Cyrille Chavet · Philippe Coussy *Editors*

Advanced Hardware Design for Error Correcting Codes

Springer

Advanced Hardware Design for Error Correcting Codes

Cyrille Chavet • Philippe Coussy
Editors

Advanced Hardware Design for Error Correcting Codes

Springer

Editors
Cyrille Chavet
Université de Bretagne Sud
Lorient, France

Philippe Coussy
Université de Bretagne Sud
Lorient, France

ISBN 978-3-319-10568-0 ISBN 978-3-319-10569-7 (eBook)
DOI 10.1007/978-3-319-10569-7
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014951358

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

For many years, experts more expert than the rest have regularly heralded the end of research focused on the physical layer of telecommunications. Some claim that the best has already been delivered from the promises offered by the theory of communication, others say that the theoretical limits predicted will never be reached by simple means. As Costello and Forney explained in an award winning IEEE article [1], this pessimistic standpoint is nothing new and some were already proclaiming "Coding is dead" in the early 1970s, only 20 years after the pioneering work of Claude Shannon. Other experts, this time in the field of microelectronics, have also regularly announced the end of CMOS technology, starting as early as the mid-1980s when the submicron barrier for mass production seemed insurmountable to some. "CMOS is dead" was also commonly heard.

Fortunately, these doubts were swept away each time they were raised. And this was often because of the need for increasingly demanding telecommunications: farther, faster, more reliable, that microelectronics increased its efforts in the miniaturization of components. Conversely, the steady progress of the semiconductor industry has opened the way to new processing information algorithms unforeseen at the time of the first generations of integrated circuits. Information, as understood by Shannon, and the transistor were born around the same time in the legendary Bell Labs and, from that time, have continued to join hands to lead to ever more effective telecommunications systems which have become indispensable in our daily lives.

Among the processes made possible today by high density integration on silicon, distributed error correction coding (or channel coding) came to occupy a place of prime importance. To put it simply, distributed coding is to monolithic coding what a combination of small mathematical relationships is to one complex equation. It was by adopting the point of view of distributed computing that error correction coding proved able to find its best practical solutions to achieve optimality, or nearly. Rather than trying to build a codeword by a single coding operation and recover it through a single decoding step, it is wiser to adopt the strategy of "divide and rule", not at the cost of lesser performance, quite the contrary.

In the early 1990s, the competition (all virtual) between monolithic coding and distributed coding in the race for optimality designated its winner. On the one hand, a team from the prestigious Jet Propulsion Laboratory in Pasadena was working to develop a Viterbi decoder for a 16384-state convolutional code: the Big Viterbi Decoder (BVD) [2]. It had to consist of 256 identical integrated circuits each processing 64 states, plus additional control circuits. The correction power significantly exceeded the state of the art but an entire table was needed to lay the decoder. On the other hand, in a little known French laboratory, an electronics engineer wondered whether two small convolutional codes, typically of 8 or 16 states, associated in an original way and iteratively decoded one after another could not do better than the Californian code. The answer was affirmative: three integrated circuits (same number as iterations) were enough to provide better correction power than the BVD.

A large part of the community of digital communications therefore became interested in this new way of building a redundant code that its inventor [3] later named turbo code [4] to keep it significantly shorter than "parallel concatenation of recursive systematic convolutional codes decoded iteratively". It also provided the opportunity to take a new look at and give a new impetus to LDPC codes [5, 6] to which many researchers turned their attention, whether for optimization or implementation. Different distributed structures, in parallel, in series or both together, were proposed one after the other. From the most compact of distributed codes—a turbo code with only two component codes—to more distributed—LDPC codes or the most recent and promising polar codes [7]—all kinds of solutions were possible. In addition to studies on "modern coding" [8], the philosophy of decoding by message passing was also expanding its ramifications to applications other than channel coding, for example equalization [9, 10], demodulation [11] or joint source-channel coding [12]. "Do not lose any of the pieces of information available in the receiver whatever the level" became the leitmotif of many researchers.

Some were also questioning what once had been considered absolute certainties: but no, on balance, coding is not only a matter of mathematics. Because information theory was built on non-trivial mathematical concepts, such as entropy and mutual information, it was indeed long believed that practical solutions would be exclusively provided by mathematics. But math is especially used to justify and set parameters, rarely to create and build. While algebra, probability and graph theory continue to be part of the arsenal of skills of engineers and researchers in communication technologies, other knowledge and skills have also become indispensable: computer science, electronics, and, in particular, parallel architectures needed to obtain high throughputs. The days when it was legitimate to invent a code or any algorithm without proposing practical ways of decoding or implementation are over. Not only must processes be compatible with what electronics can provide but other significant constraints such as energy consumption in embedded systems or the speed of information transfer in highly distributed structures can be crucial.

The proliferation of new applications (very high throughput cellular systems, sensor networks, Internet of Things, etc.) and the demand for improved performance are still ongoing challenges. It is no longer just a matter of bits per second per hertz

or bit error rate; now it is also necessary to consider other criteria such as joules per bit (in transmission as in reception processing), flexibility and interoperability. A new generation of researchers has emerged, mastering at the highest levels the interdisciplinarity necessary to cope with these multiple constraints. Some of these researchers have come together to write this book with the latest ideas and developments in the design of circuits for error correction encoding and decoding. Tomorrow's telecommunications are in their hands and we can certainly say alongside them: "Coding and CMOS are still alive, and for a long time to come".

[1] Costello DJ, Forney GD (2007) Channel coding: the road to channel capacity. Proc IEEE 95(6):1150–1177
[2] Statman J, Rabkin J, Siev B (1989) Big Viterbi decoder (BVD) results for (7,1/2) convolutional code. TDA Progress Report 42–99, JPL, November 1989
[3] Berrou C (1991/1995) Error-correction coding method with at least two systematic convolutional codings in parallel, corresponding iterative decoding method, decoding module and decoder. Patents no 9,105,280 (France, April 1991), no 5,446,747 (USA, August 1995)
[4] Berrou C, Glavieux A, Thitimajshima P (1993) Near Shannon limit error-correcting coding and decoding: turbo-codes. In: Proceeding of IEEE ICC '93, Geneva, pp 1064–1070, May 1993
[5] Gallager RG (1962) Low-density parity-check codes. IRE Trans Inf Theory IT-8:21–28
[6] MacKay DJC, Neal RM (1995) Good codes based on very sparse matrices. In: Boyd C (ed) Cryptography and coding 5th IMA Conference, Lecture notes in computer science, no 1025. Springer, Berlin, pp 100–111
[7] Arikan E (2009) Channel polarization: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. IEEE Trans Inf Theory 55(7):3051–3073
[8] Richardon T, Urbanke R (2008) Modern coding theory. Cambridge University Press, New York
[9] Douillard C, Picard A, Didier P, Jézéquel M, Berrou C, Glavieux A (1995) Iterative correction of intersymbol interference: turbo-equalization. Eur Trans Telecom 6(5):507–511 (special issue on turbo decoding)
[10] Laot C, Glavieux A, Labat J (2001) Turbo equalization: adaptive equalization and channel decoding jointly optimized. IEEE J Select Areas Commun 19(9):1744–1752
[11] Hoeher P, Lodge J (1999) Turbo DPSK: iterative differential PSK demodulation and channel decoding. IEEE Trans Commun 47(6):837–843
[12] Hagenauer J, Görtz N (2003) The turbo principle in joint source channel coding. In: Proceeding of ITW 2003, Paris, pp 275–278, April 2003

Claude Berrou
May 2014

Contents

Chapter 1
User Needs

David Gnaedig

TurboConcept is an industry-reference provider of Intellectual Property Cores (IP Cores) for advanced Forward Error Correction (FEC) techniques (turbo codes and LDPC codes). We propose IP Core products, which offer to our customers the best trade-offs between error correction capability, throughput, silicon cost, and power consumption. Since 2007, TurboConcept is part of the Newtec group, specializing in satellite communications equipments and systems. We have developed IP cores implementing encoders and decoders addressing most of the families of error correcting codes:

- Turbo codes: since the development of the first DVB-RCS turbo decoder in 1999 (a duo-binary turbo code), we have developed IPs for all flavors of convolutional turbo codes that are specified in standards such as the CCSDS, WCDMA, Homeplug AV, WiMAX, LTE.
- LDPC codes since their adoption in 2002 by the DVB-S2 standard. We have naturally extended this IP to cover also DVB-T2, DVB-C2, and have then developed products for WiMAX, G.hn, and more recently WiFi 802.11ac.
- Turbo product codes: two- and three-dimensional product codes.
- Convolutional codes for WiFi, LTE, WCDMA.
- BCH codes, that are used in concatenation with LDPC codes in DVB-S2 standard for instance but also for other proprietary concatenation schemes.

We also propose proprietary error correcting codes based on either turbo or LDPC codes for various applications: satellite communication, wireless back-haul, ...

Our FEC IP cores target both ASIC and FPGA designs. During the first years of the company most IP cores were designed for FPGA devices. But since 2005 with

D. Gnaedig (✉)
TurboConcept, 185 rue Joseph Fourier, 29280 Plouzane, France
e-mail: david.gnaedig@turboconcept.com

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_1

the emergence of the WiMAX market and later on with the throughput increase of 3G cellular systems, and in particular the evolution toward LTE, we have developed a range of IPs that are integrated into ASIC designs. Our sales volume is today a balance between FPGA and ASIC users. It is also worth noting that for some markets (for instance LTE base station) we propose two different IP cores, one optimized for ASIC designs and another optimized for FPGA design. For ASIC, the IP is optimized for best area and specifically memory area and low power consumption. For FPGA designs, the IP architecture takes advantage of specific resources available for "free" in FPGA devices: dual port RAMs, a large number of registers enabling high pipe-lining and thus very high clock frequency, multiple clock domains, ...

Our story starts with turbo codes and related iterative decodable codes, back in the second half of the 1990s. At that time, it was a real challenge to have an FPGA or an ASIC hosting such a complicated function as a turbo decoding, especially when compared to legacy convolutional codes. The first prototype chip initiated by turbo codes inventors was based on a "one-chip-per-iteration" pipelining [5]. Moore's law is obviously a great enabler in the wide adoption of turbo and LDPC codes: it helped greatly to minimize the complexity overhead of iterative decoding, as compared to legacy FECs, even in the context of ever-higher-throughput applications. Algorithmic advances also helped significantly: the transposition of the probabilistic decoding equations into the logarithmic domain led to low complexity algorithms to decode efficiently turbo codes (log-MAP [1]) or LDPC codes (min-sum [2]). At the architectural level, significant breakthroughs have enabled to develop low complexity and high throughput decoder architectures. First, sliding window algorithms [3] enabled to reduce drastically the required memory of the BCJR algorithm while maintaining acceptable performance. Second, the concept of parallel soft-in–soft-out implementations accessing in concurrence to a shared memory, applicable to both turbo and LDPC code decoding enabled to reach several tens of Mbits/s and up to several Gbits/s for LDPC codes using actual technologies. Additionally, the concept of "shuffled scheduling" (also referred to as "layered" or "turbo" scheduling) of LDPC codes has also contributed to almost dividing by two the required number of iterations and thus increasing the throughput equivalently. Finally, resolving memory access issues has been a critical issue in massively parallel architectures. It has been tackled by taking into account the constraints of the architecture early in the design of the code. Such architecture-aware code design techniques have been used for:

- Turbo codes through the design of a structured interleaver (e.g., DVB-RCS code with the ARP interleaver, or quadratic interleaver used in the LTE specification. This structure of the interleaver can be exploited by the parallel decoding architecture to enable collision free memory accesses.
- LDPC codes through the design of a prototype parity matrix which specifies the complete parity matrix of the code in a condensed way. The expansion factor that enables to derive the binary parity matrix from the prototype matrix offers a natural level of parallelism that is then exploited when designing high throughput architectures.

The choice of a given code for a given application is usually driven by the requirements of the application in terms of latency, SNR operation range, target BER, flexibility (block size, code rates, ...). But we have also encountered cases where the choice of an error correcting code is driven by marketing objectives rather than technical reasons. For instance LDPC codes are seen sometimes as a "new" technology while turbo codes are present since many years into various applications and therefore LDPC may be selected by customers even if turbo codes may have superior error decoding performance for this application. A single code family that would outperform other codes over all possible applications does not exist yet and will to our opinion never exist. Therefore, a trade-off shall be made depending on the most important application requirement. Usually, turbo codes have superior BER performance for small block sizes and low code rates and enable a large flexibility both in block size (using a parametric interleaver) and code rates (through puncturing). Turbo product codes have very good performance in the high code rate region typically around above 0.80 with a very low complexity. They are attractive for very high throughput applications, owing their inherent parallelism ability, but they offer a poor flexibility. Convolutional codes decoded by the Viterbi algorithm have their interest for very short block size (typically a few tens of bits) and/or for applications where the critical factor is the lowest latency due to the fact that they do not require an iterative decoding scheme. LDPC codes have better performance for very large (typically a few tens of thousands bits) to medium block sizes (around a few thousands bits) and have the advantage of enabling high throughput parallel implementation with an affordable complexity. They lack however in a large flexibility as each block size-code rate combination requires to specify one parity check matrix. Also encoding complexity of LDPC codes grows quadratically with the block size while the encoding complexity of convolutional code and convolutional turbo code grows only linearly with the block size. This apparent complexity drawback can however be greatly mitigated by introducing specific structures in the parity check matrix of LDPC code like a so-called "staircase" structure of the parity bits sub-matrix.

These general trends are continuously evolving due to large research efforts devoted to code design. LDPC codes are getting more and more efficient with small block sizes especially when considering the non-binary LDPC codes. New interleaver design techniques for turbo code bring significant improvement in the error floor region over previous generation, especially for high code rates, one of the turbo code weaknesses. In addition to the evolution of the now "old" turbo and LDPC code families, brand new code structures are being introduced: spatially couples LDPC codes, polar codes which are proven to achieve (and not only approach) capacity of a given channel.

Once the code family is selected, to define a set of codes suitable for the application, other key parameters have to be determined in light of their impact on the implementation complexity. For turbo codes, this includes the choice of the recursive systematic convolutional code (larger memory induces lower error floor but at the cost of higher complexity), the design of the interleaver that influences greatly the performance in the error floor region, the puncturing scheme. For an

LDPC code, the parity check matrix density has an influence on the error floor but also on the convergence and on the complexity. Also a specific structure in the parity bits region of the parity check matrix is helpful to enable simple encoding scheme.

With standardized applications, the choice of the code itself is obviously not part of our degrees of freedom, but a constraint to which the designed IP core product must comply with. In light of our implementation expertise, we see however how choices made on code design may be very helpful to reduce implementation complexity without sacrifying the error decoding performance. For example, for DVB-S2 codes, there exists the well-known issue of double-diagonal events present in the protograph matrix. Resolving this issue can be performed with various architectural solutions that have an impact on the implementation complexity, throughput, and/or performance. Therefore, if double-diagonal events can be avoided when designing the code, it would be beneficial for enabling low complexity LDPC decoder architectures. An active participation to standardization bodies through the proposition of specific coding schemes is the natural way to influence the choice of the channel coding scheme. To this end, TurboConcept has participated to several standardization processes: DVB-RCS, DVB-S2, and more recently to DVB-SX (as part of Newtec).

Proprietary applications give a larger degree of freedom in the code design and the adaptation of the code design to the target hardware implementation. These include satellite communication systems, wireless backhaul, magnetic storage (hard disk drives), military and governmental... We have developed coding schemes (association of code and modulation) for some of these applications.

When designing products incorporating error correcting codes we need to take into account three typical constraints: throughput, area, and power consumption. First, in terms of throughput we saw the demand for increasing throughput from a few Mbits/s in the early 2,000 years (e.g., in cellular, satellite communications applications) to today's several hundreds of Mbits/s in most wireless applications. Our latest products are scaled to offer several Gbits/s in a wireless physical layer system. Optical links and other markets have even higher throughput demands, but we are not addressing them specifically up to now. Second, for a given throughput requirement, a low implementation area is always desirable for obvious cost reasons. The area constraint greatly varies on the market. Indeed the area being mostly a cost issue, the impact of the area is essentially linked to the other cost elements from the application (e.g., cost of the radio bandwidth, number of users, other operational costs). As an example, if we consider the satellite market, the hub operating the network can afford a large (and expensive) FPGA and therefore a code of higher implementation complexity for the return link. The gain in signal-to-noise ratio translates into more available capacity and thus additional users for the same satellite frequency band. This naturally induces an increased profitability. On the other side, for a consumer equipment where the cost of implementation is of primary interest, the constraint on the area is more stringent. On an ASIC target, the area is mainly driven by the memory area and therefore, the size of the code impacting largely the memory area is an important trade-off between the implementation complexity and the error decoding performance of the code. For an FPGA implementation,

the decoder needs to fit into a low-end FPGA with limited resources (logic and memory). Assuming a throughput from a few tens of Mbits/s to a few hundreds of Mbit/s turbo codes, LDPC code can today be implemented even in low cost FPGAs. Finally, power consumption is obviously getting more and more important, and the relative importance of low power aspects is increased for mobile equipment. We characterize our products by actual numbers (technology dependant) but also by using some design rules and guidelines that ensure the core is not wasting power useless (systematic use of enable signals propagated along the data path, no free-running logic, minimal access to memory blocks, ...).

When designing our products especially when targeting very high throughput architectures we faced several algorithmic and architecture problems that needed to be solved efficiently. On the algorithmic side, increasing the throughput of decoder requires specific techniques in order to maintain good error decoding performance and fast convergence. For example for convolutional turbo code, dealing with very high code rates induces specific algorithm design as the conventional sliding window BCJR algorithms using acquisition for initializing border state metrics are not efficient (and even useless) [4] when code rates grow above 0.95, as it is the case of HSPA. For LDPC codes, higher throughput requires a higher level of parallelism that makes more challenging the selection of a good scheduling for layered decoding architectures. On the architectural side, developing high throughput decoders means that the interfaces and the interleaver (in a BICM scheme) shall support the same level of throughput. Therefore the requirement for high throughput induces to design high speed parallel interleavers. With the increase of the throughput requirement in the future this issue is getting more and more complex to solve because contrary to the code design that have been performed in light of parallelism constraints, it is rarely the case for the associated external interleaver. One last constraint is related to the validation of the performance for FER as low as 10^{-11}. This performance validation is necessary since implementation and especially fixed point representation may introduce a floor, even if the code itself has no floor, This is usually achieved by using an FPGA board able to simulate at throughput of several Gbits/s.

Advanced error correcting codes have made significant progress over the last 20 years and are now used in a large range of applications. There are still areas that need substantial improvements. First, the choice of a FEC coding scheme is often based on some simplified channel modeling (AWGN being the simplest commonality). More progress can be made by considering a refined channel model of the application and to optimize the code in light of this channel model, which is not a simple task. Indeed, refining the channel model often results in a (much) larger design space (e.g., phase noise parameters, multi-path characteristics, non-linearity models). More theoretical methodologies need to be developed in order to find good codes in this context. Second, another important aspect is to find techniques for predicting and designing codes for high order modulations especially in the error floor region. Bit interleaved coded modulation has been used in the past as a mean to design independently the code and the modulation and to achieve good performance on high order modulations. But this technique does not address the optimization

of the performance in the error floor region. Finally, techniques for finite length optimization of codes are still an area that needs to be explored. The techniques that are usually used to characterize code ensembles assume infinite block sizes and ideal BP decoder that does not suffer from correlation due to cycles in a real iterative decoder. Finding optimal code for finite length code is still a challenging problem.

The future challenge that needs to be tackled by the next generation codes and architectures is flexibility. Not flexibility in the sense of an universal decoder that would supports all types of codes of all possible standards. This kind of universal decoder does not seem today to be an industrial requirement, and it is sometimes more complex than using dedicated and optimized cores, one per application. By flexibility, we mean the ability for the code and the decoder to adapt dynamically to changing channel conditions in order to always obtain the best error decoding performance while minimizing power consumption. One mean to achieve this objective is to design codes that can be decoded algebraically when the SNR is high thus enabling high throughput and low power. When the SNR is low an iterative decoder would be used in order to achieve performance close to the Shannon limit. Algorithms such as the bit-flipping algorithm for LDPC codes seem to be very promising in this sense.

In conclusion, we envisage the evolution of error correction coding driven by three main requirements. First, continuous increase of throughput requirements: next generation broadband wireless access systems target maximal data rate in the order of 1 Gbits/s on a handheld device. Second, reducing power consumption will become a major requirement even for non-battery powered applications, and finally, improved error decoding performance (closer to Shannon capacity) by taking into account refined channels models during the code design stage.

References

1. Robertson P, Villebrun E, Hoeher P (1995) A comparison of optimal and sub-optimal decoding algorithm in the log domain. Proceedings IEEE international conference on communications, Seattle, June 1995, pp 1009–1013
2. Fossorier M, Mihaljevic M, Imai H (1999) Reduced complexity iterative decoding of low-density parity check codes based on belief propagation. IEEE Trans Commun 47:673–680
3. Viterbi AJ (1998) An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes. IEEE J Sel Areas Commun 16:260–264
4. Boutillon E, Sanchez-Rojas J-L, Marchand C (2013) Compression of redundancy free trellis stages in Turbo-decoder. Electron Lett 49(7):460–462
5. CAS 5093 40 Mb/s Turbo code decoder. December Rev 4.1. Comatlas. (May 1995)

Chapter 2
Challenges and Limitations for Very High Throughput Decoder Architectures for Soft-Decoding

Norbert Wehn, Stefan Scholl, Philipp Schläfer, Timo Lehnigk-Emden, and Matthias Alles

2.1 Motivation

In modern communications systems the required data rates are continuously increasing. Especially consumer electronic applications like video on demand, IP-TV, or video chat require large amounts of bandwidth. Already today's applications require throughputs in the order of Gigabits per second and very short latency. Current mobile communications systems achieve 1 Gbit/s (LTE [1]) and wired transmission enables even higher data rates of 10 Gbit/s (e.g., Thunderbolt [2], Infiniband [3]) up to 100 Gbit/s. For the future it is clearly expected that even higher data rates become necessary. Early results show throughputs in the order of 100 Tbit/s [4] for optical fiber transmissions.

Satisfying these high date rates poses big challenges for channel coding systems. Software solutions usually achieve only very small data rates, far away from the required speed of most applications. Therefore dedicated hardware implementations on ASIC and FPGA are mandatory to meet the requirements for high speed signal processing. To achieve speeds of Gigabits per second, these architectures need large degrees of parallelism.

Parallelism and speed can easily be increased by running several single decoders in parallel. This is however mostly an inefficient solution, because area and power increase linearly with parallelism. Moreover it implies a large latency.

N. Wehn (✉) • S. Scholl • P. Schläfer
Microelectronic System Design Research Group, University of Kaiserslautern, Kaiserslautern, Germany
e-mail: wehn@eit.uni-kl.de; scholl@eit.uni-kl.de; schlaefer@eit.uni-kl.de

T. Lehnigk-Emden • M. Alles
Creonic GmbH, Kaiserslautern, Germany
e-mail: info@creonic.com

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_2

Thus it is more advantageous to investigate efficient architectures specialized to high throughput. This may also include modifications to the decoding algorithm itself.

An important metric for analyzing high throughput architectures is area efficiency. Area efficiency is defined as throughput per chip area ($[(\text{Gbit/s})/\text{mm}^2]$). The area efficiency can be increased significantly by new architectural approaches.

We present high throughput decoders for different application relevant coding schemes, such as Reed–Solomon, LDPC and Turbo codes and point out their benefits compared to state-of-the-art architectures.

2.2 Architectures for Soft Decision Reed–Solomon Decoders

2.2.1 Introduction

Reed–Solomon (RS) codes are utilized in many applications and communication standards, either as a stand-alone code or in concatenation with convolutional codes, e.g., DVB. They are traditionally decoded using hard decision decoding (HDD), using, e.g., the well-known Berlekamp–Massey algorithm. However, using also the probabilistic information—so-called soft information—on the received bits can lead to large improvement of frame error rate (FER) in comparison to HDD.

Numerous algorithms have been proposed for soft decision decoding of RS codes. They are using different approaches to achieve a gain in FER over HDD with different complexities. A selection of interesting algorithms can be found in [5–11]. This chapter will focus on the RS(255,239) and RS(63,55) codes, because they are widely used in many applications.

Up to now, only few hardware implementations for ASIC and FPGA have been proposed for soft decoding of RS codes, especially the RS(255,239). One trend becoming apparent are implementations based on Chase decoding [7] and the closely related low-complexity chase (LCC) algorithm [8]. Hardware implementations based on LCC exhibit low hardware complexity [12–14], but this low complexity comes at the expense of a poor FER gain. Implementations based on LCC provide only little FER gain over HDD of about 0.3–0.4 dB.

The design of hardware architectures for a larger gain in FER is more challenging. Architectures and implementations based on adaptive belief propagation and stochastic Chase decoding exhibit a larger FER gain (0.75 dB), but having a low throughput [15, 16].

In this chapter a third approach for soft decoding is described that enables a large gain in FER *and* high throughput. It is based on a variant of the information set decoding algorithm, for which an efficient architecture is presented. This architecture shows a uncompromised gain in FER of 0.75 dB and a high throughput that exceeds 1 Gbit/s on a Xilinx Virtex 7 FPGA [17].

2.2.2 *Information Set Decoding*

This section introduces the algorithm, which is the basis of the high throughput hardware architecture. First, the used variant of information set decoding called ordered statistics decoding (OSD) algorithm [10] is reviewed. Then, a reduced complexity version of OSD using the syndrome weight [18] is presented.

2.2.2.1 Original OSD

OSD has been proposed in [10] and belongs to the class of information set decoders [19].

Basically information set decoding works as follows: First, divide the N received bits $\bar{\mathbf{y}}$ into two groups according to their reliability. The bit reliability is determined by the absolute value of the corresponding log likelihood ratio (LLR). The first group contains the set of K reliable bits, called the *information set*. The second group contains the $M = N - K$ unreliable bits, also referred to as low reliable bit positions (LRPs).

Before actual decoding starts the M LRPs are erased. Then the information set is used to reconstruct the M erased bits using the M parity checks in the parity check matrix $\mathbf{H}$. To do so, $\mathbf{H}$ has to be put in a diagonalized form $\hat{\mathbf{H}}$ via Gaussian elimination. If the K bits of the information set are correct, all errors in the M LRPs can be corrected. This is referred to as order-0 reprocessing in OSD or OSD(0).

To perform successful correction in the case of one error in the information set, the reconstruction process is repeated several times, each time with exactly one bit of the information set flipped. This results in a list of $K + 1$ possible codewords from which the best codeword is selected by evaluating the Euclidean distance to the received LLRs. This improved decoding is called order-1 reprocessing or OSD(1).

A key part for understanding information set decoding is the reconstruction process. To successfully reconstruct the M erased bits, the rows of the parity check matrix are used, as mentioned before. It is required that the parity check equation in each row covers mostly one of the erased M bits. To fulfill this requirement, $\mathbf{H}$ is put into a diagonalized form by Gaussian elimination, such that each row covers not more than one erased bit. Note that the Gaussian elimination does not change the channel code itself, because an RS code is a linear code.

2.2.2.2 Reduced Complexity Algorithm for Hardware

The computational bottleneck of the original algorithm are the reconstructions of the M erased bits. For example, in case of decoding an RS(255,239) with OSD(1) this operation is required 2,041 times. To overcome this problem a reduced complexity algorithm is utilized which makes use of the syndrome $\hat{\mathbf{s}}$ and its weight [18] that enables fast and efficient reconstruction.

