Software Polar Decoder on an Embedded Processor

Bertrand Le Gal, Camille Leroux and Christophe Jego
IMS Laboratory, Institut Polytechnique de Bordeaux, Talence, France.

***Abstract*—This paper presents the software implementation of a Polar Codes decoder on an embedded processor. An efficient use of computation and memory resource is made in order to devise a fast polar decoder on an embedded ARM processor. Memory footprint reduction and algorithmic simplifications are applied in order to increase the throughput of the decoder. The NEON instruction set of ARM processors is used to exploit the parallelism of the algorithm. The resulting decoder description is implemented on a Cortex A9 ARM processor. The throughput of the resulting decoder is reported and discussed for several parameters : the code rate, the code length and the multithreading mode. To the best of our knowledge, this is the first reported implementation of a polar decoder on an embedded processor core. The proposed software decoder reaches >100Mbps for a codelength of 16K. Moreover, it compares favorably with state of the art LDPC decoders implemented on embedded processors.**

I. Introduction

Polar codes [1] keep on gaining attention among both information theorists and circuit and system designers. The practical interest of these codes depends on the possibility to implement efficient decoders. To this end, several works focused on algorithmic improvements [2]–[5] and efficient dedicated architectures [6]–[11]. Recently, another implementation approach was investigated in [12] [13] where software polar decoders were proposed on x86 processor targets. In [12], the Single Instruction Multiple Data (SIMD) programming model was used in order to execute some of the computations of a modified-SC decoding algorithm in parallel. In [13], several frames are processed in parallel which enables to reach several Gbps on an x86 single core. However, x86 targets are power/energy hungry processors and may not be suitable for embedded channel decoding such as software defined radio.
Today's embedded processors also include parallel processing facilities such as SIMD instruction sets, multi-core architecture and high performance pipeline. One of the challenges when using these high performance embedded platforms is the definition of the programming model and the ability to efficiently parallelize the executed algorithm. In this paper, we take advantage of the parallelism available in ARM processors (SIMD, multi-core) in order to efficiently execute the Successive Cancellation (SC) decoding algorithm of polar codes. The proposed approach was originally mapped on x86 targets [13]. In this work, we investigate the performance of SC decoder software implementation in an embedded system context.
The remainder of the paper is organized as follows. Section II presents polar codes and the SC decoding algorithm. In section III, all the optimization techniques that contribute in the speedup of the decoder are detailed. Then, the experimental setup is introduced in section IV. Experimental results are presented in section V. A comparison with previous works is provided in section VI. Eventually, conclusion is drawn in section VII.

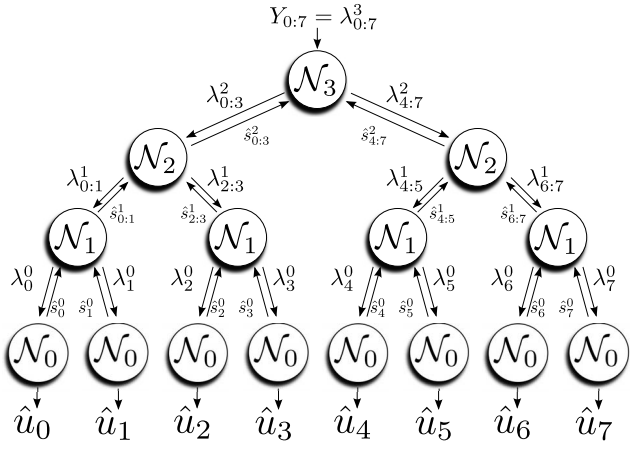

Fig. 1. Recursive tree representation of a $N = 8$ SC decoder.

II. Polar Codes

A. Definition and encoding

Polar codes are linear block codes of size $N = 2^n$, n being a positive integer. In [1], Arıkan defined their construction based on the n^{th} Kronecker power of a kernel matrix $\kappa = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. The encoding process consists in multiplying $\kappa^{\otimes n}$ by a N-bit vector U that includes K information bits and $N - K$ frozen bits that are set to a known value. The location of the frozen bits depends on both the type of channel that is considered and the noise power on the channel. The interested reader should refer to [1] for more details about the construction of polar codes.

