048 using MatLAB, 2. Add white Gaussian noise and ensure an SNR level of 2 dB, 3. Send the bit sequence to both implementations and make them decode with the number of iterations ranging from 1 to 20. 4. For both implementations the time used for decoding is measured. The time is measured in exactly the same way in the two implementations to remove bias. The time, along with amount of iterations, should be noted. The tests are performed with and without the hardware acceleration. The results of the first test, testing the execution time of the two implementations, are shown in Figure 11. The results of the BER test are shown in Figure 12.

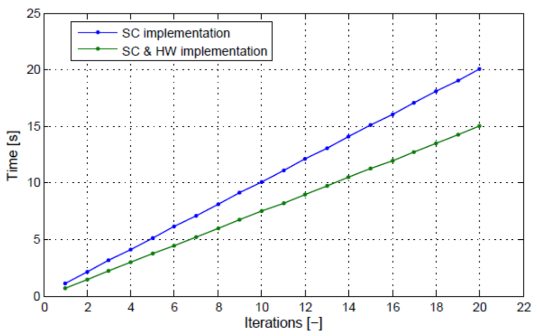

Figure 11. Execution time for the soft-core implementations, with and without hardware acceleration.

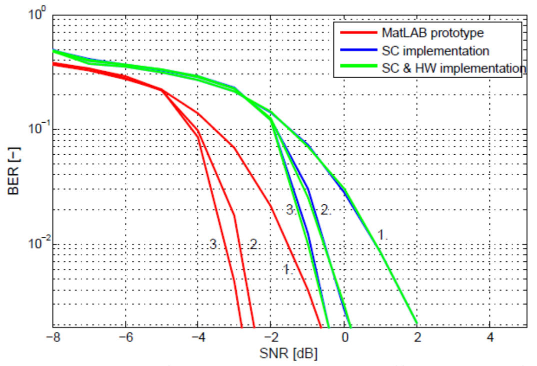

Figure 12. Results for the BER achieved at different levels of SNRs for the prototype in MatLAB and the two implementations. The numbers next to the curves indicate the number of iterations for the specific curve.

The results for the first test shows that the combined soft-core and hardware implementation is faster than the pure soft-core implementation. Specifically, at for example 10 iterations, the two implementations differ by 2.614 s in favor of the hardware accelerated implementation. The results also show that the dependency between execution time and number of decoding iterations has an approximately linear relationship. In detail, the hardware accelerated implementation is faster than the soft-core implementation by between 34.44% and 25.13% for one and 20 iterations, respectively. The cost of this acceleration is moderate since the required resources of the FPGA are increased from 10% to 16% (cf.III.B).

The results of the second test show that the curves of the two implementations have a shape similar to the curve for the prototype decoder. The BER decreases with increasing SNR and the number of decoding iterations. This suggests that the Turbo coding functionality is still preserved. At any SNR, the BERs of the implemented versions are roughly equal. However, compared to the prototype, the two implementations require SNRs 2 dB higher to achieve the same BER as the prototype.

V. Conclusion

A hardware accelerated NIOS-II implementation of a turbo decoder has been presented. In the first phase, simulations of the MatLAB prototype have shown that the BER decreases for every iteration in the decoder (down to 10^{-4}), except for low SNR cases (below -5 dB). In the second phase, two FPGA implementations of the decoder have been described and compared. The first one consists of software executing on a Nios II/f soft-core processor, while the second one adds hardware acceleration. Computationally demanding parts of the decoder have been identified, rescheduled for parallel execution and moved to the hardware accelerator. The two implementations require SNRs 2 dB higher to achieve the same BER as the MatLAB prototype. The decoding process in the hardware accelerated implementation results in approximately the same BER as for the software implementation, but the execution time is decreased by between 34 % and 25 % when the number of decoding iterations is increased from 1 to 20, respectively. The accelerated implementation increases the number of required resources from 10 to 16%, as compared to the software one.

References

[1] Shannon, C. E. (1948). A mathematical theory of communication. Bell System Technical Journal, 27:50–64.

[2] Sripimanwat, K., editor (2005). Turbo Code Applications - a journey from a paper to realization. Springer.

[3] Jin, Y., Zhang, F., and ling Wu, W. (2006). Reduced-complexity turbo equalization for turbo coded mimo/ofdm systems. The Journal of China Universities of Posts and Telecommunications, 13(1):93 – 98.

[4] Sklar, B. (2002). Fundamentals of turbo codes.

[5] Takeshita, O. and Costello, D.J., J. (1998). New classes of algebraic interleavers for turbo-codes. Information Theory, 1998. Proceedings. 1998 IEEE International Symposium on, pages 419–.

[6] M. R. Soleymani, Y. G. U. V. (2002). Turbo coding for satellite and wireless communications. Kluwer Academic Publishers.