The reduced complexity algorithm starts (as the original OSD) with determining the information set according to the bit reliabilities and diagonalization of $\mathbf{H}$ by Gaussian elimination.

Original OSD then evaluates the parity check equations given by the rows of $\hat{\mathbf{H}}$ whereas the considered low-complexity algorithm merely uses the syndrome to correct the corrupted bits.

Moreover, if the syndrome vector is calculated using the diagonalized parity check matrix, i.e., $\hat{\mathbf{s}} = \hat{\mathbf{H}}\bar{\mathbf{y}}^{\mathbf{T}}$, two distinct cases for the binary weight of the syndrome vector can be observed:

- The syndrome weight is small: In this case, it is assumed that only errors in the M bits are present, i.e., OSD(0) processing is sufficient.
- The syndrome weight is large: In this case, it is assumed that also errors in the information set are also present. Then OSD(1) processing is performed.

A fixed weight threshold to decide between the two cases is denoted by $\Theta \in \mathbb{N}$ and determined by simulation.

OSD(0) (small syndrome weight) is performed by simply flipping the M LRPs that have led to the 1s in the syndrome vector. Conducting OSD(1) (large syndrome weight) to correct one error inside the information set is done by first flipping the bit position

$$j = \underset{i=0,\ldots,N-1}{argmin} \; wgt\left(\hat{\mathbf{s}} \oplus \hat{\mathbf{h}}_{\mathbf{i}}\right)$$

where $\hat{\mathbf{h}}_{\mathbf{i}}$ denotes the ith column of $\hat{\mathbf{H}}$. After flipping the error inside the information set at position j, the syndrome is calculated again and the remaining errors outside the information set (i.e., among the LRPs) are corrected by performing OSD(0).

Note that this algorithm inherently determines the best codewords among the possible candidates only by looking at the syndrome weight. It is sufficient to select the candidate with smallest syndrome weight. In case of original OSD the Euclidean distance between candidate and received LLRs had to be evaluated many times.

For more detailed information on the syndrome weight OSD please refer to [18] or [17].

2.2.2.3 HDD Aided Decoding

One disadvantage of OSD over other soft decision decoding algorithms is the tendency for a weak FER performance if SNR increases. To improve FER OSD is extended with a conventional HDD, whose result is output if OSD fails.

A failure in OSD can be easily detected again by looking at the syndrome weight. If after OSD(1) reprocessing the updated syndrome still has a large weight, OSD can be considered as unsuccessful.

1. ***Sorting:***
   determine the information set of K reliable bits
2. ***Gaussian Elimination:***
   diagonalize $\mathbf{H}$ *at the* $N-K$ *low reliable positions to obtain* $\hat{\mathbf{H}}$
3. ***calculate the syndrome*** $\hat{\mathbf{s}} = \hat{\mathbf{H}}\bar{\mathbf{y}}^{\mathrm{T}}$
   and its binary weight $wgt\,(\hat{\mathbf{s}})$
4. **If** $wgt\,(\hat{\mathbf{s}}) > \Theta$: /* errors in the information set */
   flip the received bit at position
   $j = \underset{i=0,\ldots,N-1}{argmin}\ wgt\left(\hat{\mathbf{s}} \oplus \hat{\mathbf{h}}_{\mathbf{i}}\right)$
   update the syndrome $\hat{\mathbf{s}} = \hat{\mathbf{s}} \oplus \hat{\mathbf{h}}_{\mathbf{j}}$ *and* $wgt\,(\hat{\mathbf{s}})$, goto 5
5. **If** $wgt\,(\hat{\mathbf{s}}) \leq \Theta$: /* only errors outside the information set remaining (LRP errors) */
   For all $\hat{s}_i = 1$, *flip bit position l, for which* $\hat{h}_{il} = 1$
   output OSD result, terminate
   else
   perform HDD on $\bar{\mathbf{y}}$ *and output HDD result,* terminate

Fig. 2.1 Reduced complexity OSD(1) based on the syndrome weight that is implemented

2.2.2.4 Implemented OSD Version

The reduced complexity OSD algorithm for hardware implementation is summarized in Fig. 2.1. It features sorting to determine the information set, followed by Gaussian elimination to diagonalize the matrix. Then the syndrome weight for the diagonalized matrix is determined and a decoding strategy (OSD(1)+OSD(0) or OSD(0) only) is selected. HDD is performed only if OSD fails.

2.2.3 Hardware Architecture

In this section a hardware architecture based on the previously introduced algorithm (Fig. 2.1) is presented.

2.2.3.1 Architecture Overview

Figure 2.2 shows the overall hardware architecture. P LLRs are fed in parallel into the decoder and stored in the "I/O bit memory." During data input the received LLRs are sorted using a parallelized sorter. The M bit positions outside the information set are stored in the "LRP memory." Simultaneously, the syndrome is calculated based on the original (non-eliminated) parity check matrix $\mathbf{H}$ for reasons that will be explained below. Also HDD is carried out, whose result is stored in the "HDD memory."

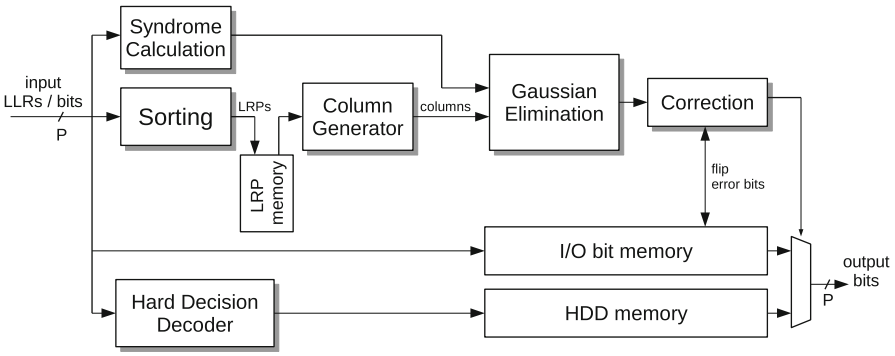

Fig. 2.2 Decoder architecture overview

After that, the column generator generated the columns of $\mathbf{H}$ corresponding to the M LRPs for Gaussian elimination. These LRP columns are fed into the Gaussian Elimination Unit to dynamically set up the unit (see below) for further processing. After having set up the Gaussian Elimination Unit, the syndrome $\mathbf{s}$ is put into the elimination unit to obtain $\hat{\mathbf{s}}$ (its version based on the diagonalized matrix).

After determining the initial syndrome $\hat{\mathbf{s}}$, the Correction Unit calculates the syndrome weight and determines the decoding strategy (OSD(0) or OSD(1)). If OSD(1) is executed, the Column Generator outputs successively every column of $\mathbf{H}$, which are transformed into the columns of $\hat{\mathbf{H}}$ by the Gaussian Elimination Unit. The Correction Unit determines the erroneous bit positions based on these columns and the syndrome and flips these bits in the "I/O bits memory." Finally, the Correction Unit decides if the best OSD codeword or the HDD codeword is output.

2.2.3.2 Sorting Unit

The first step of decoding is finding the LRPs by taking the absolute value of the received LLRs and partially sorting them. This is accomplished by the Sorting Unit depicted in Fig. 2.3. Sorting is performed by using a shift register based insertion sort.

In order to reduce the latency for sorting, the shift register is partitioned in P parts. Each part is calculated in parallel. However, the results provided by the parallelized Sorting Unit are not exactly the LRPs, but rather an approximation of the LRPs. This introduces a small loss in FER, but simulations show that this loss is less than 0.1 dB for the RS(255,239) using an input parallelism of $P = 8$.

Finally, the LRPs are read out of the shift register and stored in the "LRP memory" for further processing.

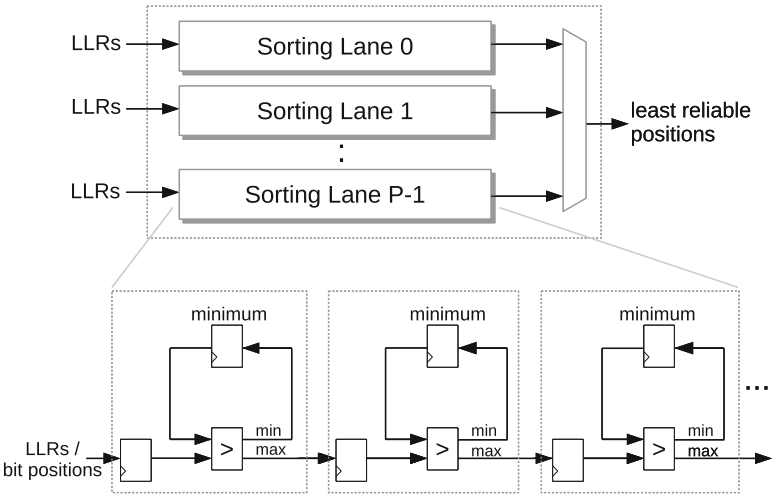

Fig. 2.3 Parallelized sorting unit

2.2.3.3 Syndrome Calculation Unit

The subsequent stages require the calculation of the syndrome using the diagonalized matrix $\hat{\mathbf{s}} = \hat{\mathbf{H}}\bar{\mathbf{y}}^{\mathbf{T}}$. However, it is advantageous to first calculate the syndrome using the original parity check matrix $\mathbf{H}$ and afterwards pass it through the Gaussian Elimination Unit to obtain $\hat{\mathbf{H}}$.

This allows to use efficient syndrome calculation using Galois field arithmetic, as it is well known in literature [20]. The syndrome unit is a parallelized implementation that can handle one received symbol (P bits) per clock cycle.

2.2.3.4 Column Generator Unit

The column generator consists of a ROM, which holds the original parity check matrix $\mathbf{H}$. The Column Generator receives a column number at its input and outputs the requested column of $\mathbf{H}$.

2.2.3.5 Gaussian Elimination Unit

Gaussian Elimination is required to diagonalize $\mathbf{H}$ and the syndrome $\mathbf{s}$. This is the most complex operation in the algorithm. Therefore sophisticated architectures are required to achieve high throughput.

An elegant architecture for Gaussian elimination has been proposed in [21]. This architecture consists of a pipelined array, which eliminates the columns on the fly. The columns of the original matrix $\mathbf{H}$ are input from the left and the corresponding

columns of the eliminated matrix $\hat{\mathbf{H}}$ are output at the right. Each of the M column eliminators is responsible for carrying out the operations needed to eliminate exactly one of the M columns corresponding to the LRPs.

The array works in two phases:

1. The Setup Phase: The M columns, which are supposed to become unit vectors after elimination, (here: the LRPs) are passed into the array to dynamically set up the structure for row adding in the column eliminators.
2. The Elimination Phase: Columns of the original matrix are passed into the array. The columns of the eliminated matrix are output after M clock cycles.

Note that linear independency of LRPs is required for full elimination. Possible dependencies are inherently checked during the setup phase. If an LRP column turns out to be dependent on some other LRP column, it is simply discarded, so that the matrix is not fully diagonalized. Since the number of dependent columns is usually very low the resulting loss in correction performance is negligible.

This two-phase architecture has proven to be an efficient solution for this application and outperforms standard Gaussian elimination architectures (e.g., systolic arrays) as can be seen in Table 2.1. For more information on the functionality and the architecture of the utilized Gaussian elimination please refer to [21] (Fig. 2.4).

2.2.3.6 Correction Unit

The unit first determines if OSD(0) or OSD(1) has to be performed. To determine the decoding strategy and the erroneous bit positions the syndrome weight has to

Table 2.1 Comparison of state-of-the-art implementations for Gaussian elimination of the binary 128×2040 matrix for the RS(255,239) code on a Xilinx Virtex 7 FPGA using Vivado 2012

Architecture	LUTs	FFs	f_{max} (MHz)	Throughput (matrices/s)
SMITH (estimated) [22]	780k	260k	–	–
Systolic array [23]	81.7k	98.6k	350	145k
Proposed [21]	**16.6k**	**32.9k**	**370**	**171k**

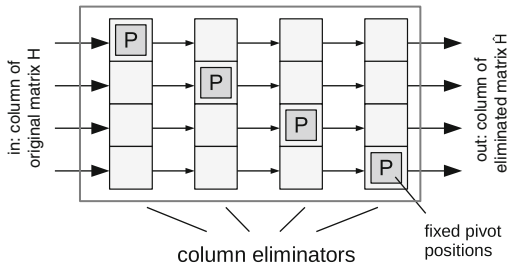

Fig. 2.4 Array for Gaussian elimination (here with $M = 4$)

be calculated. The weight calculation of binary vectors is accomplished by an adder tree consisting of P stages. Several pipeline stages have been added between the adder stages to reduce the critical path.

However, the main task of this unit is to perform the actual error correction. To determine erroneous bit positions, the syndrome and its correlation to the columns of $\hat{\mathbf{H}}$ are evaluated according to Steps 4 and 5 of the decoding algorithm (Fig. 2.1). In case of an error, the corrupted bits are flipped in the "I/O bits memory."

2.2.3.7 Hard Decision Decoder

To improve the FER, an additional HDD is employed. For the FPGA implementation a HDD IP core for decoding RS codes from Xilinx [24] is used. It supports the considered codes and provides the necessary throughput for the decoder architecture.

2.2.3.8 Fixed Point Quantization Issues

Since soft information (LLRs) is only processed in the Sorting Unit, quantization of the LLR values affects only this small part of the decoder. By simulations it is determined that an LLR quantization of 7 bits for the RS(255,239) and 5 bits for the RS(63,55) code does not noticeably impact the FER performance.

2.2.3.9 Pipelining and Latency Issues

In the proposed decoding architecture, performing OSD(0) and OSD(1) has a latency of 795 and 2,838 clock cycles, respectively (RS(255,239)). This shows that OSD(1) is much more costly than OSD(0). In conjunction with the thresholding of the syndrome weight (see Sect. 2.2.2.4), decoding throughput can be increased largely, if OSD(1) is only performed, if it is actually needed. This leads to a large improvement of throughput, especially for SNR values of practical interest (typical throughput).

Moreover, a two-stage pipelining is used:

- Stage 1: LLR input, sorting, syndrome calc., HDD
- Stage 2: Gaussian elimination and error correction

2.2.4 Implementation Results

In this section, implementation results for the RS(255,239) and the RS(63,55) codes based on the new architecture are presented. FPGA implementations have been done on a Virtex 7 (xc7vx690t-3) device using Xilinx ISE 14.4. All results shown have been obtained after place & route.

Table 2.2 Implementation results for the new architecture for the RS(255,239) and the RS(63,55) on a Virtex 7 FPGA

	RS(255,239)	RS(63,55)
LUTs	15.9k	3100
FFs	41.7k	7480
BRAMs (36K/18K)	7/8	1/5
f_{max}	280 MHz	300 MHz
Worst case throughput	200 Mbit/s	170 Mbit/s
Typical throughput	1,190 Mbit/s	690 Mbit/s
Gain	0.75 dB	1.4 dB

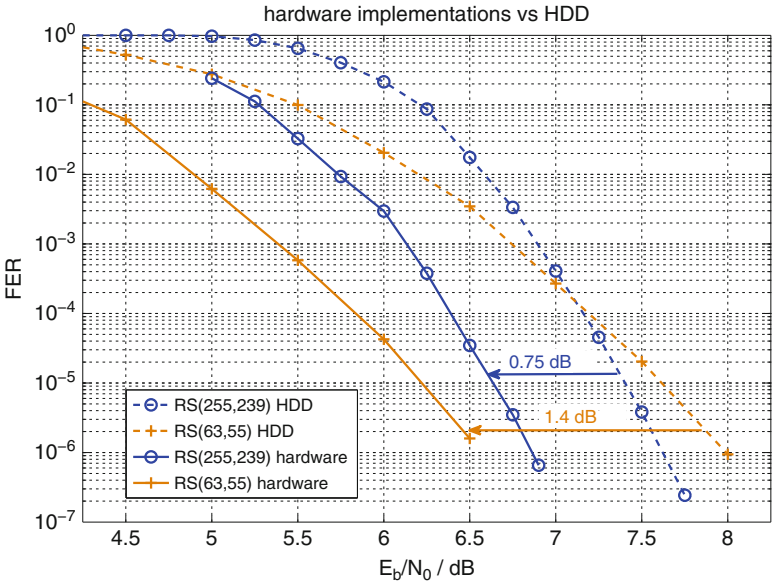

Fig. 2.5 FER for the hardware implementations

Implementation results for the RS(255,239) and for the RS(63,55) can be found in Table 2.2. For typical throughput calculations FER$= 10^{-4}$ is considered. The communication performance of the proposed decoder is shown in Fig. 2.5. For the RS(255,239) a gain of 0.75 dB and for the RS(63,55) a gain of 1.4 dB are achieved.

A comparison with other state-of-the-art FPGA soft decoders for RS codes is presented in Table 2.3. Since the other decoders rely on older FPGAs, results are given for Virtex 5, which provides a more fair comparison between the different implementations.

In terms of FER gain, the new architecture is comparable to other state-of-the-art implementations (gain of 0.7–0.75 dB). However the decoder achieves this gain in FER with considerably higher throughput and significantly less resource utilization.

Table 2.3 Comparison with other soft decoder implementations for RS(255,239) on FPGA

Implementation (Algorithm)	FPGA	LUTs	FFs	Throughput (Mbit/s)	Gain over HDD (FER= 10^{-4})
[15] (ABP)	Stratix II	43.7k	n/a	4	0.75 dB
[16] (Chase)	Virtex 5	117k	143k	50	0.7 dB
New proposed (information set)	**Virtex 5**	**13.7k**	**41.8k**	**805**	**0.75 dB**

The implementation shows that information set decoding is a viable way to implement soft decoders for RS codes efficiently, also for large throughput requirements.

2.3 Architectures for Turbo Code Decoders

Turbo codes are widely used for error correction in mobile communications, e.g., in UMTS and LTE systems. Similar as in other areas, their throughput demand is increasing, currently reaching beyond 1 Gbit/s for LTE.

Turbo code decoding is inherently serial on component and on the decoder level. A turbo code decoder consists of two component decoders which iteratively exchange data on block level. A complete data block of length B is fully processed by one component decoder before the processed block can be sent to the other component decoder. This process continues for a certain amount of iterations, typically between 5 and 8. However, the data is not directly sent to the other component decoders. Instead the data is interleaved before exchanging. This interleaving is pseudo-random with limited locality and can result in access conflicts if several messages are produced in one clock cycle by a component decoder. Resolving these access conflicts imposes constraints on an interleaver to permit a parallelized data exchange.

The component decoders are soft-in, soft-out decoders. State-of-the-art turbo code decoders are using the Bahl, Cocke, Jelinek, and Raviv algorithm, often also called Maximum-a-posteriori algorithm (MAP) [25]. This algorithm sequentially processes a block from the beginning to the end, named forward recursion, and vice versa, called backward recursion [26]. Due to the recursive nature of the forward and backward calculations, the standard MAP algorithm cannot straightforward be parallelized. Thus, we are facing two challenges for high throughput turbo code decoders:

- parallelizing the MAP algorithm and
- parallelizing the data exchange

An overview of different levels of parallelisms for turbo code decoders can be found in [27]. Parallelizing the MAP algorithm can be achieved by splitting the

complete data block into P smaller sub-blocks. So, the sub-block size is B/P. Since this splitting breaks up the recursion for forward and backward calculation, it will result in a large degradation of the communications performance. To counterbalance this effect, a so-called acquisition can be performed at the sub-block borders for the forward and/or backwards recursion. This acquisition consists of some additional recursions steps and approximates the state probability at the borders of the sub-blocks. The accuracy of this approximation strongly depends on the number of additional recursion steps. The number of additional steps is named acquisition length L_{ACQ}. In this way each sub-block can be processed independently of the other sub-block on a dedicated MAP engine. That is, in this case a component decoder consists of P parallel working MAP engines where each MAP engine serially processes a sub-block. So, instead of B clock cycles needed to process one data block of size B (here we assume that one recursion step is performed in one clock cycle), we need only B/P clock cycles. State-of-the-art MAP decoders use the same splitting technique to reduce the storage amount when processing this sub-block inside a MAP engine. This technique is called sliding windowing [28]. We use the term window instead of sub-block to avoid a confusion with the splitting on block level. The window length is denoted L_{WL}.

Since one component decoder is always idle while waiting for the result of the other decoder, we can map both component decoders on the same hardware unit without degrading the throughput. If we assume that there is no throughput penalty due to the data exchange between the two component decoders, we can calculate the throughput TP of a state-of-the-art turbo code decoder with Eq. (2.1).

$$TP = \frac{B}{(B/P + L_{MAP}) \cdot n_{half\_iter}} \cdot log_2(r) \cdot f[Mbit/s], \tag{2.1}$$

with f being the frequency and n_{half_iter} the number of invocations of a component decoder. Please note that a component decoder is invoked twotimes per decoding iteration. r is the used radix. Radix-2 means that the MAP engine processes one recursion step per clock cycle. In radix-4, two recursion steps are merged into a single one which can be processed in one clock cycle. This merging increases the area and slightly decreases the frequency, but reduces the number of clock cycles for processing a sub-block by a factor 2. L_{MAP} is the overhead due to the parallel processing of a data block and is composed of three components:

- $L_{pipeline}$: a MAP engine is pipelined to increase the frequency. This implies a certain latency which is typically 10–20 clock cycles.
- L_{ACQ} is the aforementioned acquisition length.
- L_{WL} is the window length of the sliding windowing.

From this equation we see a linear increase in the throughput with increasing parallel processing as long as L_{MAP} can be neglected. We see also that high throughput decoders require a large P. However a large P increases the impact of L_{MAP} on TP. Moreover current communication standards like LTE specify high throughputs only for high code rates ($R = 0.95$). It can be shown that large code

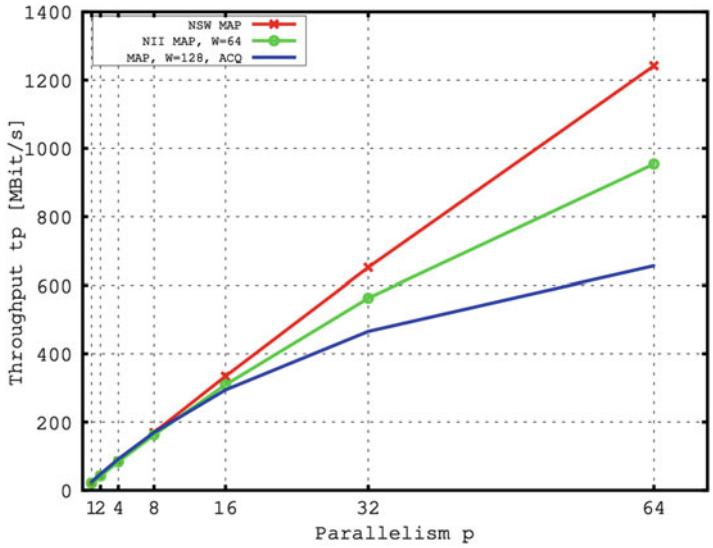

Fig. 2.6 Throughput for different MAP architectures: Non-sliding window MAP with NII and 6 iterations, sliding window MAP with NII and $l_{WS} = 64$ and 6 iterations, sliding window MAP with $l_{ACQ} = l_{WS} = 128$ and 6 iterations

rates demand large L_{ACQ} for a good communications performance which further exacerbates the dominance of L_{MAP}, e.g. in LTE $L_{ACQ} > 64$ is mandatory. In other words, the throughput starts to saturate with large P and high code rate demands, as shown in Fig. 2.6. This fact poses a further challenge for turbo decoder architectures when high throughput is required. However we can use two techniques to relax this problem:

- We can reduce the large acquisition length by exploiting the state probabilities at the borders of the sub-block from the previous decoding iteration. This technique is called next iteration initialization and largely helps to reduce L_{ACQ} [29]. Sometimes it is even possible to perform no acquisition at all by using only the information of the previous iteration.
- We quit the sliding window technique inside a MAP engine. This normally largely increases the memory and energy consumption inside the engine. However this can be avoided by so-called re-computation [30]. Here, instead of storing every metric of a forward recursion step, we store only every nth metric and re-calculate the other $n-1$ metrics. That is, we trade-off storage versus additional computations. It can be proven that the optimum n is $\sqrt{B/2P}$. For example, this technique reduces the number of metrics to be stored for LTE from 6,144 for the largest block size to 768.

So far we assumed that interleaving implies no additional latency. Obviously a component decoder produces P data per clock cycle. These data have to be

Table 2.4 Previously published high throughput 3GPP TC decoders

	This work	[33]	[34]	[35]
Radix and Parallelism	4/32	2/64	2/8	4/8
Throughput (MBit/s)	2,300	1,200	150	390
@Clock f (MHz)	500	400	300	302
Parameters affecting communications performance				
Iterations	6	6	6.5	5.5
Acquisition	NII	NII	96 + NII	30
Window length	192	64	32	30
Input quantization	6	6	6	5
Technology (nm)	65	65	65	130
Voltage	1.1 V	0.9 V	1.1 V	1.2 V
Area (mm^2)	7.7	8.3	2.1	3.57

concurrently interleaved. Since each MAP engine has its own memory, random interleavers can result in access conflicts if we have single- or dual-port memories. That is, several data have to be written simultaneously into the same memory. Such conflicts have to be resolved by serialization. Another possibility is to design the interleaver a-priori in a way such that these conflicts are avoided. Current communication standards like LTE are based on such interleavers and show no access conflicts for up to 64 simultaneous produced data. On the other side HSDPA has no conflict free interleavers due to its downward compatibility with UMTS. Parallel MAP processing was not yet an issue at the time when UMTS was defined. Sophisticated techniques exist for run-time conflicts resolution, but this discussion is not in the scope of this chapter [31]. The influence of the explained techniques on the achievable throughput is shown in Fig. 2.6.

In [32] an LTE compatible turbo code decoder was presented which used all the aforementioned techniques. It achieves a throughput of 2.15 Gbit/s on a 65 nm CMOS bulk technology under worst case PVT parameters. It uses 32 MAP engines with radix-4, next iteration initialization and no sliding window but re-computation. The detailed results and comparison with state-of-the-art decoders are shown in the Table 2.4.

2.4 High Throughput Architectures for Low Density Parity Check Decoders

As discussed in Sect. 2.3, turbo code based systems cannot provide data rates in the order of several hundred Gigabits per second. For these applications LDPC codes are the best choice. The decoding algorithms for LDPC codes have an inherent parallelism which can be exploited by highly parallel architectures.

LDPC codes have been introduced by Gallager in 1962 [36] but the high decoding complexity made the application at that time impossible. When LDPC codes have been rediscovered in the late 1990s, the throughput demands have been moderate. Serial decoder architectures have been sufficient to fulfill the requirements. As demands on the throughput rose, partially parallel architectures became necessary. Today LDPC codes are used in a wide range of applications like 10 Gbit Ethernet (10 GBASE-T, IEEE802.3an) [37], broadband wireless communication (UWB, WiGig) [38, 39], and storage in hard disc drives [40]. State-of-the-art LDPC decoders can already process data rates in the range of 10–50 Gbit/s. This is sufficient to satisfy the requirements of all mentioned standards. However, as future standards emerge, current architectures will not be able to facilitate the demanded throughputs of 100 Gbit/s and more. For higher throughputs even LDPC decoders reach their limit. This results in a gap in decoder performance which has to be closed by new approaches. Therefore a new architecture is presented which can overcome these limitations and the key aspects for next generation LDPC decoders are discussed. It is shown that new architectures significantly reduce routing congestion which poses a big problem for high speed LDPC decoders. The presented 65 nm ASIC implementation results underline the achievable gain in throughput and area efficiency in comparison to state-of-the-art architectures.