B. Successive Cancellation decoding

After being sent over the transmission channel, the noisy version of the codeword X is received in the form of a log likelihood ratio (LLR) vector Y. The decoder successively estimates each bit u_i based on the vector Y and the previously estimated bits $(\hat{u}_{0:i-1})$[1]. In order to estimate each bit u_i, the decoder computes the following LLR value:

$$\lambda_i^0 = \log \frac{\Pr(Y, \hat{u}_{0:i-1} | u_i = 0)}{\Pr(Y, \hat{u}_{0:i-1} | u_i = 1)}. \quad (1)$$

The estimated bit $\hat{u}_i$ is calculated based on the following rule:

$$\hat{u}_i = \begin{cases} 0 & \text{if } \lambda_i^0 > 0, \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

[1] $\hat{u}_{0:i-1} = [\hat{u}_0 ... \hat{u}_{i-1}]$

Due to its sequential nature, the SC decoding algorithm has strong data dependencies that limit the amount of parallelism that can be exploited within the algorithm. As suggested in [5], the SC decoding algorithm can be represented in the form of a tree that is recursively traversed in the following order: root node, left child node, right child node. The SC decoding algorithm is described as a recursively traversed tree (Figure 1). The root node $\mathcal{N}_3$ receives the channel information Y and successively exchange data with the left and right child nodes $\mathcal{N}_2$. The leaf nodes $\mathcal{N}_0$ are called by nodes $\mathcal{N}_1$ and generate the estimation $\hat{u}_i$. Assuming that a non-leaf/non-root node $\mathcal{N}_j$ receives $\lambda_{a:a+J-1}$, with $J = 2^j$, it executes Algorithm 1.

Algorithm 1 Node $\mathcal{N}_j$ update function

Require: $\lambda^j_{a:a+J-1}$

1: **Calculates** $J/2$ **concurrent** f **functions:** $\lambda^{j-1}_i = f(\lambda^j_i; \lambda^j_{i+\frac{J}{2}})$, $a \leq i < \frac{J}{2}$,

2: **Recursively calls the left child node** $\mathcal{N}_{j-1}$ **to get the partial sums:** $\hat{s}^{j-1}_{a:a+\frac{J}{2}-1} = \mathcal{N}_j(\lambda^{j-1}_{a:a+\frac{J}{2}-1})$,

3: **Calculates** $J/2$ **concurrent** g **functions:** $\lambda^{j-1}_i = g(\lambda^j_{i-\frac{J}{2}}; \lambda^j_i; \hat{s}^{j-1}_{i-\frac{J}{2}})$, $a + \frac{J}{2} \leq i < a + J$,

4: **Recursively calls the right child node** $\mathcal{N}_{j-1}$ **to get the partial sums:** $\hat{s}^{j-1}_{a+\frac{J}{2}:a+J-1} = \mathcal{N}_j(\lambda^{j-1}_{a+\frac{J}{2}:a+J-1})$,

5: **Combines partial sums of step 2 and 4:** $\hat{s}^j_i = \hat{s}^{j-1}_i \oplus \hat{s}^{j-1}_{i+\frac{J}{2}}$, $a \leq i < \frac{J}{2}$.

6: **return** $\hat{s}_{a:a+J-1}$

f and g functions are defined as:

$$\begin{cases} f(\lambda_a, \lambda_b) &= sgn(\lambda_a.\lambda_b).\min(|\lambda_a|, |\lambda_b|) \\ g(\lambda_a, \lambda_b, \hat{u}) &= (-1)^{1-2\hat{u}}\lambda_a + \lambda_b \end{cases} \quad (3)$$

III. Speeding up the software polar decoder

This section describes the different optimization techniques that were applied to the SC decoding algorithm in order to implement it on an embedded processor. These optimizations were originally applied to an x86 processor [13]. In this work, we transpose these optimizations to an ARM processor in order to investigate the performance of a software SC decoder on an embedded processor target.

A. Tree cut

In [5], it was shown that some of the computations in SC decoding are not necessary. Let us consider a node $\mathcal{N}_j$ that corresponds to the decoding of a subcode of size $J = 2^j$. Assuming that the considered subcode has a code rate 0, the node $\mathcal{N}_j$ returns $\hat{s}^j_{a:a+J-1} = 0$ regardless of $\lambda^j_{a:a+J-1}$ because $\hat{u}^j_{a:a+J-1} = 0$. For example, in Figure 1, if the left child node of $\mathcal{N}_3$ corresponds to a rate-0 code, *i.e.* $\hat{u}_{0:3} = 0$, one can directly deduce that $\hat{s}^2_{0:3} = 0$ regardless of $\lambda^2_{0:3}$. It significantly reduces the number of computation in SC decoding algorithm. A similar statement can be made if the considered subcode has a code rate 1. In such a case, $\hat{s}_i = sign(\lambda_i)$ and $\hat{u}_i = \hat{s}_i\kappa^{\otimes j}$, with $a \leq i < J$. However, for rate-1 subcodes, the saved LLR computations are counterbalanced by extra hard-decision computation: $\hat{u}_i = \hat{s}_i\kappa^{\otimes j}$. It limits the efficiency of rate-1 code simplification as confirmed by our experimentations. For this reason, the proposed software polar decoder includes this computation reduction only for rate-0 subcodes. Each node in the tree is characterized by its code rate. The code rate can either be $R = 0$ or $R \neq 0$. This information (one bit per node) is stored for each node in a static vector because this is constant for a given polar code. When a node $\mathcal{N}_j$ is called, it retrieves its associated code rate ($R = 0$ or $R \neq 0$) and depending on the value, it performs the associated processing. This simplification means that some nodes are never accessed and can simply be cut from the recursive tree representation.

Algorithm 2 Node $\mathcal{N}_j$ code rate checking function

1: **if** $(R(\mathcal{N}_j) == 0)$ **then**

2: $\quad \hat{u}_{a:a+J-1} = 0$

3: $\quad \hat{s}^j_{a:a+J-1} = 0$

4: **else**

5: $\quad$ Call **Algorithm 1**

6: **end if**

7: **return** $\hat{s}^j_{a:a+J-1}$

B. $\mathcal{N}_3$ node unrolling

Let us consider an SC decoder for $N \gg 8$, the processing of nodes located in the lower part of the tree ($0 < j < 3$) includes a lot of recursive function calls. In general, in the tree representation of the decoder, there are 2^{n-j} nodes of type $\mathcal{N}_j$. Optimizing the execution of these lower level nodes clearly benefits to the overall decoding time. In order to speed up this part of the decoding, the recursive description of node $\mathcal{N}_3$ was completely manually *unrolled*[2]. It allows the compiler to i) avoid multiple recursive function calls ii) statically reorder instructions and iii) memorize frequently accessed intermediate results into registers that are faster to access than a memory. Moreover, the most frequently called functions (e.g. f, g, etc...) are directly written in assembly in order to optimize their execution on the SIMD (NEON) arithmetic and logic units of the ARM processor.

C. Memory sharing

Accessing data stored in the memory is time consuming especially if cache misses occur. As shown in our experimental results, this is especially critical for embedded processors where the amount of cache memory is usually limited. In order to reduce the probability of cache misses, an effort was made to reduce the memory footprint of the decoder. During the decoding of one frame Y, the SC decoding algorithm needs to access (read/write) to:

- $N(\log N)$ LLRs: $\lambda^{0:n-1}_{0:N-1}$,
- N channel values: $Y_{0:N-1} = \lambda^n_{0:N-1}$,
- $N(\log N - 2)$ partial sums: $\hat{s}^{1:n-1}_{0:N-1}$,
- N decoded bits $\hat{u}_{0:N-1} = \hat{s}^0_{0:N-1}$.

Assuming that each data is stored in the form of a 32-bit integer or floating point number, the memory footprint of the decoder

[2]From our experimentations, unrolling nodes of degree $j > 3$ does not significantly improve the speed.

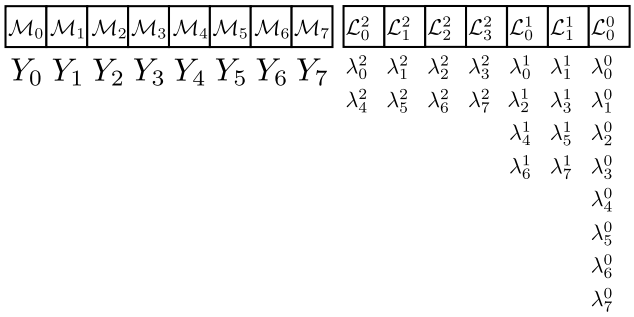

Fig. 2. Optimized Y_i and λ_i^j memory mapping for $N = 8$.