A LDPC decoder system with state-of-the-art communications performance and a throughput far beyond 100 Gbit/s is presented which is a candidate for future communications systems.

2.4.1 LDPC Decoding

LDPC codes [36] are linear block codes defined by a sparse parity check matrix $\mathbf{H}$ of dimension $M \times N$, see Fig. 2.7a. A valid codeword $\mathbf{x}$ has to satisfy $\mathbf{Hx^T} = \mathbf{0}$ in modulo-2 arithmetic. A descriptive graphical representation of the whole code is given by a Tanner graph. Each row of the parity check matrix is represented by a check node (CN) and corresponds to one of the M parity checks. Respectively each column corresponds to a variable node (VN) representing one of the N code bits. The Tanner graph shown in Fig. 2.7b is the alternative representation for the parity check matrix of Fig. 2.7a. Edges in the Tanner graph reflect the 1's in the $\mathbf{H}$ matrix. There is an edge between VN n and CN m if and only if $\mathbf{H}_{mn} = 1$. LDPC codes can be decoded by the use of different algorithms. Belief Propagation (BP) is a group

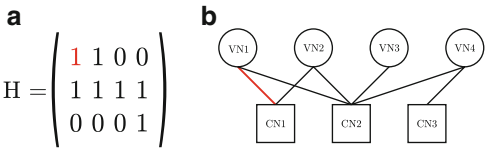

Fig. 2.7 **H** Matrix and Tanner graph hardware mapping

of algorithms which is used in most state-of-the-art decoders. Which type of BP fits best has to be chosen dependent on the required communications performance. For example, the λ-min algorithm [41] performs better than the min-sum algorithm [42] but has a significantly higher implementation complexity. All algorithms have in common that probabilistic messages are iteratively exchanged between variable and check nodes until either a valid codeword is found or a maximum number of iterations is exceeded.

2.4.2 LDPC Decoder Design Space

The LDPC decoder design space comprises a multitude of parameters which have to be tuned to the specific requirements. Each standard has different needs in means of error correction performance, number of supported code rates, codeword lengths, and throughput. There are numerous design decisions which have to be made for the hardware to satisfy these requirements. Due to their inherent parallelism, LDPC decoders are of special interest for high throughput applications. Therefore the focus is on the design decisions concerning the decoder parallelism. They have the strongest impact on the system's throughput. An in-depth investigation of the design space for slower state-of-the-art partially parallel decoders is presented in [43]. This part of the design space is not highlighted as it is orthogonal to the presented schemes. For example, different check node algorithms can be combined with all levels or parallelism presented here.

There are multiple dimensions in which the degree of parallelism can be chosen, see Fig. 2.8. The lowest level of parallelism is on the message level. Each message

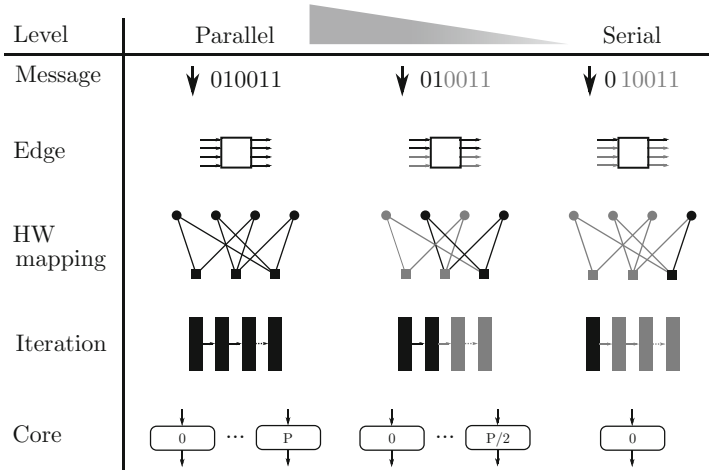

Fig. 2.8 Levels of parallelism in the high throughput LDPC decoder design space

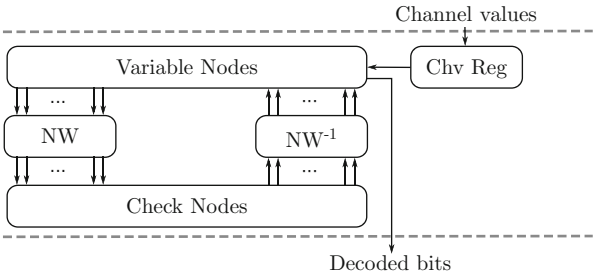

Fig. 2.9 Fully parallel hardware mapping of an LDPC decoder. All variable and check nodes are instantiated and two networks are required to exchange the messages between them. Massive routing congestion is observed for this architecture

can be transmitted in tuples of bits or fully parallel. Today fully parallel message transfer can be found in the vast majority of architectures as it has been shown to be more efficient. The second degree of parallelism is represented by the number of parallel edges. The node's in- and outgoing edges can be processed one after another, partially parallel or fully parallel. However the choice of the node's edge parallelism is directly linked to the so-called hardware mapping. The hardware mapping describes how many check and variable nodes are instantiated. When talking of a fully parallel decoder, an architecture instantiating all processing nodes is meant. In contrast partially parallel decoders have a lower number of physical nodes than the Tanner graph. They process the parity check matrix in a time multiplex fashion. This allows for easy adaption of the architecture to new standards but limits the achievable throughput.

For applications like 10 GBASE-T Ethernet only fully parallel architectures can achieve the required throughput. Figure 2.9 depicts the high-level structure of such a decoder. However in general it is not advisable to build this architecture as it has a serious drawback which is directly related with the two networks between VNs and CNs. Dependent on the code length and quantization, each of them comprises between several thousands and hundred thousands of wires which have to be routed according to the parity check matrix. To achieve a good communications performance, parity check matrices have long cycles and thus no locality, resulting in massive routing congestion. It has been shown in earlier publications [44,45] that the area utilization is heavily impaired by this fact and only 50 % of the chip is used by logic.

Fully parallel decoders can still satisfy today's requirements in means of throughput. They represent the highest level of parallelism used in state-of-the-art decoder designs. However, for future standards the throughput demands will further increase and cannot be achieved using the presented dimensions of parallelism.

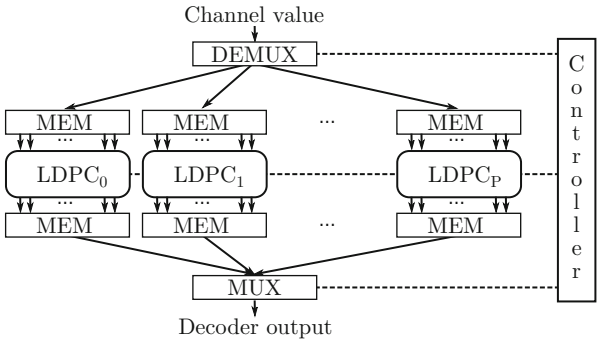

Fig. 2.10 Core duplication architecture

2.4.3 Exploring a New Dimension in the High Throughput LDPC Decoder Design Space

Considering the degrees of parallelism which are used for state-of-the-art decoders no further increase in throughput can be acquired. In the following section two more degrees in parallelism are discussed which can be explored to overcome the limitations in LDPC decoder throughput, see Fig. 2.8. Moreover it is shown that the area efficiency of decoders can even be increased by the proposed techniques.

2.4.3.1 Core Duplication

One solution to achieve the throughputs required by future standards is to instantiate several LDPC decoder cores in parallel, see Fig. 2.10. There are two possible starting points for the core duplications. Partially parallel architectures which allow for flexibility but suffer from high latencies and low throughput. The second option is to instantiate several fully parallel decoder cores allowing for reduced latency and high throughput. However due to routing congestion they cannot achieve a satisfying area efficiency and flexibility. To connect multiple instances of a decoder, a distribution network and memories are required. Moreover a control unit to keep the blocks in order must be instantiated in the system.

Summarized straightforward decoder duplication can increase the system throughput. However the latency issues caused by partially parallel architectures cannot be solved. Potential enhancements due to the increased parallelism are not explored and the system's efficiency is slightly decreased due to the introduced overhead.

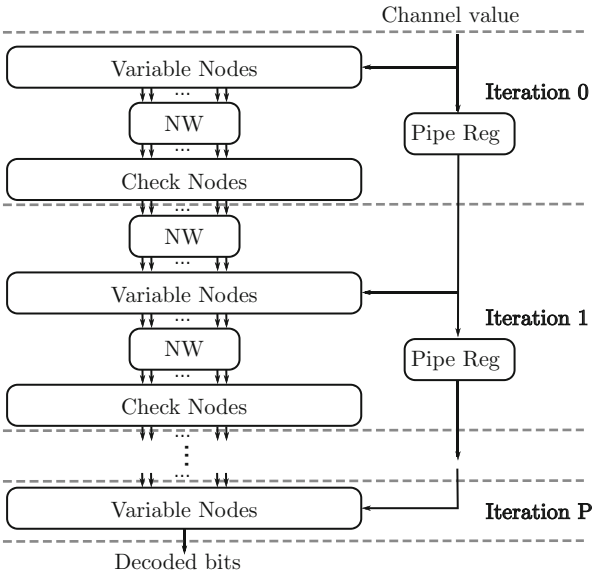

Fig. 2.11 In an unrolled LDPC decoder architecture each decoding iteration is instantiated as a dedicated hardware. A feedback from the end of the iteration back to the beginning is no more required with this approach. One of the two networks between variable and check nodes is removed and makes the routing feasible. Due to the unidirectional data flow pipelining can be applied without penalty in throughput. Synthesis results show an increased area efficiency compared to a fully parallel decoder

2.4.3.2 Unrolling Iterations

A new architecture is proposed in [46], shown in Fig. 2.11 to overcome the highlighted drawbacks. The iterative decoding loop is unrolled and an unrolled, fully parallel, pipelined LDPC decoder is instantiated. It has several advantages over core duplication and is a good choice for very high throughput architectures.

A drawback of this architecture is the need to specify the maximum number of iterations at design time. Once the decoder is instantiated there is no possibility to increase the performance by additional decoding iterations. The number of pipeline stages determines the latency, but the throughput is fixed by the cycle duration. The decoder can be considered as one big pipeline where received codewords are fed into and decoded words are returned at the end. Hence this architecture has a throughput of one codeword per clock cycle and can be pipelined as deep as required to achieve the target frequency. This allows for ultra-high throughput LDPC decoder cores.

Compared to the core duplication approach, no overhead in means of distribution networks and memory is introduced by the unrolling. Moreover there is an essential change in the resulting data flow. Where before data have iteratively been exchanged between VNs and CNs, now all data flow in one direction. Each iteration has a dedicated hardware unit and thus the decoder's overall area scales linear with the

number of decoding iterations. The result is an unidirectional wiring avoiding the overlap of opposed networks. This is a big benefit for the routing and makes the architecture more area efficient than state-of-the-art decoders which is shown in Sect. 2.4.4.

The control flow of the proposed architecture is reduced to a minimum. A valid flag is fed into the first decoding stage whenever a block is available. This flag is propagated along the decoding pipeline and enables the corresponding stages as soon as new data is available. At the same time this implies that all hardware blocks which are not used currently get clock gated.

Even though the number of decoding iterations is defined at design time, schemes like early termination can be applied to further reduce the energy consumption. Once a valid codeword is found all following decoding stages are clock gated for this block and the decoded data is bypassed in the channel value registers. By this approach besides some multiplexors no hardware overhead is introduced and the energy per decoded bit can be reduced significantly.

Even different code rates can be implemented in the unrolled architecture. Special codes like the one used in the IEEE 802.11ad standard allow for the operation of one CN instance as two CNs by cutting the node degree by two. Using these codes only minor modifications are required for the check nodes and the routing network to support all codes of the IEEE 802.11ad family. For a more detailed explanation of the CN splitting scheme see [47]. A similar scheme as proposed there can also be applied on the unrolled architecture. The control flow for the different code rates can easily be implemented by an additional flag propagated with the according input block. This allows to change the code rate for every block, e.g., in each clock cycle. Table 2.5 summarizes the benefits and drawbacks of the different approaches.

2.4.4 *Comparison of Unrolled LDPC Decoders to State-of-the-Art Architectures*

Two decoder architectures are presented which are compared to a state-of-the-art LDPC decoder from literature. The first decoder presented is a fully parallel architecture with iterative decoding. The second has also a fully parallel hardware

Table 2.5 Parallel LDPC decoder architectures

Architecture	Flexibility	Low latency	Area efficiency
Parallel inst., partially parallel architecture	+	−	0
Parallel inst., fully parallel architecture	−	0	−
Unrolled, fully parallel architecture	−	+	+

Table 2.6 State-of-the-art high throughput LDPC decoder comparison

Decoder	[48]	Iterative	Proposed unrolled
CMOS technology	65 nm	65 nm SVT	65 nm SVT
Frequency (MHz)	400	189	194
Standard	IEEE 802.3an	IEEE 802.11ad	IEEE 802.11ad
Block size	2,048	672	672
Iterations	8	6	6
Quantization (bit)	4	5	5
Post P&R area (mm^2)	5.1	1.4	7.0
Throughput (Gbit/s)	8.5	5.3	**130.6**
Area Eff. (Gbit/s/mm^2)	1.7	3.8	**18.7**

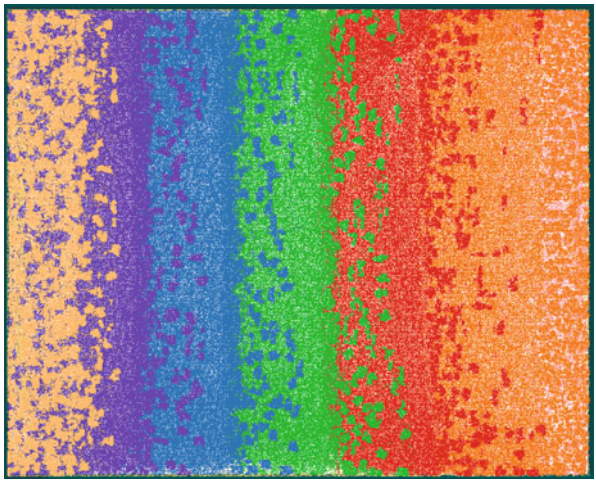

Fig. 2.12 Unrolled fully parallel LDPC decoder chip layout. Each iteration is represented by one of the vertical areas. Channel messages ripple from left to right through the decoder pipeline. All routing is very structured and pointing from one iteration to the next. A very high utilization of more than 70 % is achieved by the simplified routing

mapping but in addition the decoding iterations are completely unrolled. Both decoders support the same standard (IEEE 802.11ad) and use the same algorithm, quantization, etc.

The LDPC decoders are implemented on a 65 nm low power bulk CMOS library. The post place and route (P&R) results are summarized in Table 2.6. The physical layout of the unrolled LDPC decoder can be seen in Fig. 2.12. Comparing the synthesis results of the iterative and the unrolled decoder shows that the routing congestion is significantly reduced by the loop unrolling. The number of introduced buffers for the interconnect is significantly reduced in the unrolled decoder and leads to a high utilization of more than 70 %. A five times higher area efficiency of the unrolled decoder underlines this finding.

For the comparison, in addition to the two presented decoders, a partially parallel decoder from literature is listed. The number of iterations, quantization, and algorithm is reasonably similar to allow for a fair comparison. It can be observed that no state-of-the-art decoder architecture is capable to produce a competitive area efficiency to the unrolled architecture. The presented architecture has a throughput which is more than fifteen times higher than the one of state-of-the-art decoders. Moreover, if more throughput is required the unrolled architecture can easily be pipelined deeper to increase the core frequency. These results show the great potential of unrolled decoder architectures for future applications.

2.4.5 Future Work

The unrolled LDPC decoder architecture allows for several optimizations. In this section the most important of them are presented and current research topics are pointed out.

Unrolling the decoding loop generates a dedicated hardware instance for each iteration. While for systems working iteratively, a generic hardware fulfilling the needs of all iterations needs to be built, an unrolled architecture gives the designer the freedom to optimize the hardware for each iteration independent of the others. Thus it is possible to use specialized hardware instances for each iteration. For example, one can implement different algorithmic approximations. For example, for a decoder performing P iterations, a simplified decoding algorithm can be applied for iterations $1 \ldots i$ and an exact but more complex algorithm might be necessary only for iterations $i+1 \ldots P$. Like this a targeted communications performance can tightly be met while minimizing the required hardware resources. Even more than the area efficiency, the energy efficiency can be increased by this approach. Most blocks are decoded in the simplified first iterations of the decoding process and the higher complexity part of the decoder must not be used for them. This significantly reduces the energy per decoded bit and has almost no impact on the communications performance. Other aspects like message quantization can also be applied to this scheme and generate many new possibilities in the LDPC decoder design space. These new possibilities are currently investigated and must be considered for future architectures.

Regarding energy optimizations the proposed architecture is an excellent candidate for a Near-Threshold circuit technique [49]. For example, the throughput of 10 Gbit/s can already be fulfilled by the presented decoder running at less than 20 MHz. Thus aggressive voltage scaling to 0.5–0.6 V can be applied. This increases the energy efficiency by at least a factor of three and allows for a better energy efficiency than any other state-of-the-art decoder.

Conclusion

In this chapter, we presented soft decision Reed–Solomon, turbo, and LDPC decoder implementations with high throughput.

The introduced soft decision decoder architecture for Reed–Solomon codes is based on information set decoding. It allows a considerable improvement of error rates in combination with a high throughput. The FPGA implementation shows a throughput of beyond 1 Gbit/s and a gain of 0.75 dB over HDD for the widely used RS(255,239) code. Further research includes the evaluation of the architecture using ASIC technology and the further improvement of the correction performance.

For turbo decoding the design space has been summarized. The key techniques to a high throughput implementation have been introduced. It was demonstrated how a LTE turbo code decoder can be implemented that achieves 2.15 Gbit/s on a 65 mm ASIC technology. In the future, further investigations have to be made to ultimately increase the throughput of a turbo code by unrolling iterations.

A new LDPC decoder architecture was presented that achieves an outstanding throughput and state-of-the-art communications performance. The ASIC implementation provides a throughput of 130 Gbit/s and has a very high efficiency. Further optimizations for even higher area and energy efficiency have been discussed and will be investigated in the future.

Acknowledgements This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) within the projects "Entwicklung und Implementierung effizienter Decodieralgorithmen für lineare Blockcodes" and "Optimierung von 100 Gb/s Nahbereichs Funktransceivern unter Berücksichtigung von Grenzen für die Leistungsaufnahme."

References

1. Third Generation Partnership Project (2010) 3GPP TS 36.212 V10.0.0; 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (Release 10)
2. Intel (2014) Thunderbolt. URL http://www.thunderbolttechnology.net
3. Infiniband Association (2014). URL http://www.infinibandta.org
4. Qian D, Huang MF, Ip E, Huang YK, Shao Y, Hu J, Wang T (2011) 101.7-Tb/s (370x294-Gb/s) PDM-128QAM-OFDM transmission over 3x55-km SSMF using pilot-based phase noise mitigation. In: Optical fiber communication conference and exposition (OFC/NFOEC), 2011 and the national fiber optic engineers conference, pp 1–3
5. Wenyi J, Fossorier M (2008) Towards maximum likelihood soft decision decoding of the (255,239) Reed Solomon code. IEEE Trans Magn 44(3):423. DOI 10.1109/TMAG.2008.916381
6. Jiang J (2007) Advanced channel coding techniques using bit-level soft information. Dissertation, Texas A&M University
7. Chase D (1972) Class of algorithms for decoding block codes with channel measurement information. IEEE Trans Inf Theory 18(1):170. DOI 10.1109/TIT.1972.1054746

8. Bellorado J, Kavcic A (2006) A low-complexity method for chase-type decoding of Reed-Solomon codes. In: Proceedings of the IEEE international information theory symposium, pp 2037–2041. DOI 10.1109/ISIT.2006.261907
9. Koetter R, Vardy A (2003) Algebraic soft-decision decoding of Reed-Solomon codes. IEEE Trans Inf Theory 49(11):2809. DOI 10.1109/TIT.2003.819332
10. Fossorier MPC, Lin S (1995) Soft-decision decoding of linear block codes based on ordered statistics. IEEE Trans Inf Theory 41(5):1379. DOI 10.1109/18.412683
11. El-Khamy M, McEliece RJ (2006) Iterative algebraic soft-decision list decoding of Reed-Solomon codes. IEEE J Sel Areas Commun 24(3):481. DOI 10.1109/JSAC.2005.862399
12. An W (2010) Complete VLSI implementation of improved low complexity chase Reed-Solomon decoders. Ph.D. thesis, Massachusetts Institute of Technology
13. García-Herrero F, Valls J, Meher P (2011) High-speed RS(255, 239) decoder based on LCC decoding. Circuits Syst Signal Process 30:1643. DOI 10.1007/s00034-011-9327-4. URL http://dx.doi.org/10.1007/s00034-011-9327-4
14. Hsu CH, Lin YM, Chang HC, Lee CY (2011) A 2.56 Gb/s soft RS (255,239) decoder chip for optical communication systems. In: Proceedings of the ESSCIRC (ESSCIRC), pp 79–82. DOI 10.1109/ESSCIRC.2011.6044919
15. Kan M, Okada S, Maehara T, Oguchi K, Yokokawa T, Miyauchi T (2008) Hardware implementation of soft-decision decoding for Reed-Solomon code. In: Proceedings of the 5th international symposium on turbo codes and related topics, pp 73–77. DOI 10.1109/TURBOCODING.2008.4658675
16. Heloir R, Leroux C, Hemati S, Arzel M, Gross W (2012) Stochastic chase decoder for reed-solomon codes. In: 2012 IEEE 10th international conference on new circuits and systems (NEWCAS), pp 5–8. DOI 10.1109/NEWCAS.2012.6328942
17. Scholl S, Wehn N (2014) Hardware implementation of a Reed-Solomon soft decoder based on information set decoding. In: Proceedings of the design, automation and test in Europe (DATE '14)
18. Ahmed A, Koetter R, Shanbhag NR (2004) Performance analysis of the adaptive parity check matrix based soft-decision decoding algorithm. In: Proceedings of the conference on signals, systems and computers record of the thirty-eighth Asilomar conference, vol 2, pp 1995–1999. DOI 10.1109/ACSSC.2004.1399514
19. Dorsch B (1974) A decoding algorithm for binary block codes and$J$-ary output channels (Corresp.). IEEE Trans Inf Theory 20(3):391. DOI 10.1109/TIT.1974.1055217. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1055217
20. Lin S, Costello DJ Jr (2004) Error control coding 2nd edn. Prentice Hall PTR, Upper Saddle River
21. Scholl S, Stumm C, Wehn N (2013) Hardware implementations of Gaussian elimination over GF(2) for channel decoding algorithms. In: Proceedings of the IEEE AFRICON
22. Bogdanov A, Mertens M, Paar C, Pelzl J, Rupp A (2006) A parallel hardware architecture for fast Gaussian elimination over GF(2). In: 14th annual IEEE symposium on field-programmable custom computing machines, 2006 (FCCM '06), pp 237–248. DOI 10.1109/FCCM.2006.12
23. Kung HT, Gentleman WM (1982) Matrix triangularization by systolic arrays. Technical Report Paper 1603, Computer Science Department. URL http://repository.cmu.edu/compsci/1603
24. Xilinx LogiCORE IP Reed-Solomon Decoder (2013). http://www.xilinx.com/products/intellectual-property/DO-DI-RSD.htm
25. Bahl L, Cocke J, Jelinek F, Raviv J (1974) Optimal decoding of linear codes for minimizing symbol error rate. IEEE Trans Inf Theory IT-20:284
26. Robertson P, Villebrun E, Hoeher P (1995) A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log-domain. In: Proceedings of the 1995 international conference on communications (ICC '95), Seattle, Washington, 1995, pp 1009–1013
27. Thul MJ, Gilbert F, Vogt T, Kreiselmaier G, Wehn N (2005) A scalable system architecture for high-throughput turbo-decoders. J VLSI Signal Process Syst (Special Issue on Signal Processing for Broadband Communications) 39(1/2):63
28. Mansour MM, Shanbhag NR (2003) VLSI architectures for SISO-APP decoders. IEEE Trans Very Large Scale Integr (VLSI) Syst 11(4):627

29. Dielissen J, Huiskens J (2000) State vector reduction for initialization of sliding windows MAP. In: Proceedings of the 2nd international symposium on turbo codes & related topics, Brest, France, pp 387–390
30. Schurgers C, Engels M, Catthoor F (1999) Energy efficient data transfer and storage organization for a MAP turbo decoder module. In: Proceedings of the 1999 international symposium on low power electronics and design (ISLPED '99), San Diego, California, 1999, pp 76–81
31. Sani A, Coussy P, Chavet C (2013) A first step toward on-chip memory mapping for parallel turbo and LDPC decoders: a polynomial time mapping algorithm. IEEE Trans Signal Process 61(16):4127. DOI 10.1109/TSP.2013.2264057. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6517513
32. Ilnseher T, Kienle F, Weis C, Wehn N (2012) A 2.12Gbit/s turbo code decoder for LTE advanced base station applications. In: 2012 7th international symposium on turbo codes and iterative information processing (ISTC) (ISTC 2012), Gothenburg, Sweden, 2012
33. Sun Y, Cavallaro J (2010) Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder. Integr VLSI J. DOI 10.1016/j.vlsi.2010.07.001
34. May M, Ilnseher T, Wehn N, Raab W (2010) A 150Mbit/s 3GPP LTE turbo code decoder. In: Proceedings of the design, automation and test in Europe, 2010 (DATE '10), pp 1420–1425
35. Studer C, Benkeser C, Belfanti S, Huang Q (2011) Design and implementation of a parallel turbo-decoder ASIC for 3GPP-LTE. IEEE J Solid State Circuits 46(1):8
36. Gallager RG (1962) Low-density parity-check codes. IRE Trans Inf Theory 8(1):21
37. IEEE 802.3an-2006 (2006) Part 3: CSMA/CD Access Method and Physical Layer Specifications - Amendment: Physical Layer and Management Parameters for 10 Gb/s Operation, Type 10GBASE-T. IEEE 802.3an-2006
38. WiMedia Alliance (2009) Multiband OFDM Physical Layer Specification, Release Candidate 1.5
39. IEEE 802.11ad (2010) Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment: Enhancements for Very High Throughput in the 60 GHz Band. IEEE 802.11ad-draft
40. Kavcic A, Patapoutian A (2008) The read channel. Proc IEEE 96(11):1761. DOI 10.1109/JPROC.2008.2004310
41. Guilloud F, Boutillon E, Danger J (2003) λ-min decoding algorithm of regular and irregular LDPC codes. In: Proceedings of the 3nd international symposium on turbo codes & related topics, Brest, France, pp 451–454
42. Chen J, Dholakia A, Eleftheriou E, Fossorier MPC, Hu XY (2005) Reduced-complexity decoding of LDPC codes. IEEE Trans Commun 53(8):1288
43. Schläfer P, Alles M, Weis C, Wehn N (2012) Design space of flexible multi-gigabit LDPC decoders. VLSI Des J 2012. DOI 10.1155/2012/942893
44. Blanksby A, Howland CJ (2002) A 690-mW 1-Gb/s, rate-1/2 low-density parity-check code decoder. IEEE J Solid State Circuits 37(3):404
45. Onizawa N, Hanyu T, Gaudet V (2010) Design of high-throughput fully parallel LDPC decoders based on wire partitioning. IEEE Trans Very Large Scale Integr (VLSI) Syst 18(3):482. DOI 10.1109/TVLSI.2008.2011360
46. Schläfer P, Wehn N, Lehnigk-Emden T, Alles M (2013) A new dimension of parallelism in ultra high throughput LDPC decoding. In: IEEE workshop on signal processing systems (SIPS), Taipei, Taiwan
47. Weiner M, Nikolic B, Zhang Z (2011) LDPC decoder architecture for high-data rate personal-area networks. In: Proceedings of the IEEE international symposium on circuits and systems (ISCAS), pp 1784–1787. DOI 10.1109/ISCAS.2011.5937930
48. Zhang Z, Anantharam V, Wainwright M, Nikolic B (2010) An efficient 10GBASE-T ethernet LDPC decoder design with low error floors. IEEE J Solid State Circuits 45(4):843. DOI 10.1109/JSSC.2010.2042255
49. Calhoun B, Brooks D (2010) Can subthreshold and near-threshold circuits go mainstream? IEEE Micro 30(4):80. DOI 10.1109/MM.2010.60

Chapter 3
Implementation of Polar Decoders

Gabi Sarkis and Warren J. Gross

3.1 Introduction to Polar Codes

3.1.1 Code Construction

In [1], Arikan proved that when two bits, u_0 and u_1, are transformed as shown in Fig. 3.1a and transmitted using a binary-input, memoryless, symmetric channel, denoted W, the probability of correctly estimating one of the bits, u_0, decreases, while that of u_1 increases relative to the case where the bits are transmitted untransformed. This phenomenon is called channel polarization and it increases as the number of transformed bits, N, increases. As $N \to \infty$, the probability of correct estimation of each bit approaches either 0.5 (completely unreliable) or 1.0 (perfectly reliable), and the proportion of reliable bits is the symmetric capacity of the channel W [1]. The polarizing transformation for more than two bits is applied recursively as shown in Fig. 3.1b for $N = 4$. Later works have shown polar codes achieve the symmetric capacity of any memoryless channel [2, 3].

A polar code of length N and dimension k is constructed by placing the information bits in the k most reliable locations in the vector u_0^{N-1} and setting the remaining bits, known as the frozen bits, to predetermined values, usually 0.

3.1.2 Successive-Cancellation Decoding

Successive cancellation (SC) is the canonical algorithm for decoding polar codes and is the one used when proving their capacity-achieving performance in [1].

G. Sarkis • W.J. Gross (✉)
McGill University, Montréal, QC, Canada
e-mail: gabi.sarkis@mail.mcgill.ca; warren.gross@mcgill.ca

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_3

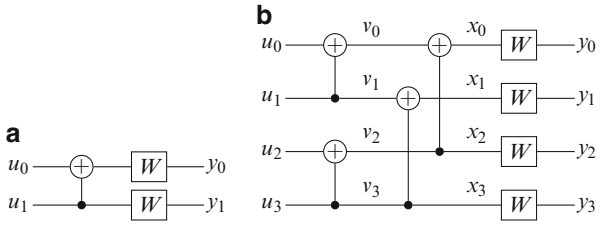

Fig. 3.1 Construction of polar codes of lengths 2 and 4, where $\oplus$ is the XOR operator. (**a**) $N = 2$; (**b**) $N = 4$

The SC algorithm estimates bits sequentially using either the predetermined values for frozen bits, or using the received channel information y and the previously estimated bits $\hat{u}_0^{i-1}$ according to the following rule for an information bit u_i

$$\hat{u}_i = \begin{cases} 0, & \text{if } \lambda_{u_i} \geq 0; \\ 1, & \text{otherwise}, \end{cases} \tag{3.1}$$

where λ_{u_i} is the log-likelihood ratio (LLR) defined as $\Pr[y, \hat{u}_0^{i-1} | \hat{u}_i = 0] / \Pr[y, \hat{u}_0^{i-1} | \hat{u}_i = 1]$ and can be calculated recursively using the min-sum (MS) approximation—with negligible impact on error-correction performance as shown via simulations in [4]—according to

$$\lambda_{u_0} = f(\lambda_{v_0}, \lambda_{v_1}) = \text{sign}(\lambda_{v_0}) \text{sign}(\lambda_{v_1}) \min(|\lambda_{v_0}|, |\lambda_{v_1}|); \tag{3.2}$$

and

$$\lambda_{u_1} = g(\lambda_{v_0}, \lambda_{v_1}, \hat{u}_0) = \begin{cases} \lambda_{v_0} + \lambda_{v_1} & \text{when } \hat{u}_0 = 0, \\ -\lambda_{v_0} + \lambda_{v_1} & \text{when } \hat{u}_0 = 1 \end{cases}. \tag{3.3}$$

λ_{v_0} and λ_{v_1} correspond to inputs and are replaced with y in the last stage of the recursive decoding.

The recursive approach to the algorithm is not suitable for a hardware implementation. Instead, the decoder uses the same equation, but proceeds from the received channel information y until an estimated bit, $\hat{u}_i$ is reached.

The major disadvantage of SC decoding is its sequential nature that leads to high decoding latency. Look-ahead techniques can be used to reduce latency as proposed in [5], where it estimates bits u_i and $u_{i+N/2}$ simultaneously with a minor increase in complexity. The latency reduction obtained by applying the look-ahead technique to a larger number of bits is limited to 50 % and results in a significant increase in complexity.

3.1.3 Belief-Propagation Decoding

In [6], it was shown that polar code can be decoded using the belief-propagation (BP) algorithm. Figure 3.1b shows that a polar code contains two constraint types: a parity-check and equality constraints corresponding to the $\oplus$ and $\bullet$ nodes, respectively. The two nodes are identical to those in BP decoding of low-density parity-check (LDPC) codes and can be decoding using the min-sum approximation as

$$\lambda_a = \lambda_b + \lambda_c \tag{3.4}$$

for the equality nodes and

$$\lambda_a = \text{sign}(b)\text{sign}(c)\min(|\lambda_b|, |\lambda_c|) \tag{3.5}$$

for the check nodes, where λ_a is the LLR corresponding to the node output, and λ_b and λ_c correspond to its inputs.

The decoder receives the channel information y and calculates messages stage by stage until a maximum number of iterations is reached or all the parity-check constraints are satisfied.

In [7], it was observed that the number of iterations required to match the error-correction performance of SC decoding was large, negating the benefits of the increased parallelism.

3.2 The Successive-Cancellation Decoder Implementation

A direct implementation of the SC decoding algorithm uses $N/2$ processing elements (PEs), which perform the functions (3.2) and (3.3) [8]. However, it was observed in [4] that the $N/2$ PEs were only used simultaneously once in the decoding, leading to low utilization of the hardware resources. That work shows that, a decoder implementing $P = 64$, instead of $N/2$, processing elements result in $<10\,\%$ throughput reduction for codes of length $N \leq 2^{20}$. Figure 3.2 illustrates these results for codes of lengths 2^{10}, 2^{12}, and 2^{20}.

The semi-parallel SC (SP-SC) decoder of [4] consists of three major parts: the processing elements, the partial-sum update logic, and the memory, which will be briefly described in this section.

3.2.1 Processing Elements

A PE is the core functional unit of the SP-SC decoder and can perform the f (3.2) and g (3.3) functions. To reduce the implementation complexity, [4] presents a

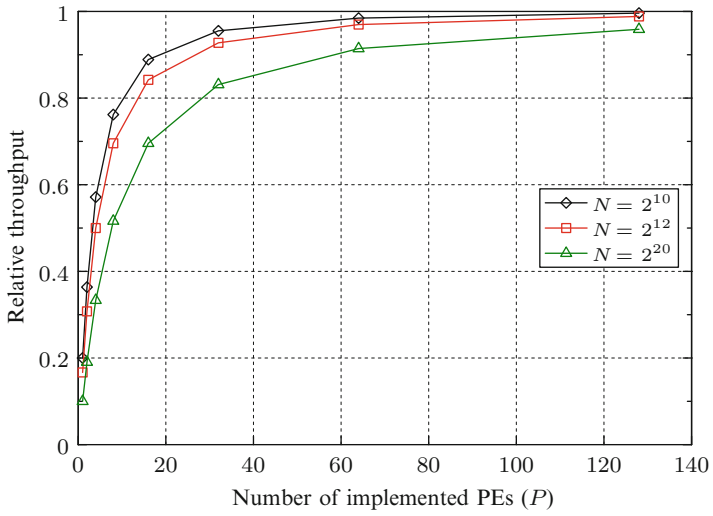

Fig. 3.2 Throughput of the SP-SC decoder relative to a fully parallel SC decoder at different code lengths

merged-PE in which hardware resources are shared between the f and g functions. No contention arises from applying this optimization as a PE is never required to perform f and g simultaneously. Since the f function is more complex than g, the decoder uses sign-magnitude representation of LLR values which simplifies the magnitude comparisons. An array of P processing elements is implemented and is fed up to $2P$ LLR values every cycle. In addition, it has access to up to P partial-sum values when g is performed. A maximum of P LLR values can be produced every cycle.

3.2.2 Partial-Sum Update Logic

Partial sums are the results of combining bit estimates in various stages of the decoder and are used as inputs to the g function. Each estimated bit is involved in multiple partial sums that are updated once the bit estimate is available. This is accomplished by storing the partial sums in independent registers with appropriate enable logic.

3.2.3 Memory

There are two major memory groups in the SP-SC decoder: the LLR memory and the partial-sum memory. The PEs read and write LLR value simultaneously;

Table 3.1 FPGA synthesis results on the Altera Stratix IV EP4SGX530KH40C2 using polar codes of different lengths

N	LUT	FF	RAM (bits)	f (MHz)	T/P (Mbps)
2^{10}	4,130	1,691	15,104	173	85 R
2^{12}	8,635	4,769	48,896	152	73 R
2^{14}	29,897	17,063	184,064	113	53 R
2^{17}	221,471	131,764	1,445,632	10	4.6 R

Table 3.2 ASIC SP-SC decoder implementation with $N = 1024$, $k = 512$, $P = 64$

Technology	UMC 180 nm
Core area	1.71 mm^2
Chip area	1.72 mm^2
Gate count	183,637
Frequency	150 MHz
Throughput	48.75 Mbps
Voltage	1.3 V
Power	67 mW
Energy efficiency	1.37 nJ/bit

therefore, the LLR memory must be able to provide the new data when write-while-reading condition occurs. This is achieved using a P-LLR bypass buffer composed flip-flops whose contents are provided in the case of a write-while-reading; while a random access memory (RAM) is used for the remaining value to save area. As mentioned in Sect. 3.2.2, the partial-sum memory is implemented using registers.

3.2.4 Implementation Results

The SP-SC decoder was first implemented on a field-programmable gate-array (FPGA) [4]. It was able to achieve a coded throughput of 85 Mbps for codes of length 1024, but the throughput decreased for longer as result of the partial-sum update logic limiting the decoder frequency. Table 3.1 summarizes the resource utilization and throughput results when $P = 64$ PEs were implemented.

An application-specific integrated-circuit (ASIC) implementation of the SP-SC decoder was presented in [9]. Its differs from [4] in that it uses registers instead of RAM for the LLR memory, and it implemented the 1-bit look-ahead technique [5], which increases the throughput by 25 %. Table 3.2 lists these results.

3.3 The Belief-Propagation Decoder Implementation

The first hardware polar decoder implemented was a semi-parallel BP decoder that uses the min-sum algorithm [7]. It was shown that it required a large number of iterations, $\sim$50, to match the error-correction performance of an SC decoder for a

Table 3.3 Implementation results for the BP and SC decoders on the Xilinx Virtex IV XC4VSX25-12 using the (1024, 512) polar code

Algorithm	LUT	FF	BRAM	T/P (Mbps)
BP	2,794	1,600	12	2.78
SP-SC	2,600	1,181	5	22.22

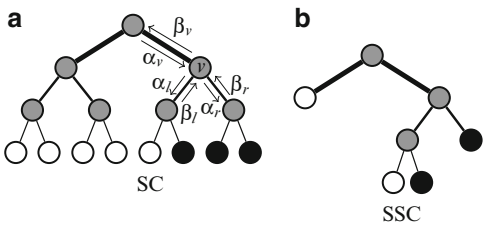

Fig. 3.3 Decoder trees corresponding to the SC and SSC algorithms

(1024, 512) polar code [4]. Table 3.3 compares the BP decoder [7] with the SP-SC decoder [4] for the (1024, 512) polar code where both decoders are configured to have the same error-correction performance. It can be seen from the table that the SP-SC decoder uses fewer memory resources and has eight times the information throughput compared to the BP decoder.

This large number of required iterations significantly degrades the BP decoder throughput in spite of the inherent parallelism of the algorithm and has limited the research targeted at implementing BP decoders.

3.4 Simplified Successive-Cancellation Decoding

The sequential nature of SC decoding and the large iteration count of BP decoder have limited the throughput of polar decoder. Simplified successive-cancellation (SSC) decoding was the first method to offer major improvements in throughput. It improves on SC decoding by exploiting the recursive nature of polar code construction to increase decoder parallelism. Figure 3.3a shows a tree representation of a polar code in which each constituent code and the two codes whose concatenation it corresponds to are represented using a nodes and its two children. The leaf nodes represent either frozen bits, the white nodes, or information bits, the black nodes.

SC decoding is performed by a node that receives a likelihood vector α_v, calculates the input to its left child, α_l, using (3.2) and uses the bit estimate of that child, β_l, to calculate the likelihood input to the right child, α_r, according to (3.3). Once the bit estimate β_r is available, it is combined with β_l to yield β_v, which is then passed to the parent node. The output β_v of a frozen bit is known a priori, 0 when frozen bits are set to 0, and the output of an information node is calculated according to (3.1) using α_v as the LLR value.

It was noted in [10] that a node whose descendants are all frozen nodes corresponds to code of rate 0 and its output β_v is known a priori. More importantly, it was shown that a node whose children are all information bits corresponds to code of rate 1 that can be decoded using maximum-likelihood decoding by applying threshold detection on α_v directly to obtain β_v. Therefore, constituent codes of rate 0 and rate 1 can be decoded directly without traversing the corresponding sub-trees in the decoder graph. This is illustrated in Fig. 3.3b, where the decoder graph is trimmed to remove such sub-trees. Such trimming was shown in [11] to improve the throughput by up to 12 times compared with SC decoding for codes of length 32768.

3.4.1 Two-Phase Successive-Cancellation Decoding

While not fully an SSC decoder, the two-phase successive-cancellation (TPSC) decoder [12] was the first to employ elements of SSC to improve decoding throughput in some parts of the decoder graph.

The aim of the TPSC decoder is to reduce implementation complexity and RAM requirements. This is achieved by exploiting the array structure of the polar codes and decoding in two distinct phases. The N bit input vector u, including the frozen bits, is arranged as a $\sqrt{N} \times \sqrt{N}$ array, U, which is encoded using $G_{\sqrt{N}}$ to yield V:

$$V = UG_{\sqrt{N}}. \tag{3.6}$$

The codeword array, X, is obtained from V using

$$X = V^T G_{\sqrt{N}}. \tag{3.7}$$

When X is rearranged into a $1 \times N$ vector, it is equal to $x = uG_N$.

The TPSC decoder divides the decoding process into $\sqrt{N}$ cycles. Each cycle consists of two phases: the first corresponds to (3.7) and the second to (3.6). Since the first phase decoder, P1, corresponds to the larger stages in the polar code, it stores computations in RAM, which is area efficient and has addressing logic built-in. While P2 uses flip-flops, which are faster than RAM, but scarcer on FPGAs.

Any soft-input hard-output polar decoder can be used to implement the P2 decoder. The authors in [12] used a parallel SC decoder with 1-bit look-ahead storing its results in registers. The P1 decoder is a soft-input soft-output SC decoder that outputs $\sqrt{N}$ LLR values in parallel. P1 loads the channel LLR values from RAM and stores the LLR inputs to the boundary stage, $\log_2 \sqrt{N}$ in registers. P2 reads those values and calculates the partial sums for the boundary stage, which are then used by P1 to continue decoding. The architecture in [12] makes no provisions for storing the results of internal calculations in P1, reducing the memory required by the decoder,

Table 3.4 TPSC implementation results on the Altera Stratix IV EP4SGX530KH40C2

N	LUT	FF	RAM (bits)	f (MHz)	T/P (Mbps)
2^{10}	1,940	748	7,136	239	112R
2^{14}	7,815	3,006	114,560	230	118R

but increasing latency as these values are recalculated for every decoding cycle. To reduce latency, the TPSC decoder implements the SSC rate-0 and rate-1 tree pruning operations, but only at the boundary stage.

The implementation results, in Table 3.4, show that the TPSC achieves higher throughput than the SP-SC decoder. The recalculation of P1 results increases decoder latency; however, the resulting architecture scales well with code length and is able to maintain a high clock frequency. The utilization of the SSC optimization also helps in increasing the throughput.

3.5 Fast-SSC Decoding

Fast-SSC decoding [13] expands on SSC decoding by directly decoding more types of constituent codes with low-complexity maximum-likelihood algorithms. In addition, it combines multiple operations to reduce the number of memory accesses and further improve throughput.

Three additional classes of constituent codes are identified in [13]: repetition codes, single parity-check (SPC) codes, and length-four codes not covered by the previous two cases.

Repetition constituent codes correspond to sub-trees whose right-most leaf is an information node while the others are frozen. They are more common in low rate polar codes than in high rate ones. Their ML decoding is simple: the elements of the input α are summed and threshold detection is used to determine result which is replicated to populate the output vector β, i.e., for all i values

$$\beta[i] = \begin{cases} 0 & \text{when } \sum_j \alpha[j] \geq 0, \\ 1 & \text{otherwise.} \end{cases}$$

The implementation of this rule is accomplished using an adder tree. It was found in [13] that the number of repetition constituent codes of length greater than 16 was small in the high-rate codes of interest. Therefore, the maximum length of repetition codes to be directly decoded was set to 16. Due to the small code lengths and the improved algorithm, the decoding of repetition constituent codes takes one clock cycle in the Fast-SSC decoder instead of up to nine in the SSC decoder.

Table 3.5 Latency of the SPC decoding algorithm for constituent codes of different lengths, N_v, when 512 LLR values can be read simultaneously

$N_v \in$	$(0,8]$	$(8,64]$	$(64,256]$	$(256,+\infty)$
Latency (cycles)	0	1	2	$N_v/512+3$

SPC codes arise when the left-most leaf in a sub-tree is frozen but not any of the other leaves. The following low-complexity algorithm, in which $h(.)$ refers to the threshold detection function (3.1), is used to implement ML decoding

$$\beta[i] = \begin{cases} h(\alpha[i]) \oplus \sum_j h(\alpha[j]) & \text{when } i = \arg\min |\alpha[i]|, \\ h(\alpha[i]) & \text{otherwise;} \end{cases}$$

where $\oplus$ is binary addition (the XOR operation). In other words, the output is the hard decision of the input with one exception: if the parity check on the hard decision is not satisfied, the decision of the bit corresponding to the least reliable input is flipped. As SPC constituent codes are common in high-rate polar codes and their lengths can be large, a pipelined design was proposed in [13] whose latency values are shown in Table 3.5, where a latency of 0 cycles indicates that the SPC decoder output will be ready within one clock cycle. For comparison, an SSC decoder has a latency of 15 cycles when $N_v = 16$.

It was observed in [13] that once rate-one, rate-0, repetition, and SPC constituent codes are accounted for, only one code of length four remains, its generator matrix is $[0001;0100]$ and can be decoded using exhaustive-search ML by testing four codewords. Its decoder was implemented using combinational logic.

3.5.1 Node Mergers

The Fast-SSC algorithm further reduces decoding latency by reducing the number of memory access requests. The first method by which this is achieved is to eliminate nodes of rate 0: since the β output of a rate 0 node is known a priori to be zero, its parent can calculate α_r immediately once α_v is available, without calculating α_l and waiting for β_l.

When the right child corresponds to an SPC or rate-1 code, the calculations of α_r, β_r, and β_v can be performed simultaneously, eliminating multiple memory read and write operations. If the left child corresponds to a rate-0 code, calculating β_v begins as soon as α_v is available.

The final form of node mergers presented in [13] is a special case for constituent codes of length 8 that correspond to a node with a repetition left child and an SPC right child. In this case, two SPC decoders are employed simultaneously: one

Table 3.6 The functions performed by the Fast-SSC decoder

Name	Description
F	Calculate α_l
G	Calculate α_r
COMBINE	Calculate β_v by combining β_l and β_r
COMBINE-0R	Same as COMBINE, but $\beta_l = 0$
G-0R	Same as G, but $\beta_l = 0$
P-R1	Calculate β_v
P-RSPC	Calculate β_v
P-01	Same as P-R1, but $\beta_l = 0$
P-0SPC	Same as P-RSPC, but $\beta_l = 0$
ML	Calculate β_v using exhaustive-search ML decoding
REP	Calculate β_v using a repetition decoder
REP-SPC	Calculate β_v using a repetition-SPC decoder

assumes that the output of the repetition decoder is 0 and the other 1. Once the output of the repetition decoder is available, the output of the correct SPC decoder is used to calculate β_v.

To summarize, the operations performed by the decoder are listed in Table 3.6.

3.5.2 Overall Decoder Architecture

Due to the large number of nodes and node combinations, it was proposed in [13] that representing the polar code structure as a precomputed list of instructions would lead to more efficient decoders. As a result, the proposed decoder has an architecture similar to that of a processor. An overall view of this decoder is shown in Fig. 3.4.

Before the decoding process starts, instructions are loaded into the instruction memory. Channel LLR values are loaded into the channel RAM via the channel loader. The controller fetches the first instruction and the decoding process starts. α values are read from α-RAM and channel RAM and written to α-RAM. Similarly, β values are written to and read from β-RAM and the estimated codeword is written to the codeword RAM. Using separate memories for internal α values and the channel LLR values is required to enable loading-while-decoding, which is required to prevent the decoder from stalling and to maintain throughput.

3.5.3 Processing Unit Architecture

The processing unit contains the logic required to perform the operations needed by all the nodes and the merged nodes. These operations are listed in Table 3.6 and Fig. 3.5 shows the architecture of the processing unit performing them. The inputs

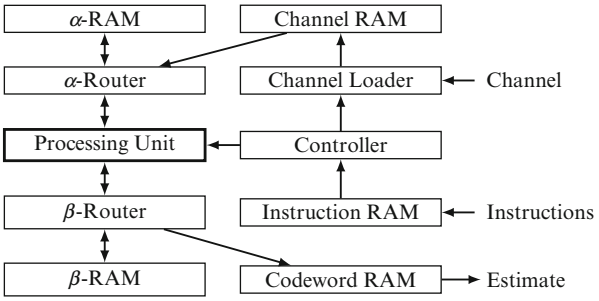

Fig. 3.4 Top-level architecture of the Fast-SSC decoder

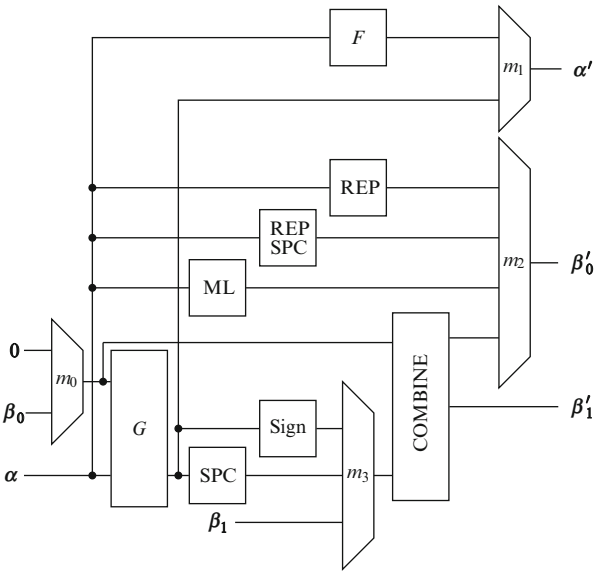

Fig. 3.5 Processing unit architecture

to this unit are: α a vector of up to $2P$ valid LLRs, β_0 a vector of up to P valid bit values generated by the left child, β_1 a vector of up to P valid bit values generated by the right child, and 0 a P-bit all-zero vector. The outputs are: α' a vector of up to P valid LLRs corresponding to the α_l or α_r outputs of the node; and β'_0 and β'_0, two vectors of up to P valid bit values each, corresponding to the β_v output of the node.

The two functions producing α outputs are F and G. The multiplexer m_1 selects between their outputs. In addition, G has a specialization when the node's left child is a rate-0 node, denoted G-0R. The multiplexer m_0 is used to set the β input to G function to 0.

Repetition, Repetition-SPC, and ML nodes are limited to constituent codes of length less than or equal to P in [13]. Therefore, their output is entirely contained

Table 3.7 Post-fitting results for a code of length 32768 on the Altera Stratix IV EP4SGX530KH40C2

P	Q	LUTs	Registers	RAM (bits)	f (MHz)
64	$(6,4,0)$	6,830	1,388	571,800	108
	$(7,5,1)$	8,234	858	675,864	100
256	$(6,4,0)$	25,866	7,209	536,136	108
	$(7,5,1)$	30,051	3,692	700,892	104

in β_0'. The multiplexer m_2 selects the correct β_0' from among the candidates, which also include the first half of COMBINE's output.

β_1' is always the second half of COMBINE's output. The left input to COMBINE is selected by m_0 as either β_0 or 0. Due to node mergers, the second input varies: when the function performed is P-01 or P-R1, it is the sign of the output from G; for P-0SPC and P-RSPC, the input come from the SPC decoder that uses the output of G as its input; finally, in the case of COMBINE and COMBINE-0X, β_1 is used. This selection is process performed by m_3.

3.5.4 Implementation Results

The Fast-SSC decoder has been implemented and verified an FPGA using different quantization schemes. It was noted in [13] that using 7-bit and 5-bit words, including one fractional bit, to represent internal and channel LLR values, respectively, was sufficient to match the performance of the floating-point decoder. This scheme is denoted (7, 5, 1). A (6, 4, 0) scheme was shown to offer excellent performance as well. The implementation results for codes of length 32768 are shown in Table 3.7.

Memory bandwidth, constrained by P, has a significant impact on throughput: for the (32768, 29492) code with (6, 4, 0), the information throughput values were 547 and 1,081 Mbps for $P = 64$ and 256, respectively. The quantization scheme used did not affect throughput significantly—throughput was degraded from 1,081 to 1,077 Mbps when switching to the (7, 5, 1) scheme—but had a significant impact on memory resources.

3.6 Implementation Comparison

When comparing the different polar decoder implementations, it is important to ensure that they are capable of sustaining their throughput. The decoders must support loading-while-decoding or their throughput will be degraded. The easiest method to implement loading-while-decoding is to buffer additional codewords. In this section, the RAM numbers were modified where needed to ensure that the decoders can buffer an additional codeword. This is indicated using a * next to the algorithm name in Table 3.8. In that table it can be seen that TP-SC uses the fewest resources and has the highest clock frequency; while Fast-SSC has the highest

Table 3.8 Post-fitting and information throughput results for a (16384, 14746) code on the Altera Stratix IV EP4SGX530KH40C2

Algorithm	P	LUTs	Reg.	RAM (bits)	f (MHz)	Info. T/P (Mbps)
SP-SC*[4]	64	29,897	17,063	265,984	113	48
TPSC*[12]	128	7,815	3,006	196,480	230	106
Fast-SSC[13]	128	13,388	3,688	273,740	106	824
Fast-SSC[13]	256	25,219	6,529	285,336	106	1,091

throughput. The high frequency of TP-SC is a result of the aggressive buffering using registers. Fast-SSC uses 1.6, 1.2, and 1.4 times the LUTs, registers, and RAM compared to TP-SC, respectively, but has 7.8 times the throughput when both decoders use $P = 128$. Increasing P to 256 in the Fast-SSC increases the LUTs and registers used significantly, and increases the information throughput to 1,091 Mbps.