is

$$\mu_A = 8N \log N \quad (Bytes). \tag{4}$$

As shown in [6], it is possible to assign only $N-1$ memory locations for LLRs and $N-1$ for partial sums. In such a memory architecture, each memory location stores different data that are not alive during the same period time of the decoding process. More specifically, the following memory mapping can be applied:

$$\begin{aligned} Y_i &\rightarrow \mathcal{M}_i \\ \lambda_i^{(j)} &\rightarrow \mathcal{L}_{i[2^j]}^{(j)} \\ \hat{s}_i^{(j)} &\rightarrow \mathcal{S}_{i[2^j]}^{(j)} \\ \hat{u}_i &\rightarrow \mathcal{U}_i \end{aligned} \tag{5}$$

The footprint of this memory mapping is

$$\mu_B = 16N - 8 \quad (Bytes). \tag{6}$$

It reduces the memory footprint by a factor $O(\log N)$. An example of the memory mapping is depicted in Figure 2 for $N = 8$. In [6], the reduced memory ensures a lower area of the implemented architecture. In a software description, this memory footprint reduction tends to reduce the cache misses even for large N.

D. Data packing

1) Bits packing: In the previously presented memory mapping, for each memory location S_i^j and U_i, a single bit is stored as an integer. It is suboptimal in the sense that a binary value is stored in a 32 bit-wide location. In order to compress the memory footprint of the software decoder, packets of 32 bits are stored in a single memory location. It reduces the size of the memory dedicated to S_i^j and U_i by a factor 32. Moreover, 32 memory accesses are replaced by a single memory access and some masking operations. These masking operations are necessary to retrieve the 32 individual bits stored at the same memory location.

2) LLRs packing: In a basic description of the decoder, the LLRs are represented in 32-bit floating-point format. BER simulations show that 8-bit fixed-point format is sufficient to obtain similar decoding performance as shown in [12]. One can actually notice a slight performance degradation for large code lengths. To illustrate this observation, BER performance curves are provided in section IV. Data packing is also applied to LLR storage in order to further reduce the memory footprint.

If the software decoder can processes F frames in parallel (as it will be discussed in the next subsection), then the memory footprint of the decoder with memory sharing and data packing includes $F \times N$ bytes for channel values, $F \times N$ bytes for LLR values, $\frac{F \times N}{8}$ bytes for partial sums and $\frac{F \times N}{8}$ bytes for decoded bits. The total memory footprint is then:

$$\mu_C = \frac{9 \times F \times N}{4} \quad (Bytes) \tag{7}$$

For a codelength of $N = 32768$, the memory sharing combined with data packing reduces the memory footprint by a factor $\frac{\mu_A}{\mu_C} \approx 60$. Table I gives the memory footprint of the software polar decoder for different code lengths and different parallelism level values F.

E. From x86 processor to ARM processor

In [12], the SIMD facilities of Intel processors are exploited to speedup a software polar decoder. A modified version of SC decoding is used [14]. In this modified version, subcodes are classified into different categories (repetition codes, single parity-check (SPC) codes and repetition-SPC codes). Instead of applying SC decoding on these particular subcodes, some more parallelizable algorithms are applied. This approach enables a better computation vectorization of the whole processing.
In [13], instead of using the intra frame parallelism, the *inter frame* parallelism is exploited. Thus, several independent frames are decoded in parallel. The main advantage of this approach is that the processor SIMD unit is always fed with data which garantees a better utilization rate and enables multi-Gbps decoding.
In this work, the same principle is applied to an ARM processor with NEON instruction set. Processing F frames in parallel requires to interleave the different frames in such a way that F contiguous data can be accessed from a single load instruction. This data alignement reduces the amount of memory adress computations. Before the decoding process, Y_i values has to be reordered (interleaved). Then, the decoding process is executed on F frames in parallel. Finally, after the decoding process, U_i values are reordered back (deinterleaved). An illustration of the whole decoding procedure is detailed in Figure 3.

IV. Experimental setup

The targeted device is a Quad-Core Cortex-A9 ARM processor. Each core has 32KB of L1 data cache and 32KB of L1 instruction cache. A 1MB unified L2 cache memory is shared between processor cores. The working frequency is 1.4 GHz when a single processor core is used, 1.3 GHz otherwise. The ARM processor is included in a NVIDIA Tegra 3 development kit. In addition to the ARM Quadcore processor, the Tegra3 SoC includes a 2GB DDR3 RAM, a GPU and some application specific accelerators.
The optimized software SC decoder was described in standard C language. The LLR values are represented in fixed-point format (8 bits) in such a way that $F = 128/8 = 16$ frames can be processed in parallel with the NEON instruction set. NEON instruction-set is used according to C built-in instructions available in the GCC toolchain. The source code was compiled with GCC compiler (version 4.6)

TABLE I. MEMORY FOOTPRINT (μ_C) OF THE DECODER IN KBYTES (ROUNDED TO THE UPPER VALUE).

N	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
ARM ($F = 1$)	2	3	5	9	17	34	68	136	272	544	1088	2176	4352
ARM-NEON ($F = 16$)	10	19	38	76	152	304	608	1216	2432	4864	9728	19456	38912

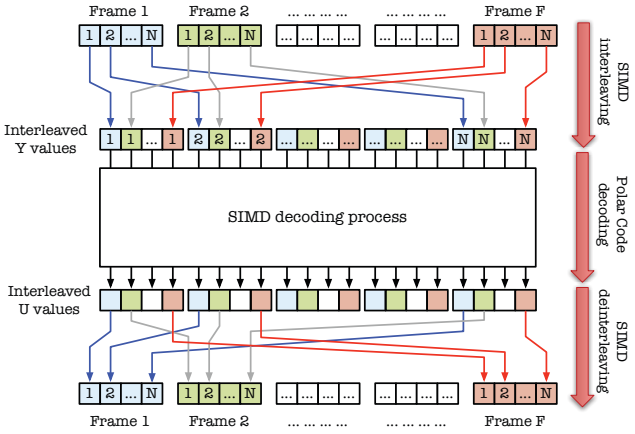

Fig. 3. SIMD decoding procedure

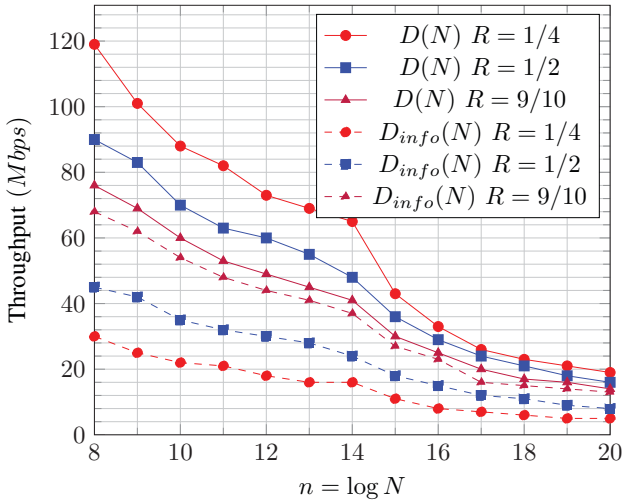

Fig. 4. Measured coded and information throughputs on a single core of the ARM processor with the NEON instruction set.

and the following compilation options were applied: `-O3 -mfpu=neon -funroll-loops -march=native -fprefetch-loop-arrays -fopenmp`.
The throughput measurement procedure is as follow. A large set of random information vectors (U) is generated, then each vector is encoded (X) and mapped to BPSK. Gaussian noise is added (Y) in order to emulate an AWGN channel. A high precision counter provided by the Boost Chrono library is used to measure the whole decoding process execution time. This process is repeated during a period of 60 seconds. The measure of the decoding time is averaged over the whole set of decoded frames. The time required to decode F frames includes *i)* the writing of the F frames into the decoder memory, *ii)* the decoding of F frames with the SIMD decoder, *iii)* the writing of the F estimated codewords and *iv)* the interelaving/deinterleaving functions.

V. EXPERIMENTAL RESULTS

In this section, the impact of the following parameters on the throughput of the proposed software decoder is reported: the codelength N, the code rate R and the number of cores M.

A. Impact of the code length N

Error correction performance of Polar Codes improves with the code length N. It is then necessary to evaluate the impact of the codelength value on the throughput. As shown in Figure 3, the whole decoding process includes the decoding function and the interleaving/deinterleaving functions. The latency of the decoder is the time required to interleave, decode and deinterleave F frames. The processing of one frame includes $O(N \log(N))$ computations, memory accesses and control statements. Since F frames are processed in parallel, the latency is invariant with F: $L(N) = O(N \log(N))$. Within $L(N)$ seconds, $F.N$ bits are decoded and the resulting *coded throughput* is:

$$D(N) = \frac{F.N}{L(N)} \approx O\left(\frac{F}{\log N}\right). \quad (8)$$

This equation is valid if no cache miss occurs and the processor frequency is constant. Figure 4 shows the coded throughput D for different values of N and R. For a fixed code rate R, the throughput decreases with $\log(N)$. One should notice that from $n = 14$ to $n = 15$, the throughput drops. This can be explained by the increasing footprint of the decoder which becomes larger than the cache and causes more cache misses during the decoding process. It means that cache misses may occur for the codes having a large code lengths. Data that cannot be stored in the cache memories are stored in the system memory outside of the ARM chip. Such external memory accesses are very costfull in terms of clock cycles. It explains the change in trends observed in Figure 4. This interesting fact shows that the memory architecture is one of the limiting factor of embedded processors for polar code decoding. Reducing the memory footprint of the decoder is then a key challenge in order to devise efficient software implementations of channel decoders in general.