References

1. Arikan E (2009) Channel polarization: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. IEEE Trans Inf Theory 55(7):3051–3073
2. Sasoglu E, Telatar E, Arikan E (2009) Polarization for arbitrary discrete memoryless channels. In: Proceedings of the IEEE information theory workshop ITW 2009, pp 144–148
3. Mori R, Tanaka T (2010) Channel polarization on q-ary discrete memoryless channels by arbitrary kernels. In: Proceedings of the (ISIT) symposium on IEEE international information theory, pp 894–898
4. Leroux C, Raymond A, Sarkis G, Gross W (2013) A semi-parallel successive-cancellation decoder for polar codes. IEEE Trans Signal Process 61(2):289–299
5. Zhang C, Yuan B, Parhi K (2012) Reduced-latency SC polar decoder architectures. In: 2012 IEEE International Conference on Communications (ICC), pp 3471–3475
6. Hussami N, Urbanke R, Korada SB (2009) Performance of polar codes for channel and source coding. In: Proceedings of the IEEE international symposium on information theory ISIT 2009, pp 1488–1492
7. Pamuk A (2011) An FPGA implementation architecture for decoding of polar codes. In: 2011 8th international symposium on wireless communication systems (ISWCS), pp 437–441
8. Leroux C, Raymond AJ, Sarkis G, Tal I, Vardy A, Gross WJ (2012) Hardware implementation of successive-cancellation decoders for polar codes. J Signal Process Syst 69(3):305–315
9. Mishra A, Raymond A, Amaru L, Sarkis G, Leroux C, Meinerzhagen P, Burg A, Gross W (2012) A successive cancellation decoder ASIC for a 1024-bit polar code in 180nm CMOS. In: 2012 IEEE Asian solid state circuits conference (A-SSCC), pp 205–208
10. Alamdar-Yazdi A, Kschischang FR (2011) A simplified successive-cancellation decoder for polar codes. IEEE Commun Lett 15(12):1378–1380
11. Sarkis G, Gross WJ (2013) Increasing the throughput of polar decoders. IEEE Commun Lett 17(4):725–728
12. Pamuk A, Arikan E (2013) A two phase successive cancellation decoder architecture for polar codes. In: Proceedings of the IEEE international symposium on information theory ISIT 2013, July 2013, pp 1–5
13. Sarkis G, Giard P, Vardy A, Thibeault C, Gross WJ (2014) Fast polar decoders: algorithm and implementation. IEEE J Sel Areas Commun (JSAC)

Chapter 4
Parallel Architectures for Turbo Product Codes Decoding

Camille Leroux, Christophe Jego, and Patrick Adde

4.1 Introduction

High throughput telecommunication systems such as long-haul optical transmission or passive optical networks require powerful error correcting codes in order to increase their optical budget. In such speed-constrained applications, the classical (255,239) Reed–Solomon code is gradually being replaced by more powerful forward error correction (FEC) schemes. In [1], turbo product codes (TPC) [2] are seen as the third generation FEC for optical transmission systems. TPC tend to be good candidates for emerging optical systems. The inherent parallel structure of the product code matrix confers to TPC a good ability for parallel decoding. Nevertheless, enhancing parallelism rate rapidly induces the use of a prohibitive amount of memory. Many architectural solutions were proposed to efficiently exploit parallelism in TPC decoding. Moreover, TPC decoding provides several level of parallelism and it is not always clear which level is the most efficient. In this chapter, several parallelism level of TPC decoding are identified. Each parallelism level is characterized in terms of the potential hardware efficiency that it may bring to the architecture. From this design space exploration, we focus on one efficient architecture that exploits different levels of parallelism.

After a brief introduction of the TPC coding and decoding concept in Sect. 4.2, a straightforward hardware implementation of a TPC decoder is presented in Sect. 4.3 in order to highlight the inherent problem of parallelization in TPC decoding.

C. Leroux • C. Jego (✉)
IMS Laboratory, Bordeaux-INP, France
e-mail: camille.leroux@ims-bordeaux.fr; christophe.jego@ims-bordeaux.fr

P. Adde
LabSticc, TELECOM Bretagne, France
e-mail: patrick.adde@telecom-bretagne.eu

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_4

Then, Sect. 4.4 defines and characterizes all the parallelism levels in TPC decoding. A review of existing architectural solutions is given before the detailed description of a TPC decoder architecture without any interleaving resource. This TPC decoder includes a fully parallel SISO decoder architecture which is also described in detail. Finally, Sect. 4.7 gives some synthesis results and demonstrates the efficiency of the proposed TPC decoder by comparison with current TPC decoders.

4.2 TPC Coding and Decoding Principles

The concept of product codes is a simple and efficient method to construct powerful codes with a large minimum Hamming distance d using cyclic linear block codes [3]. Despite the existence of several other decoding algorithms [4], the Chase–Pyndiah algorithm [2] is known to give the best trade-off between performance and decoding complexity [5]. Product codes were adopted in 2001 as an optional correcting code system for both the up link and down link of the IEEE 802.16 standard (WiMAX) [6].

4.2.1 Product Codes

Let us consider two systematic cyclic linear block codes C_1 having parameters (n_1, k_1, d_1) and C_2 having parameters (n_2, k_2, d_2) where n_i, k_i, and d_i $(i = 1, 2)$ stand for code length, code dimension, and minimum Hamming distance, respectively. As shown in Fig. 4.1, the product code $P = C_1 \times C_2$ is obtained by (a) placing $(k_1 \times k_2)$ information bits in a matrix of k_1 rows and k_2 columns, (b) coding the k_1 rows using code C_2, and (c) coding the n_2 columns using code C_1.

Considering that C_1 and C_2 are linear block codes, n_1 rows are codewords of C_2 exactly as all n_2 columns are codewords of C_1 by construction. Furthermore, the

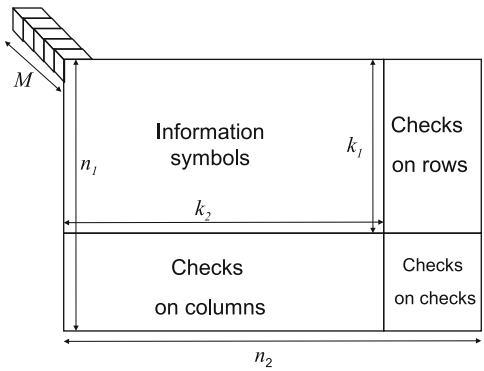

Fig. 4.1 Product code matrix structure

parameters of the resulting product code $C_P(n_p, k_p, d_p)$ are given by $n_p = n_1 \times n_2$, $k_p = k_1 \times k_2$, and $d_p = d_1 \times d_2$. The code rate R_p is given by $R_p = R_1 \times R_2$. Thus, it is possible to construct powerful product codes using two linear block codes. In the following sections, without loss of generality, we consider a squared product code, meaning that $n_1 = n_2 = n$. The most commonly used component codes are Bose Chaudhuri Hocquenghem (BCH) codes. These codes are an infinite class of linear cyclic block codes that have capabilities for multiple error detection and correction. Reed–Solomon (RS) codes can also be used as component codes. RS codes are non-binary codes in which symbols are represented on $M_{RS} = log(n+1)$ bits while $M_{BCH} = 1$. As discussed later, RS-TPC present several advantage in terms of parallelism and decoding performance [7, 8]. Without loss of generality, in the remaining of the chapter, unless specified otherwise, we assume that $M = 1$.

4.2.2 Iterative Decoding of Product Codes

Product codes usually have high dimension which precludes Maximum-Likelihood (ML) soft-decision decoding. Yet, the particular structure of this code family lends itself to an efficient iterative "turbo" decoding algorithm offering close-to-optimum performance at high enough signal-to-noise-ratios (SNRs). The Turbo-decoding of product codes consists in successively alternate decoding rows and columns using soft-input soft-output (SISO) decoders. Repeating this soft-decision decoding during several iterations enables the reduction of the bit error rate (BER). Each decoder has to compute soft information R'_{it+1} from the channel received information R and the information R'_{it} computed during the previous half-iteration. Despite the existence of several other decoding algorithms [4], the Chase–Pyndiah algorithm is known to give the best trade-off between performance and decoding complexity [5]. The Chase–Pyndiah SISO algorithm for a $t = 1$ BCH code [2, 9] is summarized below. t represents the maximum number of correctable errors for the component code.

1. Search for the L least reliable bits in the previous half-iteration output vector R'_{it} such that λ_i represents the ith minimum, $1 < i < L$.
2. Compute the syndrome $S(t_0)$ of R'_{it},
3. Compute the parity of R'_{it},
4. Generate p test patterns τ_i obtained by inverting some of the L least reliable bits ($p \leq 2^L$).
5. **For each** test pattern $(1 \leq i \leq p-1)$
   - Compute the syndrome $S(\tau_i)$,
   - Correct the potential error by inverting the bit position $S(\tau_i)$,
   - Recompute the parity considering the detection of an error and the parity of R'_{it},
   - Compute the square Euclidean distance (metric) M_i between R'_{it} and the considered test pattern τ_i.

6. Select the Decided Word (DW) among test patterns having the minimal metric (M_{DW}) and choose Cw competitors codewords $\mathbf{c}_i$ ($1 < i < Cw$) having the second, third, …,. ith minimum metric.
7. **For each** symbol of the DW,

- Compute the new reliability F_{it}:

$$F_{it} = \begin{cases} \beta_{it} = (|R'_{it}| + \sum_{i=1}^{L} \lambda_i) - \min(M_i) \text{when no competitor exists} \\ F_{it} = \min_2(M_i) - \min(M_i) \text{otherwise,} \end{cases}$$

- Compute extrinsic information $W_{it} = F_{it} - R'_{it}$,
- Add extrinsic information (multiplied by α_{it}) to the channel received word, $R'_{it+1} = R + \alpha_{it} W_{it}$.

As explained in [10], decoding parameters L, p, Cw and the number of quantization bits q of the soft information have a considerable effect on decoding performance and complexity. The α_{it} coefficient allows decoding decisions to be damped during the first iterations. β_{it} is an estimation of F_{it} when no competitor exists. As detailed in [11], it is based on the least reliable bits value.

Figure 4.2 shows the BER performance of various $t = 1$ BCH and RS codes. In general, for a fixed t value, the code rate increases with N. This explains why the BER curves are shifting to the right when N increases. However, for large codelengths, the slope is steeper.

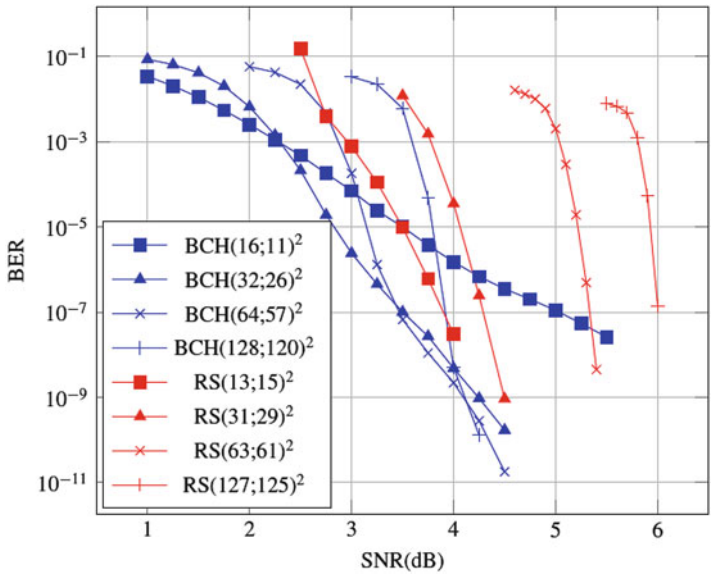

Fig. 4.2 BER performance of various BCH and RS product codes on an AWGN channel

4.3 Straightforward Hardware Implementation of a TPC Decoder

4.3.1 Global TPC Decoder Architecture

In a straightforward implementation of a TPC decoder, the channel information matrix R (consisting in n^2 q-bits LLRs) is stored in a memory. As shown in Fig. 4.3, since the SISO decoder reads R_A during the whole decoding process, this memory has to be duplicated so that the next channel information matrix R_B can be written while the decoder processes the current matrix R_A. A single sequential SISO decoder reads information from the R memory and performs the decoding process by updating the R' messages iteratively. Assuming I decoding iterations, the SISO decoder should update $2 \times I \times n^2$ LLRs.[1] In the most favorable case, let us assume that the SISO decoder is able to update one LLR per clock cycle, the resulting throughput is $T = f/(2In^2)$, where f is the clock frequency. For a $(32, 26)^2$ BCH code with six decoding iterations and a clock frequency of 500 MHz, the resulting throughput is 40 Kb/s. This kind of architecture is clearly too slow for high throughput applications. In this chapter, various methods to enhance the parallelism are reviewed.

4.3.2 Sequential SISO Decoder Architecture

The TPC decoder architecture described in Fig. 4.3 includes a SISO decoder that sequentially process incoming LLRs. Figure 4.4 shows the structure of such a sequential SISO decoder. It is subdivided into four units.

The reception unit

- computes the syndromes of the incoming vectors,
- selects the p least reliable bits.

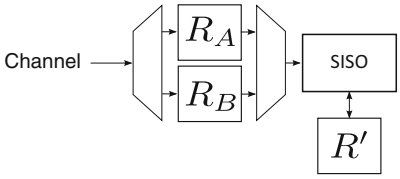

Fig. 4.3 Sequential implementation of a TPC decoder

[1] A full iteration corresponds to a row-wise decoding followed by a column-wise decoding, which explains why the R' matrix has to be updated $2I$ times.

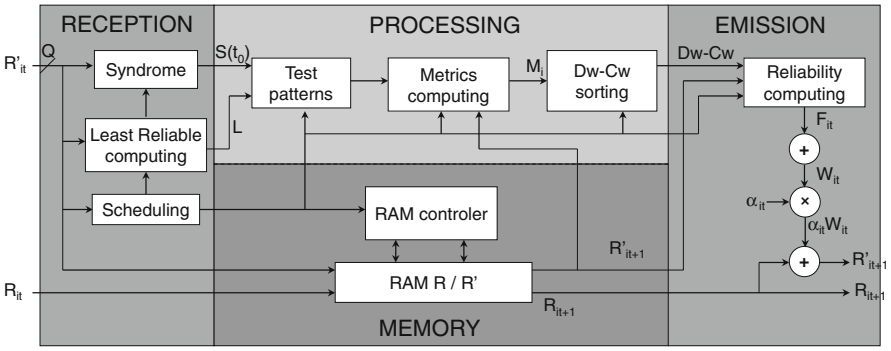

Fig. 4.4 Architecture of a SISO decoder

The processing unit

- determines the test vectors by inverting some of the least reliable bits,
- computes the metric of each test vector,
- selects the most likely test vector (the one with the minimum soft-distance)
- selects the Cw concurrent test vectors (2nd minimum, 3rd minimum, etc.).

The emission unit

- computes the new reliability F_{it} of each outcoming bit of the considered vector,
- computes and ponderates the extrinsic information $\alpha_{it}W_{it}$,
- generates the soft-output LLRs for the next half-iteration R'_{it+1}.

The memory unit stores the channel information and the soft information for the current half-iteration.

In terms of latency, the syndrome and the least reliable bits can only be computed once the whole $n\times$-LLR vector has been shifted in. Only then, the test vectors processing and the soft-output computation can be performed. This means that it takes at least n clock cycles to read R and $[R'_{it}]$, 1 clock cycle to perform the test vector computation,[2] and n clock cycles to write back the n LLRs in the R' memory. This means that n LLRs require at least $2n+1$ clock cycles to be updated which corresponds to a throughput of $T = f/(4In^2)$ (we assume that $n \gg 1$). Taking the SISO latency into account, the previously estimated 40 Kb/s TPC decoder has in fact a throughput of only 20 Kb/s.

The hardware complexity of a sequential SISO decoder is rather low, thanks to its serial-processing nature. The SISO decoder designed in [10, 12] has an equivalent complexity of a few thousands gates. The computational complexity of the SISO decoder depends on the choice of algorithmic parameters. As mentioned in Sect. 4.2.2, the Chase–Pyndiah algorithm includes parameters L, p, Cw, q which impact on both the decoding performance and the computational complexity of the

[2]This assume that one is able to design a parallel processing unit that computes and select metrics in a single clock cycle.

TPC decoder. Depending on the application one should identify a parameter set that enables sufficient decoding performance while minimizing the hardware footprint of the resulting SISO decoder. In [10,12], a case of study is detailed for a $(32,26)$ BCH code SISO decoder. Depending on the parameter set that is selected, the complexity of a SISO decoder varies by a factor 2. This shows that the algorithmic parameter set is an important factor to take into account when designing a TPC decoder.

4.4 From Parallelism Levels to Parallel Architectures

An architecture can be characterized by different metrics such as throughput, latency, hardware complexity, power consumption, routing density, etc. In this study, we aim at high speed architectures with low hardware complexities. Consequently, the performance is measured with throughput (T) while the cost function is the hardware area (A). In such a context, the efficiency of an architecture is defined as the throughput/complexity ratio : $E = T/A$. An efficient architecture would process a high data rate with a low hardware area.

The parallelism of an architecture can be defined as "the ability of the system to process several data in parallel." We formally define the parallelism P of a decoder as the number of bits that can be processed/decoded in a single clock cycle. The parallelism directly impacts the performance of an architecture. In order to quantify the benefit/disadvantage brought by the application of a parallelism P_i to an architecture, we define three metrics, the *speed gain* G_S, the *area ratio* R_C, and the *efficiency gain* G_E:

$$\begin{cases} G_S(P_i = p) = \frac{T_{P_i=p}}{T_{P_i=1}} \\ R_A(P_i = p) = \frac{A_{P_i=p}}{A_{P_i=1}} \\ G_E(P_i = p) = \frac{E_{P_i=p}}{E_{P_i=1}} = \frac{G_S(P_i=p)}{R_A(P_i=p)} \end{cases}$$

A parallelism level P_i is considered to be *effective* if $G_S(P_i) > 1$, while it is *efficient* when $G_E(P_i) > 1 \iff G_S(P_i) > R_A(P_i)$. One should notice that several parallelism levels may be combined but it may also be impossible to associate them. The exploitations of several parallelism levels at the same time depend on the architecture that implements these levels. In the remaining of this section, all parallelism levels in TPC decoding are detailed and characterized from the highest level (frame parallelism) down to the lowest level (intra-symbol parallelism). For each level, we provide a condition that makes the use of the considered level efficient. We also provide some reference of existing TPC decoders that use these parallelism levels.

4.4.1 Frame Parallelism

The highest level of parallelism can be observed at the frame level and this is known as *frame parallelism*. It is a form of spatial parallelism and is suitable to any decoding scheme. In TPC decoding, a frame is defined as a product code matrix. The frame parallelism consists in duplicating the processing resources, e.g., the turbo-decoder. By using this parallelism level in TPC decoding, P_{frame} matrices can be decoded at the same time. Considering P_{frame} turbo-decoders that have the same throughput T_0, the *speed gain* and *area ratio* are equivalent: $G_S = R_A = P_{frame}$. Consequently the efficiency does not increase with P_{frame}: $G_E = 1$. Actually, this level of parallelism is only limited by the affordable silicon area. Although frame parallelism makes TPC decoder architecture more effective, it does not improve its efficiency. Moreover some buffering/multiplexing resources are needed to broadcast incoming LLRs to the different decoders. The only advantage of frame level parallelism is the design time since it can be implemented by a straightforward duplication of resources on the silicon.

4.4.2 Iteration Parallelism

In a sequential TPC decoder implementation, each iteration is performed by the same SISO decoder that reads and writes data in the Interleaving Memories (IM). It is however possible to exploit the *iteration parallelism* by duplicating the elementary decoder and the associated memories in a pipelined structure. The memories have to be duplicated so that all SISO decoders can work in parallel. The maximum depth of such a structure equals to the maximum number of iteration it_{max}. Iteration parallelism is a type of temporal parallelism. Here again, the throughput benefit equals to the complexity ratio : $G_S = R_A = P_{it}$. It means that the iteration parallelism does not improve efficiency. Figure 4.5 shows a pipelined TPC decoder. It includes I stages; each of which processing one frame. This explains why the channel memory R has to be duplicated. It is also possible to implement less than

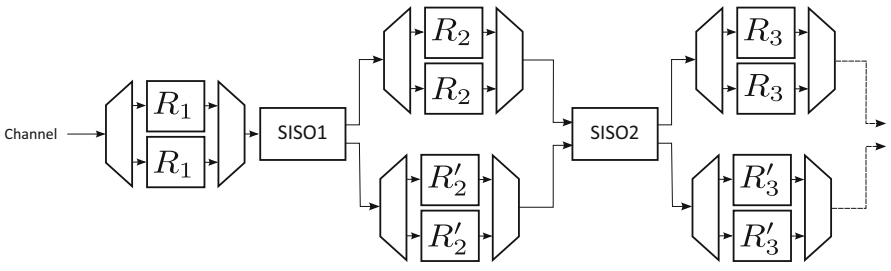

Fig. 4.5 Pipelined TPC architecture

I stages and to loop back on the hardware resources. The iteration parallelism was applied in [7] where five iterations are duplicated over five different FPGA devices. It enables to reach a throughput of 5 Gb/s.

4.4.3 Sub-block Parallelism

In a product code matrix, each row (column) is encoded independently from the others (See Sect. 4.2.1). This interesting property may also be used during the decoding process, where each row (column) is decoded independently. In an implementation prospective, it means that more than one decoder can be assigned to row (column) decoding. Considering a product code matrix of size n^2, a maximum number of n decoders can be duplicated for row (column) decoding. We designate this parallelism level as *sub-block parallelism* P_{sb}. Assuming that the duplication of SISO decoders does not induce interleaving resources duplication, G_E can be expressed as:

$$G_E = \frac{P_{sb}(A_{SISO} + A_\pi)}{P_{sb}A_{SISO} + A_\pi}$$

$$G_E > 1 \Longleftrightarrow P_{sb} > 1$$

A_{SISO} and A_π are the areas of the SISO decoder and the interleaving resource, respectively.

In [10,13–15] solutions based on Barrel-shifter and Omega network are proposed to avoid data access conflicts when $P_{sb} = n$. This makes the *complexity ratio* lower than the *speed gain*, which means that the *efficiency gain* of the architecture increases.

4.4.3.1 Barrel Shifter

In a straightforward application of sub-block parallelism, one simply duplicates the SISO decoders. The decoder is then composed of P_{sb} SISO decoders, a memory storing n^2 q-bits LLRs from the channel R and one memory storing n^2 q-bits LLRs for the matrix $[R'_{it}]$. However, this architecture is limited by memory access conflicts. Depending on the considered iteration, the P_{sb} SISO decoders need to access a total of P_{sb} data either row-wise or column-wise. In [13], this problem is overcome for $P_{sb} = n$: a barrel shifter is introduced between SISO decoders and the interleaving register file in order to allow row/column-wise data accesses of P_{sb} data in parallel as shown in Fig. 4.6. This comes at the extra cost of a barrel-shifter with area of $O(n \log n)$. This solution enables to use the sub-block parallelism at its highest rate only: $P_{sb} = n$. The extra-complexity consists in a simple barrel shifter with a

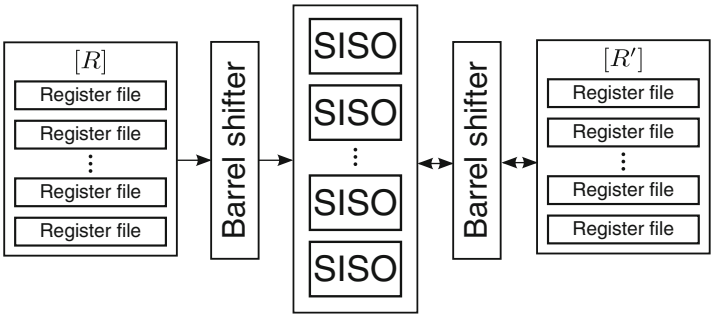

Fig. 4.6 Barrel-shifter-based parallel TPC decoder

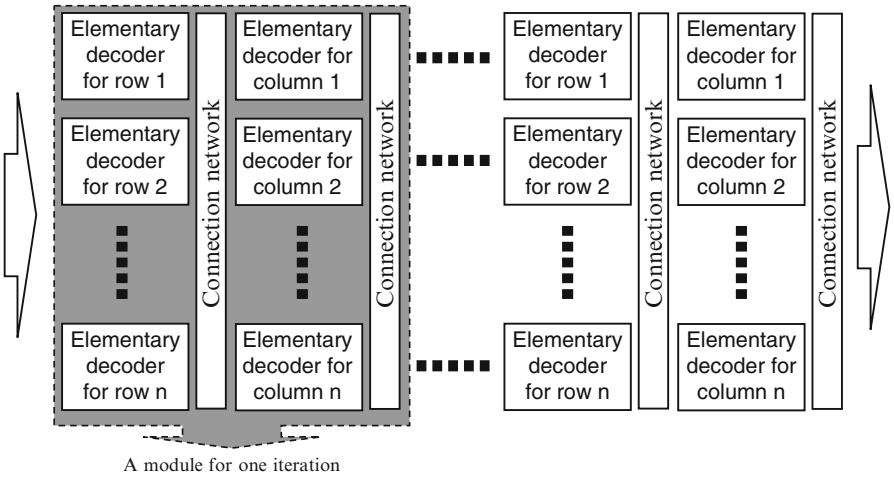

Fig. 4.7 Omega network-based parallel TPC decoder without interleaving memory

complexity of $O(n\log(n))$. However, it still includes a large amount of interleaving memory for storing R'. This is especially problematic if one wants to use iteration parallelism where the interleaving resources have to be duplicated.

4.4.3.2 Omega Network

In [10, 14, 15], it is suggested to replace the interleaving memory by a simple interconnection network (Omega network). This is made possible by the cyclic nature of the component codes (BCH or RS codes): applying a circular shift on a codeword ends up in another codeword. In terms of decoding, this means that the decoding process can start with any bits in the codeword. The decoding process is then applied on a shifting diagonal. This avoids data access conflicts as long as data are correctly routed from one iteration to another as shown in Fig. 4.7.

The interleaving-memory-less architecture was prototyped on an FPGA device [10]. This TPC decoder also has a maximal sub-block parallelism ($P_{sb} = n$), while the hardware complexity of the interleaving resources is drastically reduced since interleaving memory is no more needed.

4.4.4 Symbol Parallelism

A finer-grained parallelism is the *symbol parallelism*. It can be defined as the ability of a SISO decoder, to process P_{sym} symbols of the same sub-block (row or column) in parallel. In a sequential SISO decoder, input data are shifted in a serial manner. Every incoming symbol implies some internal metrics to be updated (syndrome, least reliable bits, ...). By increasing P_{sym}, some parts of the decoder datapath have to be duplicated, (e.g., the reliability computation stage). However, the other blocks, such as the test pattern metric computation, or the competitor vector determination block, remain identical when P_{sym} increases. Consequently, the *area ratio* is lower than the *speed gain* : $G_E > 1$. For an architecture that avoid interleaving resource duplication, the following inequality is verified:

$$G_E > 1 \Longleftrightarrow A_{DEC}(P_{sym} = p) < p \times A_{DEC}(P_{sym} = 1)$$

$A_{DEC}(P_{sym} = p)$ is the hardware complexity of a SISO decoder with a symbol parallelism equal to p. Increasing P_{sym} also means that the interleaving memory should be able to read/write more than one data during the same clock cycle. In [12, 16] solutions were provided in order to exploit this parallelism while avoiding interleaving memory duplication. Logic synthesis results confirm that the efficiency increases with P_{sym}.