B. Impact of the code rate R

As explained in section III-A, when a rate-0 subcode is reached in the tree representation, all computations related with leaf nodes can be avoided. It enables a reduction in the total number of computation to be performed during the decoding of one frame. As the code rate decreases, the number and the size of rate-0 subcodes tends to increase. It means that the total

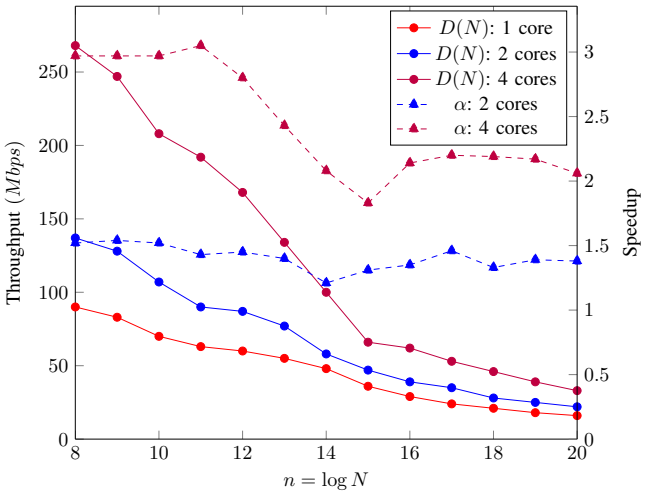

Fig. 5. Coded throughput for different M values.

number of computation is lower for low rate codes. Figure 4 shows the coded throughputs for 3 different code rates: $R = \{1/4, 1/2, 9/10\}$. As expected the highest coded throughput is reached by the lowest rate polar code ($R = 1/4$). The coded throughput represents the data rate at the input of the decoder. Another relevant parameter is the *information throughput* that correspond to the information data rate at the output of the decoder:

$$D_{info}(N) = R.D(N) \tag{9}$$

The information throughput is also reported in Figure 4. One should notice that the trend is reversed in the sense that the highest information throughput is reached when processing codes with higher code rates R. It means that the computation reduction induced by lower rate codes is counter balanced by the reduction in code rate.

C. Impact of the multithreading mode

Modern embedded CPUs include multiple processing cores. The performance of computation intensive algorithms running on a multi-core processor depends on the number of available cores. It is also highly impacted by the memory architecture of the processor. General purpose CPU includes a large amount of memory on 3 levels. For embedded processor, the amount of memory and the number of levels of cache memory are usually lower. It generates more cache misses and limits the speedup brought by multicore processing. In the targeted ARM processor, each of the 4 cores has a dedicated 32KB L1 cache and the 4 cores share a 1MB L2 cache.
OpenMP directives [15] are included in the description of the SC decoder to enable multicore execution. One thread is used per available core. Assuming M cores are available, the M threads are first created, then each core processes $F = 16$ frames in parallel. Finally threads are synchronized before a new set of F frames is launched. The maximal theoritical speedup $\alpha_{max} = M$ could be reached if i) the software decoder fits into the dedicated L1 cache *ii)* the synchronization time is negligible compared to the decoding time and *iii)* the execution of unrelated tasks (Operating System routine for example) does not interfer with the execution of the program. Obviously these conditions are never met and the obtained speedup is usually lower than M. Figure 5 shows the measured throughput $D(N, M)$ of the SC decoder for $M = \{1, 2, 4\}$ cores and a code rate $R = 1/2$. The speedups $\alpha = D(N, M)/D(N, 1)$ are also reported. A maximum speedup of $\sim$3 is reached for $256 \leq N \leq 2048$ and $M = 4$. The maximum speedup decreases with N for the three aforementioned reasons. Despite the decreasing speedup, these experiments show that an SC decoder implemented on a multicore embedded processor can reach throughputs superior to 100Mbps for $256 \leq N \leq 16384$.

VI. Comparison with similar works

To the best of our knowledge, this is the first implementation of a software polar decoder on an embedded processor. In this section, the proposed software implementation is compared with LDPC decoders implemented on embedded processors.

A. LDPC decoder on an ARM processor

In [16], the NEON instruction set of an ARM processor is used to decode LDPC codes. The SoC used in this work is a Samsung Exynos 4412. It includes a quad-core Cortex-A9 ARM processor which has characteristics close to the processor used in this work. Indeed, each core has a 32KB L1 data cache and a 32KB L1 instruction cache. Moreover, the four cores share a 1MB unified L2 cache. The maximum working frequency is 1.6 GHz (1.15 times higher than our targeted processor). A Min-sum LDPC decoder is used with 20 iterations of flooding scheduling. The targeted code is the long frame (64K code) of the DVB-S2 standard with rate 1/2. The decoding performance of such a decoder is shown in Figure 6. The decoding performance of various rate-1/2 polar codes under SC decoding is also reported. These curves were generated with the proposed software decoder with 8-bit precision fixed-point representation. $n = 14$ polar code outperforms the decoder implemented in [16]. In terms of throughput the DVB-S2 software decoder has a coded throughput of 3.2 Mbps on 4 cores while our SC decoder run at 48 Mbps on a single core and 100 Mbps on 4 cores. In [16], the throughput is not reported for a larger number of iterations. In order to achieve the performance of the $N = 64K$ LDPC code with 50 decoding iterations, the polar decoder requires a larger codelength: $n = 20$ as shown in Figure 6. This polar decoder has a throughput of 16 Mbps on a single core and 33 Mbps on 4 cores. This comparison suggests that polar codes are competitive candidates in terms of error correction performance and throughput when targetting embedded processors.

B. LDPC decoder on a Cell processor

In [17], various LDPC decoders are implemented on a Cell processor for the Wimax standard. This Cell System-on-Chip is composed one PowerPC processor and Syngergistic Processor elements (SPE). This massively parallel system uses a SIMD-based programing model where instructions are parsed and fed into the SPEs from the PowerPC. LDPC decoder implemented

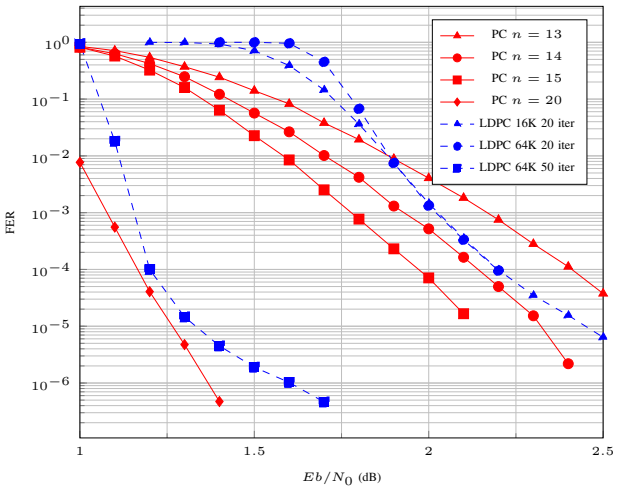

Fig. 6. Decoding performance comparison of DVB-S2 LDPC codes and R=1/2 Polar Codes

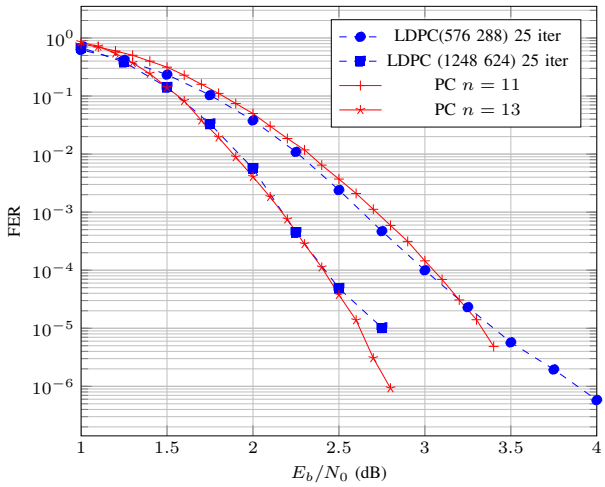

Fig. 7. Decoding performance comparison of Wimax LDPC codes and R=1/2 Polar Codes.

on the CELL processor uses six SPEs to process 96 (6×16) codewords in parallel, achieving throughput up to 72-80 Mbps, for ten decoding iterations. The working frequency of the system is 3.2 GHz for the $45nm$ CELL processor. It consumes 20 W and it has a die size of $115mm^2$.