4.4.4.1 Memory Merging

In [16] an architecture that uses symbol parallelism in conjunction with sub-block parallelism is proposed. The idea is to store several LLRs at the same address and to design elementary decoders able to process $P_{sym} = m$ symbols during the same clock period (denoted as m-decoders). A half-iteration structure includes n/m decoders each decoding m symbols in one clock period and an interleaving memory of size $4 \times q \times M \times n^2$. This scheme actually exploits symbol parallelism on one dimension of the matrix and sub-block parallelism on the other dimension in such a way that $P_{sb} = P_{sym} = m$. The resulting throughput is $O(m^2)$ while the overhead factor of the decoder complexity is $\sim \frac{m^2}{2}$. In this work, the maximum reached parallelism rate is $m^2 = 64$, with $m = 8$ SISO decoders.

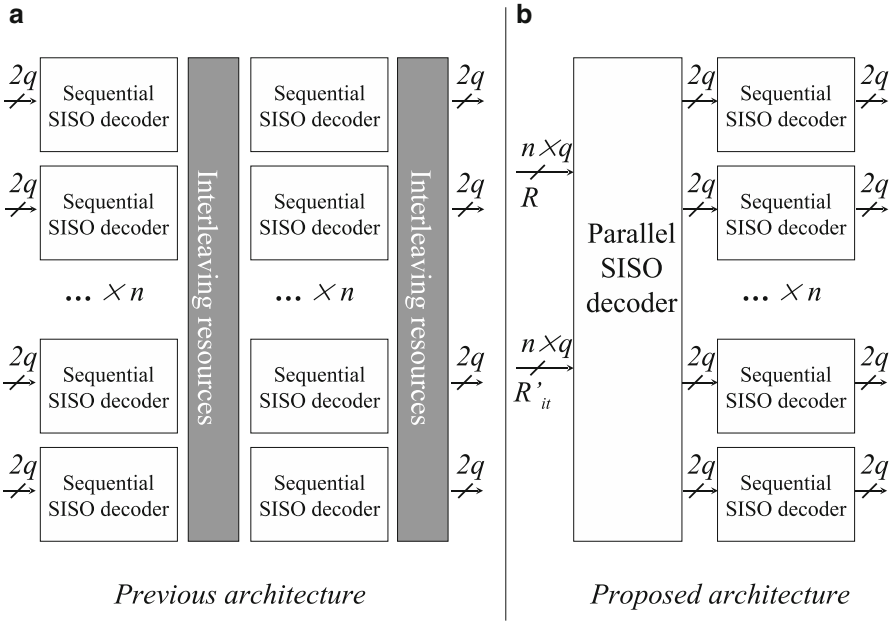

Fig. 4.8 Omega network-based parallel TPC decoder (**a**) and fully parallel SISO-based TPC decoder architecture (**b**)

4.4.4.2 Fully Parallel SISO Decoder

In [12], an architecture with $P_{sym} = n$ is proposed. A fully parallel SISO decoder enables to decode a whole column in a single clock cycle after a few cycles of latency. The n generated LLRs are then directly fed into n sequential SISO decoders which perform row decoding. In such an architecture the interleaving resources are simply removed since generated data are immediately consumed. Logic synthesis results show the higher efficiency of this architectural solution in comparison with the previously described ones. This can be easily explained by the fact that the complexity of one fully parallel SISO decoder is lower than n SISO decoders. This TPC decoder architecture will be described in detail in Sect. 4.5 (Fig. 4.8).

4.4.5 Intra-symbol Parallelism

In TPC decoding, BCH codes are often used for their good decoding performance/-complexity trade-off. In [7, 17], it was shown that using RS codes as component codes can provide similar decoding performance with a reasonable computational complexity overhead.

From an architectural point of view, the non-binary structure of RS codes enables to exploit an extra parallelism level, the *intra-symbol parallelism* P_{is}. In an RS code of size n, a symbol consists in $M = \log(n+1)$ bits (see Fig. 4.1). An RS-SISO decoder can either shift-in symbols bit by bit or symbol by symbol. It provides a maximal parallelism rate of $\max(P_{is}) = \log(n+1)$.

Similarly to the symbol parallelism, the resource sharing within the RS-SISO decoder increases the efficiency. However the *efficiency gain* provided by P_{is} is hard to estimate because it is highly related to the internal architecture of the SISO decoder. Nevertheless, it is possible to give a condition that guarantees $G_E(P_{is}) > 1$:

$$A(P_{is} > 1) < P_{is} \times A(P_{is} = 1)$$

In [7], a $(31,29)^2$ RS turbo product code decoder was designed and prototyped. It has an architecture similar to [10] but it includes RS SISO decoders that process one RS symbol per clock cycle. Moreover, the iteration parallelism is used in such a way that the decoding iterations are duplicated on the 5 FPGA devices. The resulting TPC decoder reaches 5 Gb/s.

4.4.6 Comparison of Parallelism Levels

Table 4.1 summarizes benefits of parallelism levels in TPC decoding. For each parallelism P_i, the maximum *speed gain*, the *efficiency gain*, and the P_i value that maximizes the efficiency are given. *Frame parallelism* is only limited by technological issues (e.g., silicon area). This parallelism improves the effectiveness of the architecture; it is straightforward to implement but it does not improve efficiency. *Iteration parallelism* has the same impact but is upper bounded by the maximum number of iteration required by the decoding process.

Application of lower levels of parallelism (P_{sb}, P_{sym}, and P_{is}) improves the architecture efficiency. It is even maximized for highest parallelism value. However, the use of these parallelism levels is not as straightforward as P_{frame} and P_{it}. It requires some specific schedulings and/or implementation strategies.

The TPC decoder architectures mentioned in this section use different levels of parallelism and end up with different hardware efficiency. Table 4.2 provides a comparison of the state-of-the-art TPC decoders in terms of parallelism levels and

Table 4.1 Comparison of parallelism levels in TPC decoding

P_i	$\max(G_S)$	G_E	$\arg(\max(E))$
P_{frame}	∞	$\simeq 1$	$[0;+\infty[$
P_{it}	IT_c	$\simeq 1$	IT_c
P_{sb}	n	≥ 1	n
P_{sym}	n	≥ 1	n
P_{is}	$\log(n+1)$	≥ 1	$\log(n+1)$

Table 4.2 Current TPC decoder architecture comparison

Architecture	P_i	$A_\pi(1/2iter)$	$A_{Dec}(1/2iter)$
[13]	$P_{sb} = n$	$O(2qn^2) + O(2n\log(n))$	$O(n)$
[10, 14]	$P_{sb} = n$	$O(n\log(n))$	$O(n)$
[16]	$P_{sb} = m; P_{sym} = m$	$O(2qn^2)$	$O(m^2/2)$
[12]	$P_{sb} = 1; P_{sym} = n$	$\emptyset$	$< O(n)$

hardware area. For each reference, we provide the exploited parallelism levels. The hardware area is given for a half-iteration for both the interleaving resource and the decoding resources. All these architectures could use the frame and iteration parallelism P_{frame} and P_{it} by duplicating resources.

The TPC decoder in [13] uses n sequential SISO decoders, two memories of size n^2 for R and R' and two barrel shifters. In such an architecture, the critical part is the memory resources that grow with n^2. The TPC decoder in [16] uses a combination of P_{sb} and P_{sym}. This is the first architecture that uses P_{sym}. However the SISO decoders were designed for a maximum parallelism of $m = 8$. Moreover this architecture uses memory resources for interleaving which dominate the resulting hardware area. In [10, 14], the sub-block parallelism is fully exploited. The memories and the barrel shifters are replaced by an omega network to route data from one iteration to the next. The hardware area is then dominated by the n duplicated SISO decoders.

In [10, 13, 14, 16], the rebuilding of the product code matrix is necessary between each half-iteration: memory blocks and/or routing networks are used between half-iterations to read and store R'_{it} and R. Actually, more than 50 % of the complexity is in the memory for IM-based architecture, while it represents less than 10 % for omega network-based structure [14, 15]. On the decoding resources side, increasing the parallelism rate by duplicating computation resources is inefficient since the reuse of available resources is not optimized. In [12], a fully parallel SISO decoder is cascaded with n sequential SISO decoders in such a way that interleaving resources are completely removed. In fact, the internal memory of SISO decoders is sufficient to store the required R and R' matrices. The fully parallel SISO decoder is less complex than n sequential SISO decoders which make this architecture even more efficient. In the next section, due to its higher efficiency, the TPC decoder architecture of [12] is described in detail.

4.5 TPC Decoder Architecture Based on Symbol Parallelism

4.5.1 Proposed IM-Free Architecture Using Fully Parallel SISO Decoder

Considering that one can design a SISO decoder with $P_{sym} = n$, a product code matrix can be decoded without any interleaving resource as shown in Fig. 4.9.

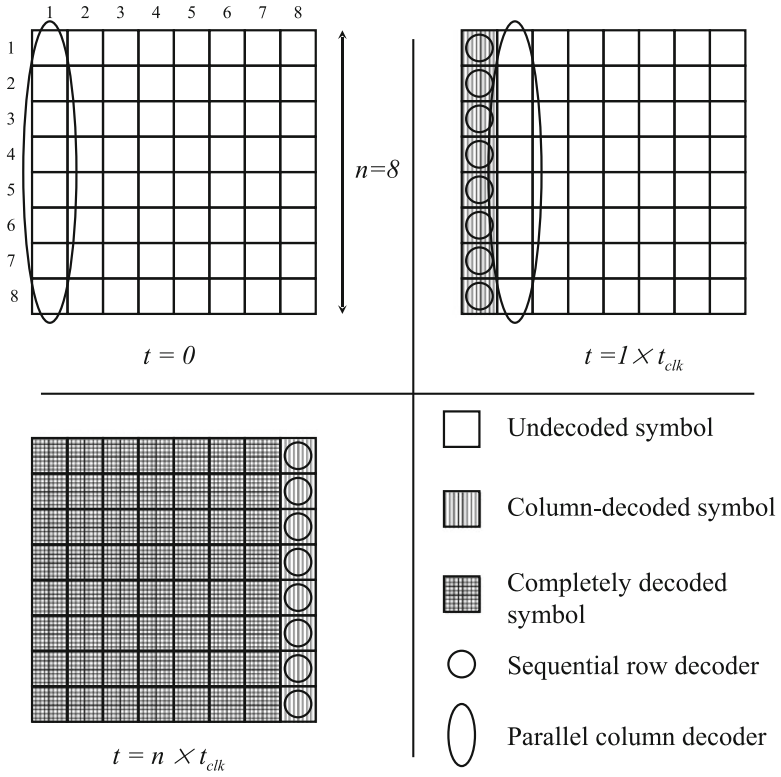

Fig. 4.9 Proposed parallel decoding scheduling of a product code matrix

At $t = 0$, the fully parallel SISO decoder processes the column 1. During the next clock period, n sequential SISO decoders ($P_{sym} = 1$) start decoding the first symbol of each row while the parallel decoder processes the column 2. During the nth clock period, sequential decoders complete matrix decoding while the parallel decoder is already decoding the next matrix. Thus, data generated by the parallel decoder is immediately consumed by the sequential decoders. Consequently, no IM or data routing resources are required between the fully parallel decoder and sequential decoders. The resulting architecture is compared to [10, 14] in Fig. 4.10 for one implemented iteration. This architecture uses row-wise P_{sb} and column-wise P_{sym}. More specifically, we have:

$$\begin{cases} P_{sym}(col) = P_{sb}(row) = n \\ P_{sb}(col) = P_{sym}(row) = 1 \end{cases}$$

One should notice that $P_{sb}(col) = P_{sym}(row)$ can be further exploited.

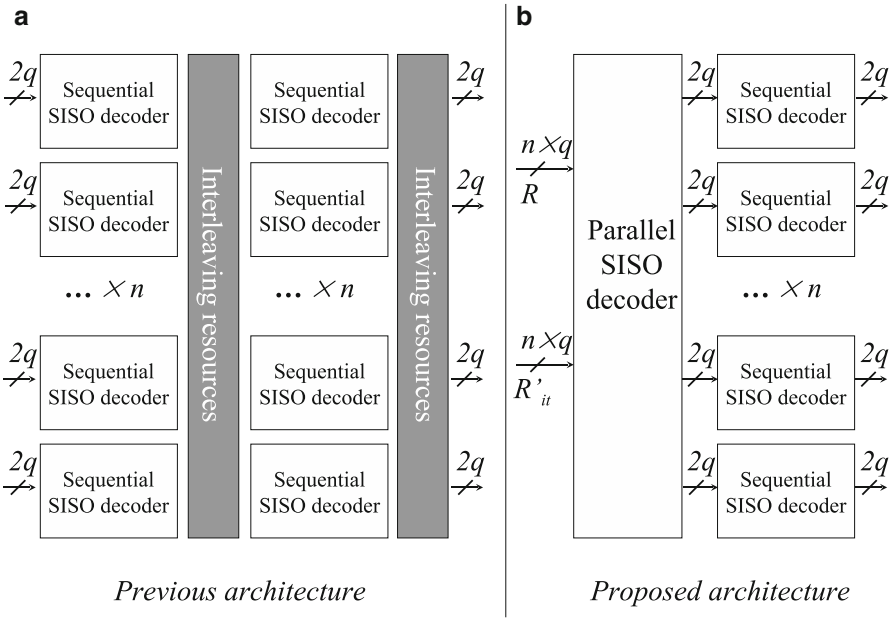

Fig. 4.10 Previous TPC decoder architecture (**a**) and proposed fully parallel SISO-based TPC decoder architecture (**b**)

4.5.2 Toward a Maximal Parallelism Rate

Starting from the IM-free architecture presented in the previous section, parallelism can be further enhanced. Figure 4.11 shows the alternate product code matrix parallel decoding scheme in which $P_{sb}(col) = P_{sym}(row) = m$ and $P_{sym}(col) = P_{sb}(row) = n$. The TPC decoder consists in $m\times$ n-decoders for column decoding and $n\times$ m-decoders for row decoding. An m-decoder can process m symbols in one clock period with $1 \leq m \leq n$. In such an architecture, the maximum reachable parallelism rate $P = n^2$ can be achieved by using n fully parallel SISO decoders for column decoding and n fully parallel SISO decoders for row decoding. Intra-symbol parallelism can also be exploited to increase the total parallelism to $P = P_{sb} \times P_{sym} \times P_{is} = n^2 \log(n)$. However, all these new architectural solutions require to design a SISO decoder able to process n symbols in one clock period.

4.6 Architecture of a Fully Parallel Combinational SISO Decoder

The proposed IM-free TPC decoder architecture requires a fully parallel combinational SISO decoder. To the best of our knowledge, only sequential SISO decoders able to process $m \leq n$ symbols in one clock period have been previously

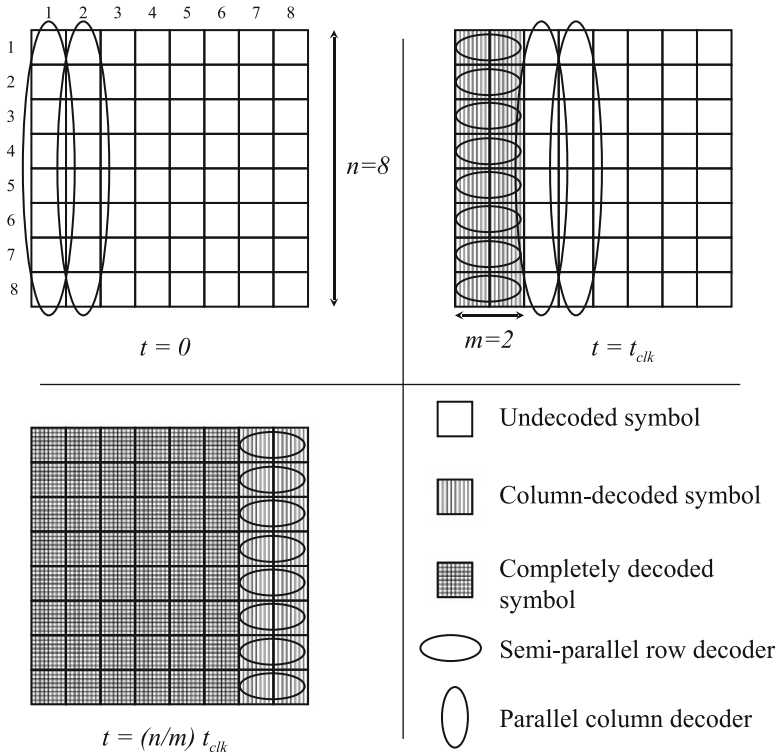

Fig. 4.11 Alternative turbo decoding scheduling for enhanced parallelism rate

designed. The design of a fully parallel combinatorial SISO decoder is a challenging issue. In the following section, such an architecture is described.

4.6.1 Algorithmic Parameter Reduction

As explained earlier in Sect. 4.2, the Chase–Pyndiah algorithm includes parameters (L, τ_p, Cw, q) which impact on both the performance and the complexity of the turbo decoding. BER simulations were performed with different parameters: $L = \{2;3;4;5\}$, $\tau_p = \{4;8;16\}$, $Cw = \{0;1;2;3\}$, $q = \{3;4;5\}$. Performing eight iterations, the parameter set $\mathscr{P}_0 = \{L = 5, \tau_p = 16, Cw = 3, q = 5\}$ gives the best BER performance for a high complexity [5]. However, algorithmic simulations showed that the reduced parameter set $\mathscr{P}_1 = \{L = 3, \tau_p = 8, Cw = 0, q = 5\}$ only induce a performance loss of 0.25 dB at BER$= 10^{-6}$ while it becomes null below BER$= 10^{-9}$. Further reducing these parameters would induce a notable performance loss. For example by simply reducing the number of test patterns:

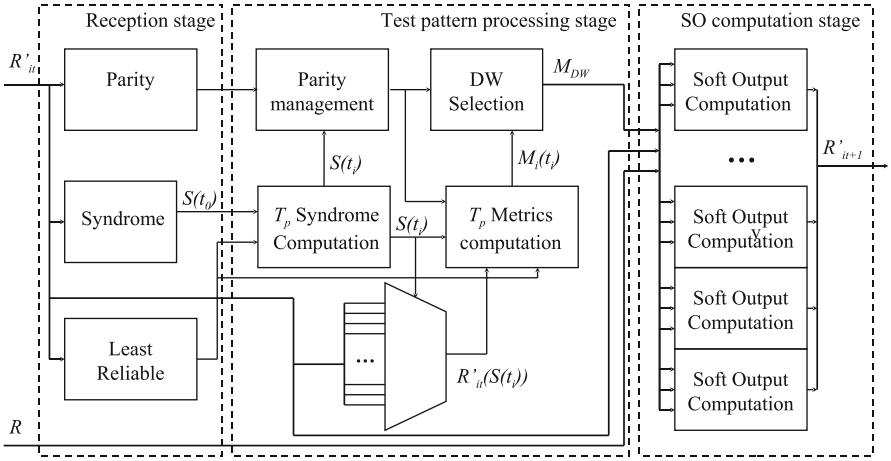

Fig. 4.12 Combinatorial version of the fully parallel SISO decoder

$\mathscr{P}_2 = \{L = 2, \tau_p = 4, Cw = 3, q = 5\}$, the performance loss reaches 0.5 dB. Consequently, using $\mathscr{P}_1$ enables the architecture to be simplified at very low performance lost below BER=10^{-9}.

4.6.2 Fully Parallel SISO Decoder Architecture

Figure 4.12 depicts the architecture of the fully parallel SISO decoder. In the first attempt a purely combinational architecture was designed. Later, a critical path study mandated the insertion of pipeline stages within the structure. The SISO decoder is split into three stages, namely the reception stage, the test pattern processing stage, and the soft-output computation stage.

4.6.2.1 Reception Stage

The reception stage corresponds to steps (1–3) of the Chase–Pyndiah algorithm detailed in Sect. 4.2. The syndrome of the incoming vector R'_{it} can be derived as $S(R'_{it}) = H \times sign(R'_{it})$ where H is the parity check matrix of the BCH code. A straightforward implementation of such a matrix multiplication is depicted on Fig. 4.13. The H matrix, the corresponding parity check equations, and the syndrome $S(t_0) = [s_2, s_1, s_0]$ implementation of a BCH(7,4) code are detailed.

It can be noticed that some parity check equations have similar terms. For instance, the term $(x_1 \oplus x_0)$ is used in both s_1 and s_2 computation. This means that a reuse of computation resources for an even more efficient implementation is possible. The parity of the incoming vector R'_{it} is computed with a similar structure

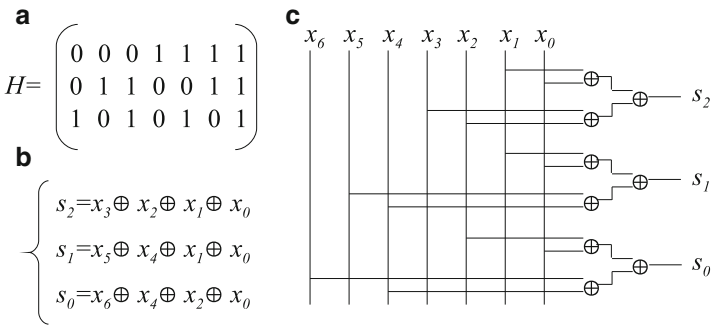

Fig. 4.13 BCH(7,4) code: (**a**) Parity check matrix. (**b**) Parity check equations. (**c**) Syndrome parallel computation implementation

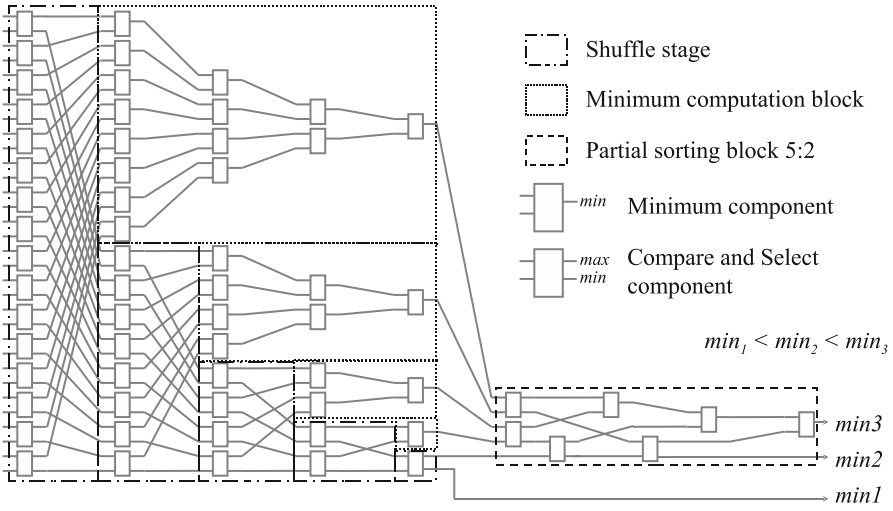

Fig. 4.14 Sorting network for least reliable bits selection

by "xoring" $(n-1)$ incoming bits. Selecting the least reliable bits among the incoming vector in parallel requires a sorting network. Such structures are composed of interconnected Compare and Select operators (CS). The interconnection scheme depends on the considered sorting algorithm. Many parallel sorting algorithm are conceivable [18]. However, most of them are optimized for a complete sorting, while the Chase–Pyndiah algorithm only requires a partial sorting (i.e., extracting L minima). Consequently we devised a network optimized, in terms of area and critical path, for the partial sorting of L=3 values among n=32, as depicted in Fig. 4.14. The structure is based on shuffle networks coupled with local minima computation blocks. After the first shuffle stage, min_1 is in the lower section while the upper section can either contain min_2 or min_3 or no minimum. The same reasoning is

applied recursively. After five shuffle stages, the minimum is determined while five values can still be min_2 and min_3. A local sorting of five values enables the determination of min_2 and min_3 value. This partial sorting network requires 35 CS operators and 29 minimum elements. The critical path consists of nine comparison stages.

4.6.2.2 Test Pattern Processing Stage

The test pattern processing stage corresponds to steps (4–5) of the Chase–Pyndiah algorithm detailed in Sect. 4.2. Instead of being processed sequentially, test patterns are processed in parallel. The syndrome of each test pattern is computed by adding $S(t_0)$ with the position of the inverted reliable bits. The parity management block computes the parity of R'_{it+1} considering the parity of R'_{it} and the detection of an error which is the case when $S(t_i) \neq 0$. Metrics of each test pattern are then computed by adding the contribution of each inverted bit in the current test pattern (least reliable bits, syndrome corrected bits, and the new parity bit). The minimum metric is determined in the DW selection block. The structure is a simple minimum selection tree. The multiplexer selects $R'_{it}(S(t_i))$ in order to compute test pattern metrics.

4.6.2.3 Soft-Output Computation Stage

The last stage is a duplication of n soft-output computation blocks. As shown in Fig. 4.15, this block first computes the new reliability F_{it} of each symbol. Since no competitor word is considered, the β value is automatically assigned. The β value is based on an estimation of the competitor word metric value. It is calculated from the reliability of the corrected bit and the least reliable bits. Then, the extrinsic information is computed and damped by the coefficient α_{it} which is devised to be a power of 2 making the multiplication a simple bit shifting. Finally, the channel information is added to generate the soft output R'_{it+1}. Within this block, all computation are performed in sign and magnitude format. Other arithmetic format were explored but the chosen one requires less computation resources than others.

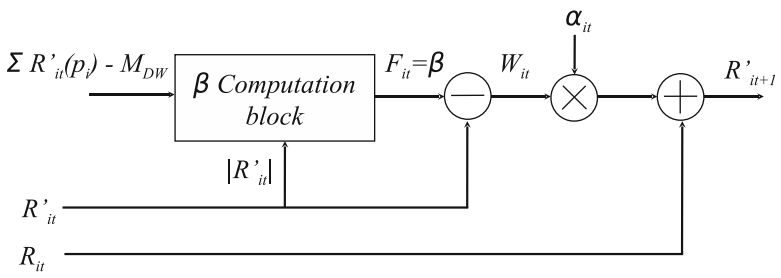

Fig. 4.15 Soft-output computation stage

4.7 Comparison with Existing TPC Decoders

4.7.1 Logic Synthesis Results of a BCH(32,26) SISO Decoder

In Sect. 4.4, we demonstrated that exploiting symbol parallelism is efficient if $A_{DEC}(P_{sym} = p) < p \times A_{DEC}(P_{sym} = 1)$. In order to verify this inequality, we compare one parallel ($P_{sym} = n$) BCH SISO decoder *vs* $n\times$ sequential ($P_{sym} = 1$) SISO decoders. Five versions of the BCH(32,26) parallel SISO decoder that have from one to five pipeline stages were designed. The one-pipeline stage version is a fully combinational architecture with register banks only at the input and output stages. Table 4.3 summarizes logic synthesis results of the five different parallel SISO decoders and compares them with $n = 32$ duplicated sequential SISO decoders. s is the number of pipeline stages inserted in the SISO decoder, $f_{\max}$ is the maximum frequency reached during logic syntheses. The throughput T is calculated such that $T = P \times f_{max}$, A represents the area of the design in equivalent gate count, and E is the efficiency: $E = \frac{T}{A}$. Logic syntheses were performed using Synopsys Design Compiler with an ST-microelectronics 90 nm CMOS process. The area is transposed in logic gate count. One equivalent logic gate corresponds to the area of a two-input NAND gate. It enables a more technology-independent measure of the hardware complexity.

As expected, the maximum frequency of the combinational decoder ($s = 1$) is lower than a sequential version. However, by inserting pipeline stages inside the combinatorial structure, an equivalent frequency is reached with $s = 5$. For this last version, the throughput is even higher than n sequential SISO decoders. The hardware cost of the pipeline stages insertion depends on registers location in the decoder architecture. This is the reason why $A(s = 4) < A(s = 3)$. In this particular case, having $s = 4$ pipeline stages enables register stages to be assigned at regular intervals, for a lower hardware cost. In terms of efficiency, a parallel SISO decoder can reach the same throughput as n sequential SISO decoders with a six times lower complexity. The efficiency gain increases with s.

These synthesis results demonstrate the higher efficiency of parallel SISO decoding for the code BCH(32,26). Now, if one considers larger code with the

Table 4.3 Comparison of parallel and sequential BCH(32,26) SISO decoder performance

	Parallel SISO decoder ($P_{sym} = 32$)					32 sequential SISO decoders ($P_{sym} = 1$)
s	1	2	3	4	5	3
$f_{\max}$ (MHz)	125	333	500	500	714	700
T (Gb/s)	4.0	10.7	16.0	16.0	22.9	22.4
A (Kgates)	18	26	31	26	34	200
E (Mb/s/gate)	0.15	0.27	0.34	0.41	0.44	0.07
G_E	2.1	3.9	4.9	5.9	6.3	1

same correction power (i.e., BCH(64,57), BCH(128,120)), the complexity of the reception stage and the soft-output computation stage would grow linearly with the code size n. However the complexity of the test pattern processing stage would only increase linearly with $p < n$. Consequently, the overall complexity of the parallel SISO decoder is lower than a duplication of n sequential SISO decoders. It confirms that a fully parallel SISO decoder enables a better reuse of computation and memory resources and makes the whole TPC decoder more efficient.

One should notice that, for higher correction power ($t > 1$), the algebraic decoding requires more complex algorithms such as Berlekamp–Massey algorithm [19, 20] which make the decoder complexity significantly higher. This is the reason why $t = 1$ BCH codes were selected is this study.

4.7.2 Comparison with Existing TPC Decoder Architectures

Table 4.4 compares hardware performance of existing TPC decoders architectures in an ultra-high-throughput context ($T > 10\,\text{Gb/s}$). For each architectural solution, the decoder main features, the targeted code, the levels of parallelism that were used in order to reach $T = 10\,\text{Gb/s}$, the resulting total parallelism ($P_{total} = \prod_i P_i$), the maximum number of iteration it_{max} are given. We consider that one iteration is actually implemented. The resulting throughput is $T = P_{total} \times f_{max}/it_{max}$. Finally, the gate count (A), the efficiency ($E = T/A$), and the achieved coding gain at BER=10^{-9} are given. Such a low BER is usually targeted in very high speed application (e.g., data transmission over Passive Optical Networks).

For a fair comparison, architectures described in [7, 13, 14, 16] were synthesized with the same technology: ST Microelectronics, CMOS 90 nm with a clock frequency $f_{max} = 500\,\text{MHz}$. For the remaining architectures, we gathered information from the published papers and technical reports.

Two versions of the P_{sym}-based TPC decoder were synthesized. The first one consists in four parallel SISO decoders together with 32 $P_{sym} = 4$-SISO decoders. The reached throughput is then sufficient for 10 Gb/s applications. The second version uses only fully parallel SISO decoder, 32 of such decoders are duplicated for each half-iteration. The maximum throughput is 85 Gb/s for the best efficiency. This architecture uses row-wise P_{sb} and column-wise P_{sym}. The barrel-shifter-based solution [13] can achieve 10 Gb/s with 2.6 Mgates. In order to reach a sufficient parallelism level, it was necessary to use frame parallelism. The efficiency of this approach is six times lower than the P_{sym}-based TPC decoders. This low efficiency is mainly due to the use of interleaving memory.

For the same reason, the TPC decoder with multi-access data [16] has a low efficiency and also requires the use of frame parallelism to achieve 10 Gb/s.

In [14], the elimination of interleaving memories improves the efficiency but the maximum parallelism rate is limited by the code size n. This makes the use of frame parallelism mandatory in an ultra high speed context.

Table 4.4 Comparisons of state-of-the-art TPC decoders

Decoder features	Code	P_i	P_{total}	it_{max}	T (Gb/s)	Area (Mgates)	E (Kb/s/gate)	Coding gain (dB) @BER=10^{-9}
This work	BCH(32,26)^2	$P_{sym} = 4, P_{sb} = 32$	128	6	10.7	0.4	26.8	8.0
	BCH(32,26)^2	$P_{sym} = 32, P_{sb} = 32$	1024	6	85.3	2.0	42.7	8.0
Barrel shifter + IM [13]	BCH(32,26)^2	$P_{sb} = 32, P_{frame} = 4$	128	6	10.7	2.6	4.1	8.4
Omega network + no IM [14]	BCH(32,26)^2	$P_{sb} = 32, P_{frame} = 4$	128	6	10.7	1.6	6.7	8.4
	BCH(64,57)^2	$P_{sb} = 64, P_{frame} = 2$	128	6	10.7	2.0	5.4	8.6
	BCH(128,120)^2	$P_{sb} = 128$	128	6	10.7	2.7	4.0	8.7
Multi-data access IM [16]	BCH(32,26)^2	$P_{sym} = 8, P_{sb} = 8, P_{frame} = 2$	128	6	10.7	3.5	3.1	8.4
Omega network + no IM [7]	RS(15,13)^2	$P_{is} = 4, P_{sb} = 15, P_{frame} = 2$	120	6	10	0.3	33.3	8.4
	RS(31,29)^2	$P_{is} = 5, P_{sb} = 31$	155	6	12.9	0.8	16.1	8.4
	RS(63,61)^2	$P_{is} = 3, P_{sb} = 63$	378	6	15.8	1.3	12.1	7.5
Commercial RS decoder (ASICS ws)	RS(255,239)	$P_{is} = 8, P_{frame} = 4$	32	X	10.7	0.12	89	5.0
Omega network + no IM [7]	RS(31,29)^2	$P_{is} = 5, P_{sb} = 31$	155	1	35	0.4	95	5.2
Commercial TPC decoder (Mitsubishi)	BCH(144,128) xBCH(256,239)	?	?	4	10.0	18.0	0.6	10

The study in [7] shows that RS-TPC are a practical solution for 10 Gb/s transmission over optical networks. As we mentioned in Sect. 4.4, using RS codes enables the use of intra-symbol parallelism. With an omega-network-based architecture, this decoder also presents good efficiency gain for similar decoding performance. One should notice that the P_{sym}-based fully parallel architecture is applicable to RS decoding as well. We expect that the application of intra-symbol parallelism would further increase the overall efficiency of the TPC decoder. Moreover, when comparing a single iteration of RS-TPC decoding with a commercial RS(255,239) code decoder, one can observe that superior efficiency is achieved for slightly better decoding performance.

Mitsubishi proposed a TPC decoder for 10 Gb/s optical transmissions. The component code is a BCH(144,128)xBCH(256,239). These codes are more powerful than $t = 1$ BCH codes that are used in this study. However the implementation is very costly in terms of hardware complexity. Indeed, 18 Mgates are necessary to implement such a decoder, which makes the efficiency very small. This is the cost that has to be paid for a 2 dB extra coding gain provided by this TPC decoder.

Conclusion

TPC decoding is a realistic solution for next generation high throughput optical communications such as long-haul optical transmissions or passive optical networks. The structure of the product codes makes them very suitable for parallelization. However the exploitation of some parallelism levels may not be efficient in terms of throughput/complexity ratio. This is particularly true when interleaving memory has to be duplicated.

In this chapter, we review and characterize all parallelism levels in TPC decoding. This analysis helps to better understand and classify existing TPC decoders. In these TPC decoders, high throughput architecture complexity is made prohibitive by the amount of memory usually required for data interleaving and pipelining.

After this design space exploration, we focus on an architecture that jointly exploits sub-block parallelism and symbol parallelism. This structure enables any interleaving resource to be removed. This TPC decoder requires a fully parallel SISO decoder capable of processing n symbols in one clock period. Such a SISO decoder architecture is described and includes an optimized parallel sorting network.

ASIC-based logic syntheses confirm the better efficiency of the IM-free TPC decoder architecture compared to others. Actually, when compared to other works, the area is reduced while the same throughput is achieved. A BCH(32,26)^2 product code can be decoded at 33.7 Gb/s with an estimated silicon area of 10 µm^2 in 65 nm CMOS technology.

References

1. Akita M, Fujita H, Mizuochi T, Kubo K, Yoshida H, Kuno K, Kurahashi S (2002) Third generation fec employing turbo product code for long-haul dwdm transmission systems. In: Optical fiber communication conference and exhibit, 2002 (OFC 2002), 17–22 March 2002, pp 289–290
2. Pyndiah R, Glavieux A, Picart A, Jacq S (1994) Near optimum decoding of product codes. In: IEEE Global Telecommunications Conference, 1994 (GLOBECOM '94)
3. Elias P (1954) Error-free coding. IEEE Trans Inf Theory 4(4):29–37
4. Forney GJ (1966) Generalized minimum distance decoding. IEEE Trans Inf Theory IT-12:125–131
5. Adde P, Pyndiah R, Raoul O (1996) Performance and complexity of block turbo decoder circuits. In: Proceedings of the third IEEE international conference on electronics, circuits, and systems, 1996 (ICECS '96), vol 1, 13–16 October 1996, pp 172–175
6. IEEE Standard for Local and Metropolitan Area networks (2001) Part 16: Air interface for fixed broadband wireless access systems, December 2001
7. Bidan RL, Leroux C, Jego C, Adde P, Pyndiah R (2008) Reed-solomon turbo product codes for optical communications: from code optimization to decoder design. EURASIP J Wirel Commun Netw 2008:909–912
8. Leroux C, Jego C, Adde P, Jezequel M (2008) On the higher efficiency of parallel reed-solomon turbo-decoding. In: ICECS'08: 15th international conference on electronics, circuits and system, 31st August - 3rd September, 2008
9. Chase D (1972) A class of algorithms for decoding block codes with channel measurement information. IEEE Trans Inf Theory IT:170–182
10. Leroux C, Jego C, Adde P, Jezequel M (2007) Towards Gb/s turbo decoding of product code onto an FPGA device. In: IEEE international symposium on circuits and systems, 2007 (ISCAS 2007), 27–30 May 2007, pp 909–912
11. Adde P, Pyndiah R (2000) Recent simplifications and improvements in block turbo codes. In: 2nd international symposium on turbo codes & related topics, Brest, France, 4–7 September 2000, pp 133–136
12. Leroux C, et al. (2011) Turbo product code decoder without interleaving resource: from parallelism exploration to high efficiency architecture. J Signal Process Syst 64(1):17–29
13. Chi Z, Parhi K (2002) High speed VLSI architecture design for block turbo decoder. In: IEEE international symposium on circuits and systems, 2002 (ISCAS 2002), vol 1, 26–29 May 2002, pp I-901–I-904
14. Jego C, Adde P, Leroux C (2006) Full-parallel architecture for turbo decoding of product codes. In: Electronics letters, vol 42, 31 August 2006, pp 55–56
15. Leroux C, et al. (2009) High-throughput block turbo decoding: from full-parallel architecture to FPGA prototyping. J Signal Process Syst 57(3):349–361
16. Cuevas J, Adde P, Kerouedan S, Pyndiah R (2002) New architecture for high data rate turbo decoding of product codes. In: IEEE global telecommunications conference, 2002 (GLOBECOM '02), vol 2, 17–21 November 2002, pp 1363–1367
17. Piriou E, Jego C, Adde P, Le Bidan R, Jezequel M (2006) Efficient architecture for Reed Solomon block turbo code. In: Proceedings of the IEEE international symposium on circuits and systems, 2006 (ISCAS 2006), 21–24 May 2006, 4 pp
18. Akl SG (1985) Parallel sorting algorithms. Academic, New York
19. Berlekamp ER (1968) Algebraic coding theory. vol 111, New York, McGraw-Hill
20. Massey JL (1969) Shift-register synthesis and bch decoding. IEEE Trans Inf Theory IT:122–127

Chapter 5
VLSI Implementations of Sphere Detectors

Johanna Ketonen, Markus Myllylä, Yang Sun, and Joseph R. Cavallaro

5.1 Soft Detection

The multiple input multiple output (MIMO) detection problem of an uncoded system can be considered as a so-called integer least squares problem, which can be solved optimally with a hard-output maximum likelihood (ML) detector [1]. The ML detector solves optimally the so-called closest lattice point problem by calculating the Euclidean distances (EDs) between the received signal vector and points in the lattice formed by the channel matrix and the received signal, and selects the lattice point that minimizes the Euclidean distance to the received vector [2]. The ML detection problem can be solved with an exhaustive search, i.e., checking all the possible symbol vectors and selecting the closest point. The ML detector achieves a full spatial diversity with regard to the number of receive antennas; however, it is computationally very complex and not feasible as the set of possible points increases.

The received frequency domain (FD) signal can be described with the equation $\mathbf{y} = \mathbf{H}\mathbf{x} + \eta$, where $\mathbf{x} \in \mathbb{C}^N$ is the transmitted symbol vector, $\eta \in \mathbb{C}^M$ is a vector containing circularly symmetric complex Gaussian distributed noise with variance σ^2, $\mathbf{H} \in \mathbb{C}^{M \times N}$ is the frequency domain channel matrix containing complex Gaussian fading coefficients, and N is the number of transmit (TX) antennas and M is the number of receive (RX) antennas. The entries of $\mathbf{x}$ are chosen independently

J. Ketonen
Department of Communications Engineering, University of Oulu, Oulu, Finland

M. Myllylä
Nokia Networks, Nokia, Oulu, Finland

Y. Sun • J.R. Cavallaro (✉)
ECE Department, Rice University, 6100 Main St., Houston, TX 77005, USA
e-mail: cavallar@rice.edu

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_5

from a complex QAM constellation Ω with sets of Q transmitted coded binary information bits $\mathbf{b} = [b_1, \ldots, b_Q]^{\mathrm{T}}$ per symbol.

The ML detector calculates the Euclidean distances (EDs) between the received signal vector $\mathbf{y}$ and lattice points $\mathbf{Hx}$, and returns the vector $\mathbf{x}$ with the smallest distance, i.e., it minimizes

$$\hat{\mathbf{x}}_{\mathrm{ML}} = \arg \min_{x \in \Omega^N} ||\mathbf{y} - \mathbf{Hx}||^2, \tag{5.1}$$

where $\mathbf{x}$ is the transmitted signal vector and $\mathbf{H}$ is the channel matrix. The ML detector performs an exhaustive search over all possible lattice points and the complexity is exponential in N.

The ML detector is optimal for uncoded systems, but for coded systems a posteriori probabilities (APP) for the decoder are required. Practical communication systems apply forward error correction (FEC) coding in order to achieve near capacity performance. The optimal way to process the spatially multiplexed and FEC coded data sequence would be to use a joint detector and decoder for the whole coded data sequence and decode the most probable data sequence. The complexity of the optimal receiver would be prohibitive as it depends on the length of the code block [3]. The optimal receiver is then approximated with an iterative receiver [4] with a separate soft-input soft-output (SfISfO) detector and soft in soft out (SISO) decoder, which exchange reliability information between the units. A structure of such a receiver is presented in Fig. 5.1.

The MAP detector provides the optimal APPs or log-likelihood ratios (LLR) [5] for the decoder. Given the interleaving of $\mathbf{b}$ and assuming the noise in the system is white Gaussian and the bits are approximately statistically independent, the a posteriori LLR for the transmitted bit k can be written as [3]

$$\begin{aligned} L_D(b_k|\mathbf{y}) &= \ln \frac{\Pr(b_k = +1|\mathbf{y})}{\Pr(b_k = -1|\mathbf{y})} \\ &= L_A(b_k) + \ln \frac{\sum\limits_{\mathbf{b} \in \mathcal{L}_{k,+1}} \exp(-\frac{1}{2\sigma^2}||\mathbf{y} - \mathbf{Hx}||^2 + \frac{1}{2}\mathbf{b}_{[k]}^T \mathbf{L}_{A,[k]})}{\sum\limits_{\mathbf{b} \in \mathcal{L}_{k,-1}} \exp(-\frac{1}{2\sigma^2}||\mathbf{y} - \mathbf{Hx}||^2 + \frac{1}{2}\mathbf{b}_{[k]}^T \mathbf{L}_{A,[k]})}, \end{aligned} \tag{5.2}$$

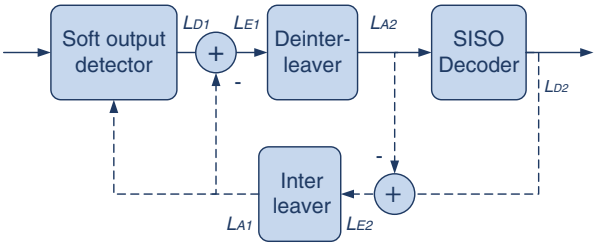

Fig. 5.1 The iterative receiver

where $\mathcal{L}_{k,+1} \bigcap \mathbb{B}_{k,+1}$ is a list of candidate points $\mathbf{x}$. $\mathbb{B}_{k,a}$ is the set of 2^{NQ-1} bit vectors having $b_k = a, a \in \{-1, 1\}$, $\mathbf{b}_{[k]}$ is a subvector of $\mathbf{b}$ without b_k, and vector $\mathbf{L}_{A,[k]}$ includes all L_A values except for b_k. The list $\mathcal{L}$ can be obtained by neglecting the insignificant elements in $\mathbb{B}$ such that the K candidate points in $\mathcal{L}$ include $\hat{\mathbf{x}}_{\mathrm{ML}}$ and $2^{MQ} > K \geq 1$ [3]. This can be achieved for example with a list sphere detector (LSD).

The approximation of the logarithm in (5.2) can be calculated using a small look-up table and the Jacobian logarithm [6]

$$\mathrm{jacln}(a_1, a_2) := \ln(e^{a_1} + e^{a_2}) = \max(a_1, a_2) + \ln(1 + \exp(-|a_1 - a_2|)). \quad (5.3)$$

The Jacobian logarithm in (5.3) can be computed without the logarithm or exponential functions by storing $r(|a_1 - a_2|)$ in a look-up table, where $r(\cdot)$ is a refinement of the approximation $\max(a_1, a_2)$. Max-log approximation further simplifies (5.2) when the refinement term is left out with negligible loss in performance. With these simplifications, $L_D(b_k|\mathbf{y}) - L_A(b_k)$ can be written as

$$\begin{aligned} L_E(b_k|\mathbf{y}) = & \max_{b \in \mathcal{L}_{k,+1}} \left\{ -\frac{1}{2\sigma^2} ||\mathbf{y} - \mathbf{H}\mathbf{x}||^2 + \frac{1}{2} \mathbf{b}_{[k]}^T \mathbf{L}_{A,[k]} \right\} \\ & - \max_{b \in \mathcal{L}_{k,-1}} \left\{ \frac{1}{2\sigma^2} ||\mathbf{y} - \mathbf{H}\mathbf{x}||^2 + \frac{1}{2} \mathbf{b}_{[k]}^T \mathbf{L}_{A,[k]} \right\}. \end{aligned} \quad (5.4)$$

5.1.1 Tree Search Algorithms

The tree search algorithms can be used to solve or approximate the hard output ML solution with reduced complexity compared to the full-complexity ML detector. They are based on preprocessing and tree search algorithms and their application to the MIMO detection problem has gained renewed attention in the literature during the last few decades [7]. The search over the lattice points can be performed with a tree structure due to the QR decomposition applied on the channel matrix. The tree search algorithm then aims to find the shortest path in a search tree formed by the MIMO channel matrix and the transmitted symbols, i.e., solves the exact ML solution or suboptimal solution depending on the algorithm search strategy. The algorithms in the literature are often divided into three categories according to the search strategy: the breadth-first (BF) search, the depth-first search (DF), and the metric-first (MF) search [8–10].

A class of algorithms, generally called sphere detectors (SD) [11–15], solve the ML solution with a reduced number of considered candidate symbol vectors. They take into account only the lattice points that are inside a sphere of a given radius. The condition that the lattice point lies inside the sphere can be written as

$$||\mathbf{y} - \mathbf{H}\mathbf{x}||^2 \leq C_0. \quad (5.5)$$

After QR decomposition of the channel matrix $\mathbf{H}$ in (5.5), it can be rewritten as $||\mathbf{y}' - \mathbf{R}\mathbf{x}||^2 \leq C_0'$, where $C_0' = C_0 - ||(\mathbf{Q}')^{\mathrm{H}}\mathbf{y}||^2$, $\mathbf{y}' = \mathbf{Q}^{\mathrm{H}}\mathbf{y}$, $\mathbf{R} \in \mathbb{C}^{N\times N}$ is an upper triangular matrix with positive diagonal elements, $\mathbf{Q} \in \mathbb{C}^{M\times N}$ and $\mathbf{Q}' \in \mathbb{C}^{M\times(M-N)}$ are orthogonal matrices.

The squared partial Euclidean distance (PED) of $\mathbf{x}_i^N$, i.e., the square of the distance between the partial candidate symbol vector and the partial received vector, can be calculated as

$$d(\mathbf{x}_i^N) = \sum_{j=i}^{N} \left| y_j' - \sum_{l=j}^{N} r_{j,l} x_l \right|^2, \tag{5.6}$$

where $i = N \ldots, 1$, y_j' is the jth element of $\mathbf{y}'$, $r_{j,l}$ is the j,lth element of the matrix $\mathbf{R}$, x_l is the lth element of the candidate vector $\mathbf{x}_i^N$, and $\mathbf{x}_i^N$ denotes the last $N-i+1$ components of vector $\mathbf{x}$ [15].

Hard output sphere detectors may cause significant performance degradation when used in a system with FEC. However, there are methods proposed in the literature to modify hard output detectors to give soft reliability information of the transmitted bits as an output. A tree search algorithm can be used to obtain a list of candidates $\mathcal{L}$ and their Euclidean distances which are used to calculate the APPs L_D of the coded bits in $\mathbf{b}_p$. The size of the candidate list and the bounding of the tree search define the trade-off between complexity and the quality of the soft output information. List detector algorithms continue the tree search until a defined list is obtained. LSDs can be used to approximate the MAP detector and to provide soft outputs for the decoder [3]. The algorithms can often be derived from the sphere detector algorithms with minor modifications.

A tree search detector structure is presented in Fig. 5.2. The channel matrix $\mathbf{H}$ is first decomposed as $\mathbf{H} = \mathbf{QR}$ in the QR-decomposition block. The Euclidean distances between the received signal vector $\mathbf{y}$ and the possible transmitted symbol vectors are calculated in the tree search block. The candidate symbol list $\mathcal{L}$ from the tree search block is demapped to a binary form. The tree search algorithm can be any algorithm that produces a list of candidate symbols, for example the LSD. The LLRs are calculated from the list of Euclidean distances in the LLR block. Limiting the range of LLRs reduces the required list size [16].

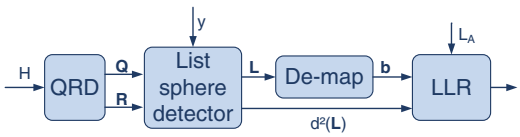

Fig. 5.2 The structure of the tree search detector

5.2 Breadth-First Detection

Breadth-first algorithms, such as the M algorithm [10] or the K-best Algorithm [17, 18] with sphere radius, extend the search in a layer-by-layer basis with multiple paths and always proceed in the depth direction of the tree. The algorithms always keep a constant number of candidate paths in each layer of the tree if no sphere radius constraint is introduced, but also require sorting of the candidate paths at each tree layer. The fixed complexity sphere decoder (FSD) [19] and the selective spanning with fast enumeration (SSFE) algorithm [20] also have a fixed complexity as they search over a fixed number of lattice points around the received signal. They both have a predefined number of nodes to be searched in the tree. Breadth-first algorithms guarantee a fixed number of visited nodes, which makes the algorithm very suitable for implementation. However, the breadth-first search strategy does not guarantee the ML solution and the search as such is inefficient in term of visited nodes especially with higher order modulation compared to the other tree search strategies.

5.2.1 K-Best Detection

The K-best algorithm [17] is a breadth-first search based algorithm, which keeps the K nodes which have the smallest accumulated Euclidean distances at each level. If the PED is larger than the squared sphere radius C_0, the corresponding node will not be expanded. The K-best algorithm without the sphere constraint can also be seen as the M-algorithm [10]. Here, $C_0 = \infty$, but a set the value for K is used instead, as is common with the K-best algorithms. The K-best LSD algorithm description is given as Algorithm 1. The main loop of the algorithm runs from $i = 1, \ldots, 2N$ in a real valued system, i.e., the real and complex parts of the signal are treated separately [15, 21].

The K-best tree search with no sphere constraint is illustrated in Fig. 5.3. A list size of two is assumed. The tree search proceeds level by level, expanding all the child nodes of each parent node. If the number of child nodes exceeds the list size,

Algorithm 1 The K-best LSD algorithm

Inputs: $\mathbf{Q}, \mathbf{R}, \mathbf{y}, C_0', K$, P (modulation used, P-QAM)
Preprocessing: Calculate $\mathbf{y}'$
Algorithm:
for $i = 1, \ldots, N$
1. Denote the partial candidate by $\mathbf{x}_{i+1}^N$.
1.1 Determine all admissible candidate child nodes x_i (with given C_0') and the corresponding PEDs $d(\mathbf{x}_i^N)$.
1.2 Store the partial candidates and their PEDs to a temporary stack memory.
2. Sort the partial candidates according to their PEDs
3. Store the K smallest PEDs and symbol vectors to the final list stack memory.
end
Give the candidates and their EDs as outputs.

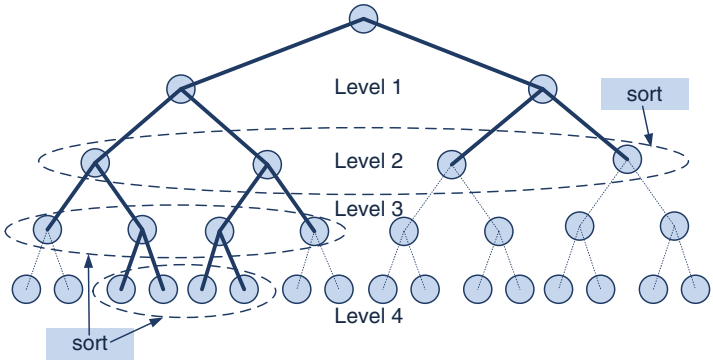

Fig. 5.3 The K-best tree search

sorting is performed to find the K nodes with the smallest PEDs. The tree search starts from the top of the tree on the first level in the figure. Both nodes are spanned, and on the second level, all the child nodes are spanned as well. Sorting is performed to find the two nodes with the smallest PEDs. The tree search continues until the fourth level is reached and the two leaf nodes with the smallest Euclidean distances are given as output. The breadth-first tree search can be modified to decrease the latency [22].