In Figure 7, we show that PC(2048,1024) and PC(8192,4096) are sufficient to achieve the same performance as LDPC(576,288) and LDPC(1248,624) respectively[3]. In terms of coded throughput these LDPC decoders reach 32.6 Mbps on a Cell processor while our PC decoders run at 63 Mbps and 55 Mbps on a single core of an ARM processor. They reach 196 Mbps and 134 Mbps when using the 4 cores. One should further notice that the ARM processor used in this work does not belong to the last generation of ARM processors. The more recent Cortex A15 should improve the presented performance even further. Moreover, the Cell/BE processor has a much higher processing power compared to ARM processor.

VII. Conclusion

In this paper, we present the first software implementation of a polar code decoder on an embedded processor. The cache memory available in embedded processors is lower than x86 processors which limits the maximum reachable throughput. Despite this lack of memory, it is shown that the parallel processing facilites included in modern embedded processors can be used to efficiently implement software polar decoders. Moreover, the proposed software implementations compare favorably with existing software implementation of LDPC decoder on similar targets. The proposed implementation run upto 10 times faster with equivalent decoding performance.

References

[1] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *Information Theory, IEEE Transactions on*, vol. 55, no. 7, pp. 3051–3073, 2009.

[2] N. Hussami, S. Korada, and R. Urbanke, "Performance of polar codes for channel and source coding," in *Information Theory, 2009. ISIT 2009. IEEE International Symposium on*, 2009, pp. 1488–1492.

[3] I. Tal and A. Vardy, "List decoding of polar codes," in *Proc. ISIT*, 2011.

[4] K. Niu and K. Chen, "Crc-aided decoding of polar codes," *Communications Letters, IEEE*, vol. 16, no. 10, pp. 1668–1671, 2012.

[5] A. Alamdar-Yazdi and F. R. Kschischang, "A simplified successive-cancellation decoder for polar codes," *IEEE Communications Letters*, Dec. 2011.

[6] C. Leroux, I. Tal, A. Vardy, and W. Gross, "Hardware architectures for successive cancellation decoding of polar codes," in *2011 IEEE ICASSP*, May 2011.

[7] C. Leroux, A. J. Raymond, G. Sarkis, I. Tal, A. Vardy, and W. J. Gross, "Hardware implementation of successive-cancellation decoders for polar codes," *Journ. of Sig. Proc. Syst.*, Dec. 2012.

[8] C. Leroux, A. Raymond, G. Sarkis, and W. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *Signal Processing, IEEE Transactions on*, 2012.

[9] A. Mishra, A. J. Raymond, L. G. Amaru, G. Sarkis, C. Leroux, P. Meinerzhagen, A. Burg, and W. Gross, "A successive cancellation decoder ASIC for a 1024-bit polar code in 180nm CMOS," in *Asian Solid-State Circuits Conference*, Nov. 2012.

[10] C. Zhang and K. Parhi, "Low-latency sequential and overlapped architectures for successive cancellation polar decoder," *IEEE Transactions on Signal Processing*, 2013.

[11] A. Pamuk, "An FPGA implementation architecture for decoding of polar codes," in *Wireless Communication Systems (ISWCS), 2011 8th International Symposium on*, Nov. 2011.

[12] P. Giard, G. Sarkis, C. Thibeault, and W. Gross, "A fast software polar decoder," *arXiv:1306.6311*, Jun. 2013.

[13] B. L. Gal, C. Leroux, and C. Jego, "Multi-gb/s software decoding of polar codes," *submitted to IEEE Transactions on Signal Processing*.

[14] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast polar decoders: Algorithm and implementation," *IEEE Journal on Selected Areas in Communications*, 2014.

[15] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2008.

[16] S. Gronroos and J. Bjorkqvist, "Performance evaluation of ldpc decoding on a general purpose mobile cpu," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013.

[17] G. Falcao, V. Silva, J. Marinho, and L. Sousa, "Ldpc decoders for the wimax (ieee 802.16e) based on multicore architectures," *in WIMAX New Developments. Upena D Dalal and Y P Kosta (Ed.)*, 2009.

[3] Wordlengths used are: 6 bits for channel information and 8 bits for internal decoder computations.