5.2.2 Selective Spanning with Fast Enumeration

The SSFE algorithm [20] can also be thought of as a breadth-first tree search algorithm. It can be also thought of as a fixed complexity detector. The algorithm spans each level of the tree based on the node spanning vector $\mathbf{m} = [m_1, \ldots, m_M]$. The number of spans for each node on a level is specified with the element of $\mathbf{m}$ corresponding to that level. As the spanned nodes are not discarded, the length of the final candidate list can be obtained by multiplying the elements of $\mathbf{m}$. For example, in a 2×2 antenna and 64-QAM system, the vector $\mathbf{m} = [64, 8]$ would lead to a final candidate list of 512. Here, a real valued system model is used. Such a system model simplifies the Euclidean distance calculation and the slicing operation as the closest constellation point selection can be done on a one dimensional axis.

The PED on each level i of the tree search can be calculated as

$$d_i(\mathbf{x}^i) = d_{i+1}(\mathbf{x}^{i+1}) + \left\| e_i(\mathbf{x}^i) \right\|^2, \tag{5.7}$$

where $d_{i+1}(\mathbf{x}^{i+1})$ is the PED from the previous level. The slicer unit selects a set of closest constellation points $\mathbf{x}^i$, minimizing

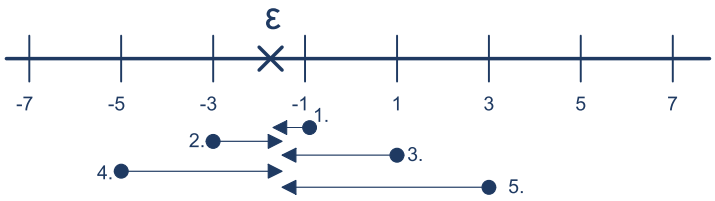

Fig. 5.4 The slicing operation in SSFE with 64-QAM

Algorithm 2 The SSFE algorithm

Inputs: $\mathbf{Q}, \mathbf{R}, \mathbf{y}$, $\mathbf{m}$, P (modulation used, P-QAM)
Preprocessing: Calculate $\mathbf{y}'$ and $\mathbf{h}_i = 1/\mathbf{R}_{i,i}$
Algorithm:
for $i = 1, \ldots, N$
1. Calculate ε for each candidate in $\mathbf{x}^i$
2. Slice the $\mathbf{m}_i$ closest points
3. Calculate the PEDs to the sliced lattice points
end
Give the candidates and their EDs as outputs.

$$\left\|e_i(\mathbf{x}^i)\right\|^2 = \left\| y_i' - \sum_{j=i+1}^{M} r_{i,j}x_j - r_{i,i}x_i \right\|^2. \tag{5.8}$$

Minimizing $\left\|e_i(\mathbf{x}^i)\right\|^2$ is equivalent to the minimization of $\|e_i(\mathbf{x}^i)/r_{ii}\|^2 = \|(y_i' - \sum_{j=i+1}^{M} r_{i,j}x_j)/r_{i,i} - x_i\|^2$, where $\varepsilon = (y_i' - \sum_{j=i+1}^{M} r_{i,j}x_j)/r_{i,i}$. The closest constellation points based on ε are selected in the slicer unit.

The real valued axis for 64-QAM is shown in Fig. 5.4. The slicing order given ε is also depicted. If five constellation points are sliced, the slicer would select the constellation points in the order of $\{-1, -3, 1, -5, 3\}$. The process is similar to the Schnorr–Euchner enumeration (SEE) [23]. The SSFE algorithm could then be thought of as the M-algorithm combined with SEE. The SSFE algorithm does not require sorting, which makes it more attractive for implementation than the M-algorithm or the K-best detector. The SSFE algorithm is summarized as Algorithm 2.

5.2.2.1 Implementation Choices

The top level architecture of the K-best LSD for a 2×2 antenna system is shown in Fig. 5.5. The 4×4 antenna system LSD is based on the same architecture, but four more PED calculation blocks and sorters are added to the design. The architecture for the SSFE has a similar pipelined structure, where each level of the tree is processed separately.

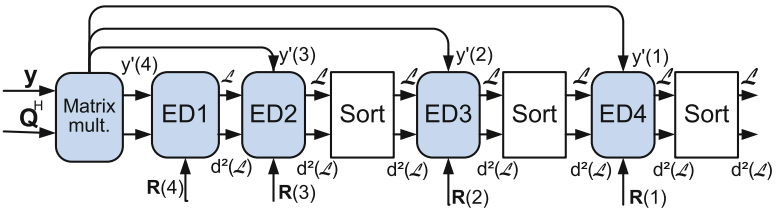

Fig. 5.5 The top level architecture of the 2×2 K-best LSD

The K-best LSD architecture is modified from [24]. A 2×2 and a 4×4 antenna system with a real signal model [25] is assumed. The received signal vector $\mathbf{y}$ is multiplied with matrix $\mathbf{Q}$ in the matrix multiplication block. Matrix $\mathbf{R}$ is multiplied with the possible transmitted symbols after the QRD is performed, i.e., when the channel realization changes. PEDs between the last symbol in vector $\mathbf{y}'$ and possible transmitted symbols are calculated in block PED1 in a 2×2 antenna system with $d(\mathbf{x}_4^2) = ||y_4' - r_{4,4}'||^2$. The resulting lists of symbols and PEDs are not sorted at the first stage. The distances are added to the PEDs $d(\mathbf{x}_3^2) = ||y_3' - (r_{3,3}' + r_{3,4}')||^2$ calculated in the PED2 block. The lists are sorted and K partial symbol vectors with the smallest PEDs are kept. PED3 block calculates $d(\mathbf{x}_2^2) = ||y_2' - (r_{2,2}' + r_{2,3}' + r_{2,4}')||^2$, which are added to the previous distance and sorted. The last PED block calculates the PEDs $d(\mathbf{x}_1^2) = ||y_1' - (r_{1,1}' + r_{1,2}' + r_{1,3}' + r_{1,4}')||^2$. After adding the previous distances to $d(\mathbf{x}_1^2)$, the lists are sorted and the final K symbol vectors are demapped to bit vectors and their Euclidean distance is used in the LLR calculation.

High level synthesis (HLS) was used to obtain the implementation results. Even though HLS tools have been developed for decades, only the tools developed in the last decade have gained a more widespread interest. The main reasons for this are the use of an input language, such as C, familiar to most designers, the good quality of results, and their focus on digital signal processing (DSP) [26]. HLS tools are especially interesting in the context of rapid prototyping where they can be used for architecture exploration and to produce designs with different parameters [27]. While the results may not always be as optimal as with hand-coded HDL, the tool allows experimenting with different architectures in a short amount of time. The complexity results can be close to the hand-coded ones with small designs [28]. There can be a bigger difference with large designs.

The implementation of algorithms was done by writing the architecture description with fixed-point ANSI C++ language and then applying the Catapult C Synthesis tool [29] to produce a register transfer level (RTL) description. After obtaining the RTL with the desired timing and complexity results, synthesis was performed with Synopsys Design Compiler specific tools to obtain the final complexity results. The algorithms in this section and in Sect. 5.3.4 were implemented with 0.18 µm complementary metal-oxide semiconductor (CMOS) ASIC technology for a 4×4 MIMO system with 16- and 64-QAM. The ASIC power estimation was

Table 5.1 Implementation results with 4×4 16-QAM

Receiver	Gates		Power		Detection rate
	Tree search	LLR	Tree search	LLR	
SSFE/SSFE 2 it.	135.2 k	19 k/34.6 k	488.9 mW	79 mW/158 mW	186 Mbps/163 Mbps
8-best/8-best 2 it.	97 k	17.3 k/33.1 k	341.5 mW	68.3 mW/140.5 mW	140 Mbps/126 Mbps
16-best	148.4 k	20.2 k	499.6 mW	79.2 mW	70 Mbps

Table 5.2 Implementation results with 4×4 64-QAM

Receiver	Gates		Power		Detection rate
	Tree search	LLR	Tree search	LLR	
SSFE/SSFE 2 it.	177.4 k	25.7 k/50.4 k	568.6 mW	110.5 mW/236.7 mW	269 Mbps/222 Mbps
8-best/8-best 2 it.	183.7 k	21.5 k/45.2 k	551.4 mW	87 mW/197.9 mW	210 Mbps/180 Mbps
16-best	217.2 k	24.5 k	717.3 mW	96.6 mW	105 Mbps

done with the Synopsys PrimeTime tool. Results for the enhanced tree search, other antenna configurations, adaptive systems, and other detectors can be found in [30, 31].

5.2.2.2 VLSI Implementation

The complexity and performance of two breadth-first tree search algorithms are compared. The complexity results for the SSFE and K-best detectors are presented in Table 5.1 for 16-QAM and in Table 5.2 for 64-QAM. Results for the LLR calculation are also given for a fair comparison of the two detectors. The detection rate of a receiver can be calculated as $\frac{QN}{D_{\text{rec}}}$, where Q is the number of bits per symbol, $D_{\text{rec}} = D_{\text{det}} + (D_{\text{LLR}} + D_{\text{dec}})N_{\text{iter}}$, D_{det} is the latency of the detector, D_{LLR} is the latency of LLR calculation, D_{dec} is the latency of the decoder, and N_{iter} is the number of iterations between the detector and the decoder. LLR calculation and decoding can be performed simultaneously and in a pipelined manner with detection and their latency does not have to be included in the throughput latency. In an iterative receiver, the throughput latency is determined by the minimum of D_{det} and $D_{\text{LLR}} + D_{\text{dec}}$. The receivers were designed to have a detection rate, which would be enough for the 3GPP Long Term Evolution (LTE) 20 MHz bandwidth.

The word lengths for the K-best LSD and LLR calculation are mainly 16 bits and computer simulations have been performed to confirm that there is no performance degradation [30]. The sorters are insertion sorters. The list size values of 16 and 8 are used in the implementation.

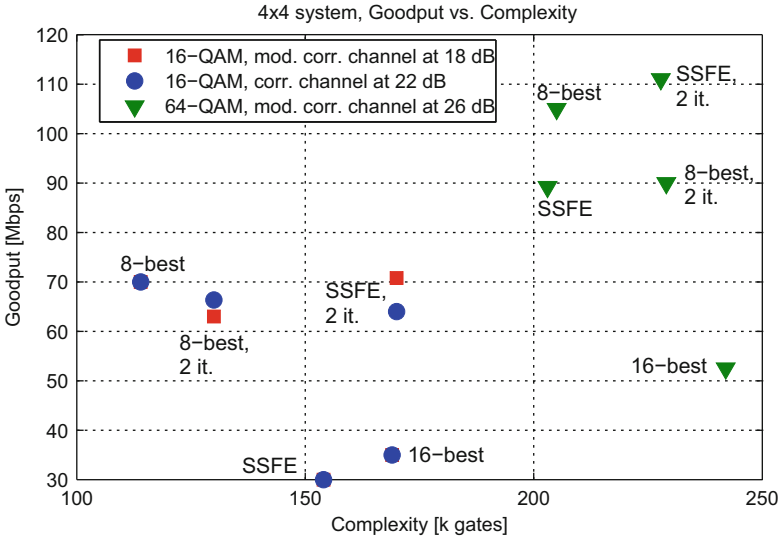

Fig. 5.6 Complexity-performance trade-off in a 4×4 antenna system

The SSFE list size is 12 and the node spanning vector is [3,2,2,1,1,1,1,1]. The clock frequency of the detectors was 280 MHz except for the 64-QAM SSFE where only a 269 MHz clock frequency was achieved. In the receiver with two global iterations, the tree search is performed only once and the complexity is the same as with one iteration. However, the LLR calculation is different in the two cases as the feedback from the decoder is used in the iterative detector. Decoding reduces the detection rate in the iterative receiver. The 8-best detector has a lower complexity and power consumption than the SSFE in the 16-QAM case, but the detection rate is also lower. The power consumption is also lower in the 64-QAM case, but the detection rate of the SSFE is higher.

The complexity-performance trade-off is illustrated in Fig. 5.6. The goodput, i.e., the minimum of the transmission throughput and hardware detection rate of information bits in a 20 MHz bandwidth with a 1/2 code rate, is compared to the hardware complexity. The figure then illustrates the communication performance of each detector compared to its complexity. The transmission throughput results were obtained with computer simulations in a realistic communication system model. The K-best with list size of 16 has a high complexity and low goodput. The goodput of the SSFE with two global iterations is close to that of the 8-best with one iteration with 16-QAM, but has a higher complexity. With 64-QAM, SSFE with two iterations achieves the highest goodput. Extra iterations do not bring any benefit with the K-best tree search as the detection rate is low. Even though the SSFE algorithm does not include sorting, the slicing operation induces extra complexity compared to the K-best algorithm and the difference between the two tree search algorithms remains small. The iterations between the detector and the decoder can

improve the communication performance but at the same they increase the latency and complexity, resulting in a low overall gain. However, some pipelining and parallelization techniques can be used to improve the throughput [32].

5.3 Depth-First and Metric-First Detection Algorithm Implementations

In this section, we introduce some examples of soft-output depth-first and metric-first search based detection algorithms and their VLSI implementations [31]. The considered soft-output sphere detection algorithms are first presented in Sect. 5.3.1. Implementation trade-offs are then presented in Sect. 5.3.2, and the architectural choices in Sect. 5.3.3. Finally, the implementation results are presented in 5.3.4.

5.3.1 Algorithm Descriptions

The considered depth-first and metric-first search based LSD algorithms are introduced in this section.

5.3.1.1 Depth-First Algorithm

Depth-first algorithms are based on a sequential search and go through a variable number of nodes in the search tree depending on the channel realization and the signal to noise ratio (SNR). The algorithms explore the tree along the depth until the cost metric of the path is below a defined threshold called a sphere radius. They then return and pursue another unexplored path. DF algorithms are able to find the exact ML solution if the search is not bounded. The Pohst enumeration method is often considered to be the original sphere algorithm [12]. The algorithm search complexity is bounded by selecting a constant sphere radius, which limits the search in the tree to the most likely paths. More advanced adaptive sphere radius was introduced as the Viterbo–Boutros (VB) implementation [14], and the SEE [23] can be seen as even more efficient modification of the Pohst enumeration and VB implementation [11].

We consider a depth-first search based sphere detection algorithm called the SEE—LSD and it is listed as Algorithm 3. It is an extension of SEE-SD [23] to a LSD, and the algorithm continues the search until all admissible nodes have been checked and the required candidate list $\mathcal{L}$ has been obtained. The output candidate list $\mathcal{L}$ includes the most probable candidates, i.e., the candidates with the lowest ED. The sequential algorithm initially starts from the root layer and extends the partial candidate $\mathbf{s} = \mathbf{x}_N^N$ with the best admissible node determined by the SE enumeration. The search tree pruning loop in the algorithm extends the considered

Algorithm 3 $[\mathcal{L}]$ = SEE-LSD($\mathbf{y}', \mathbf{R}, N_{\text{cand}}, \Omega, N$)

Initialize set $\mathcal{L}$, and set $C_0 = \infty$, $m = 0$, $n_1 = 1$, $i = N$
Initialize $\mathcal{N}(\mathbf{s} = \mathbf{x}_N^N, d(\mathbf{s}) = 0)$
WHILE ($i \neq N$ and $n_1 \neq |\Omega|$) {
  IF ($n_1 = |\Omega|$) { Set $i = i + 1$, determine n_1 and continue with $\mathcal{N}(\mathbf{s} = \mathbf{x}_{i+2}^N, d(\mathbf{s}))$ }
  ELSE
    Determine the n_1th best node x_i for $\mathbf{s}_c = (x_i, \mathbf{x}_{i+1}^N)$ and calculate $d(\mathbf{s}_c)$
    IF ($d(\mathbf{s}_c) < C_0$)
      IF ($\mathbf{s}_c$ is a leaf node, i.e., $i = 1$)
        1. Store $\mathcal{N}_F(\mathbf{s}_c, d(\mathbf{s}_c))$ in $\{\mathcal{L}\}^m$
        2. Set $m = m + 1$ or, if $\mathcal{L}$ is full, set m according to $\{\mathcal{L}\}^m$ with max ED and $C_0 = \{d(\mathbf{s})\}^m$
        3. Continue with $\mathcal{N}(\mathbf{s} = \mathbf{x}_{i+1}^N, d(\mathbf{s}))$, $n_1{+}{+}$ and $i = 1$ if $n_1 + 1 \leq |\Omega|$
      ELSEIF ($i \neq 1$ or $n_1 + 1 = |\Omega|$) { Set $i = i - 1$ and $n_1 = 1$, and continue with $\mathcal{N}(\mathbf{s}_c, d(\mathbf{s}_c))$ }
    ELSEIF ($d(\mathbf{s}) \geq C_0$ and $i \neq N - 1$) { Set $i = i + 1$, determine n_1 and continue with $\mathcal{N}(\mathbf{s} = \mathbf{x}_{i+2}^N, d(\mathbf{s}))$ }
    ELSE {End the algorithm} }

partial candidate $\mathbf{s} = \mathbf{x}_{i+1}^N$ with the next best available child node in each iteration until the PED of the extended partial candidate exceeds the sphere radius C_0 or a leaf node $\mathbf{s} = \mathbf{x}_1^N$ is found. In the case of a leaf node $\mathbf{s} = \mathbf{x}_1^N$, the candidate information $\mathcal{N}(\mathbf{s}, d(\mathbf{s}))$, which includes the candidate $\mathbf{s}$ and the corresponding ED $d(\mathbf{s})$, is added to the final candidate list $\mathcal{L}$ if the ED $d(\mathbf{s})$ is lower than the current sphere radius C_0. The radius is always updated to be equal to the highest ED in the final list when the final candidate list is full and a new leaf node is found. If the extended candidate exceeds the C_0 or all the admissible nodes have been checked, the algorithm moves one layer higher and continues with the next best admissible node. The next best admissible node is determined based on the previously extended nodes.

5.3.1.2 Metric-First Algorithm

Metric-first algorithms are based on a sequential search method and the search always proceeds along a path with the best cost metric among the stored paths in the tree search [8, 33]. MF algorithms are based on Dijktra's algorithm [34, 35], which was originally used to solve the single-source shortest path problem for a graph. The application of metric-first algorithms for MIMO detection has been applied in [36–38]. MF algorithms find the exact ML solution and the search strategy is efficient in terms of visited nodes in the search tree, but requires storing and ordering of the paths studied [33].

The increasing radius (IR)—LSD is listed as Algorithm 4. The IR-LSD algorithm uses the metric-first search strategy and it is a modification of Dijkstra's algorithm [34] to a LSD algorithm. The algorithm is optimal in the sense of the number of nodes in the tree structure visited [33, 34]. The output candidate list $\mathcal{L}$ includes the most probable candidates, i.e., the algorithm always gives exactly the same output

Algorithm 4 $[\mathcal{L}] = \text{IR-LSD}(\mathbf{y}', \mathbf{R}, N_{\text{cand}}, \Omega, N)$

Initialize sets $\mathcal{S}$ and $\mathcal{L}$, and set $C_0 = \infty$, $m = 0$, $n_1 = 1$
Initialize $\mathcal{N}(\mathbf{s} = \mathbf{x}_N^N, d(\mathbf{s}) = 0, n_2 = 2, i = N)$
WHILE $(C_0 < d(\mathbf{s}))$ {
1. Determine the n_1th best node x_i for $\mathbf{s}_c = (x_i, \mathbf{x}_{i+1}^N)^T$ and calculate $d(\mathbf{s}_c)$
2. Determine the n_2th best node x_{i+1} for father candidate
$\mathbf{s}_f = (x_{i+1}, \mathbf{x}_{i+2}^N)^T$ and calculate $d(\mathbf{s}_f)$ if $n_2 \leq |\Omega|$
IF $(d(\mathbf{s}_c) < C_0)$
IF ($\mathbf{s}_c$ is a leaf node, i.e., $i = 1$)
1. Store $\mathcal{N}_F(\mathbf{s}_c, d(\mathbf{s}_c))$ in $\{\mathcal{L}\}^m$
2. Set $m = m + 1$ or, if $\mathcal{L}$ is full, set m according to $\{\mathcal{L}\}^m$ with max ED and $C_0 = \{d(\mathbf{s})\}^m$
3. Continue with $\mathcal{N}(\mathbf{s} = \mathbf{x}_{i+1}^N, d(\mathbf{s}), n_1 + 1, 1)$ if $n_1 + 1 \leq |\Omega|$
ELSE { Store $\mathcal{N}_c(\mathbf{s}_c, d(\mathbf{s}_c), n_2 = 2, i - 1)$ in $\mathcal{S}$ }
IF ($\mathcal{N}_f$ calculated and $d(\mathbf{s}_f) < C_0$) { Store $\mathcal{N}_f(\mathbf{s}_f, d(\mathbf{s})_f, n_2 + 1, i)$ in $\mathcal{S}$ }
3. Continue with $\mathcal{N}$ with min PED from $\mathcal{S}$ and set $n_1 = 1$ }

as the SEE-LSD algorithm. The algorithm always extends the partial candidate with the lowest PED in one extend loop. The algorithm operates in a sequential fashion; it initially starts from the root layer with partial candidate $\mathbf{s} = \mathbf{x}_N^N$ and determines the next best admissible node x_i at layer i with SEE. The child candidate is then defined as $\mathbf{s}_c = (x_i, \mathbf{x}_{i+1}^N)^T$. The algorithm also, if possible, extends the father candidate $\mathbf{s}_f = \mathbf{x}_{i+2}^N$ with the next best admissible node x_{i+1}. The SEE, which is used to determine the next best admissible node, requires the information of already extended nodes, and the information is defined as n_1 and n_2 for the considered candidate and its father candidate, respectively. The algorithm uses two memory sets for storing the candidates, the final candidate set $\mathcal{L}$ and the partial candidate set $\mathcal{S}$. In the algorithm search, the partial child candidate information $\mathcal{N}_{\mathcal{S}}(\mathbf{s}_c, d(\mathbf{s}_c), n_1)$ and the possible father candidate information $\mathcal{N}_{\mathcal{S}}(\mathbf{s}_f, d(\mathbf{s}_f), n_2)$ are stored to set $\mathcal{S}$ after each tree pruning loop. In the case the child candidate $\mathbf{s}_c$ is a leaf node and smaller than the current radius C_0, the candidate information $\mathcal{N}_{\mathcal{L}}(\mathbf{s}_c, d(\mathbf{s}_c))$ is stored to the final list set $\mathcal{L}$. The sphere radius C_0 is updated when $\mathcal{L}$ is full and the candidate with the largest ED is replaced with a new leaf candidate. After storing the candidate(s), the algorithm finds the candidate information $\mathcal{N}_{\mathcal{S}}$ with the minimum PED $d(\mathbf{s})$ from set $\mathcal{S}$ and continues the algorithm if the PED is smaller than the current radius C_0. It should also be noted that $n_1 = 0$ is used in the tree pruning loop if the extended node is not a leaf node, and the n, which is read from $\mathcal{S}$, is used as n_2.

5.3.2 *Implementation Trade-Offs*

The algorithms presented in Sect. 5.3.1 are not as such directly feasible for hardware implementation without some modifications. In order to reserve the hardware resources for the algorithm to meet the given timing constraints, we need to determine the so-called worst case scenario and determine the algorithm complexity

accordingly. The SEE-LSD and the IR-LSD, however, visit a variable number of nodes depending on the channel realization, and the implementation of these algorithms as such is not feasible for a system with a fixed latency requirement. A simple way to fix the complexity is to limit the maximum number of L_{node} nodes visited by the LSD algorithm [31]. If the sphere search is not completed within the defined L_{node}, the algorithm is stopped and the current final candidate list $\mathcal{L}$ is given as an output. Another more sophisticated alternative is to use a scheduling algorithm as, e.g., in [31, 39, 40]. The idea behind the scheduling algorithm is that, e.g., in an OFDM system, higher maximum limits L_{node} can be allocated for subcarriers with a difficult channel realization while subcarriers with easier channel realization can be allocated with lower limits.

The LSD algorithms are often assumed to apply a real equivalent system model [15, 21] especially in the implementation of the algorithms. However, complex valued signal models are also applied in the literature [3, 39]. The definition of the signal model does not affect the mathematical equivalence of the expressions, but it affects the lattice definition where the LSD algorithm search is executed. The real signal model was shown to be clearly the better choice to be applied with LSD algorithms [31], and we also consider it here.

5.3.3 Architectural Choices

Architectural design choices for the considered LSD algorithms are presented in this section.

5.3.3.1 SEE-LSD

A scalable architecture for the SEE-LSD algorithm, which consists of a tree pruning unit (TPU), a control unit (CNTR), and a memory unit, is shown in Fig. 5.7. The architecture operates in sequential fashion and prunes a single node in the search tree in each iteration. The TPU executes the tree pruning, and the CNTR determines the partial candidate for the next iteration and the possible final candidate to be stored in the memory unit. The problem of variable complexity is solved by applying an input variable L_{node}, which sets a maximum limit for the number of nodes to be pruned by the architecture as discussed in Sect. 5.3.2.

The SEE-LSD algorithm TPU microarchitecture is illustrated in Fig. 5.8. The TPU microarchitecture is divided into two sub-units that can be implemented with different levels of parallelism and pipelining. It should be noted that the SEE-LSD algorithm TPU microarchitecture has to be able to calculate the tree pruning for partial candidates in different search layers. Typically the TPU should be made as fast as possible with the proper parallelism and pipelining configuration as the latency of the unit directly affects the throughput of the SEE-LSD algorithm architecture. The first unit calculates the part of PED calculation that is independent

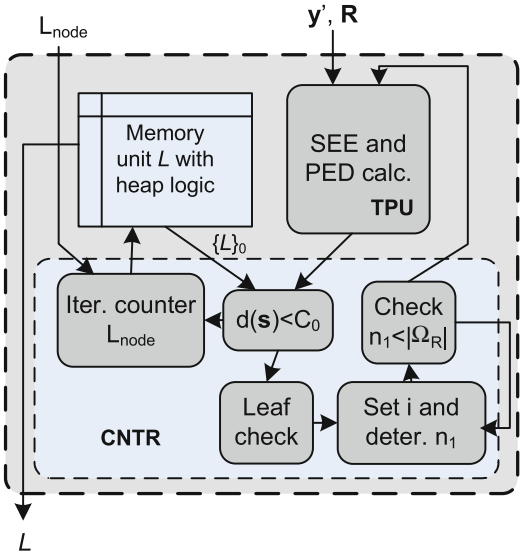

Fig. 5.7 A scalable architecture for the SEE-LSD algorithm

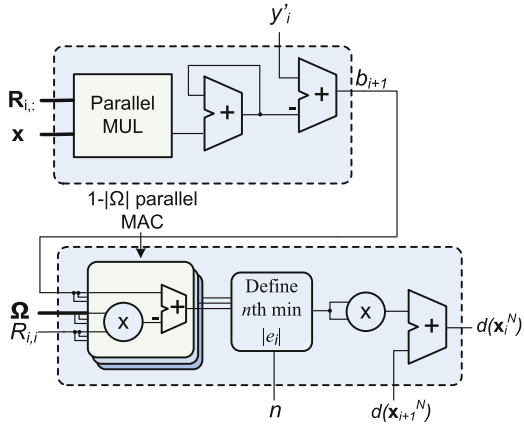

Fig. 5.8 The microarchitecture for the extension of the candidate

of the new symbol x_i in (5.6), where i is the current search layer. The second unit executes the SEE, i.e., determines the nth best node x_i, and calculates the PED of the extended partial candidate accordingly [31].

The memory unit is used to store the N_{cand} final candidates with the smallest EDs, which are found during the SEE-LSD algorithm tree search. The memory unit is designed as a binary heap [35,41] data structure, which keeps the stored elements in order according to the selected cost metric. The memory unit $\mathcal{L}$ is implemented as max-heap, where the new element $\mathcal{N}_{\text{F}}(\mathbf{s}_{\text{c}}, d(\mathbf{s}_{\text{c}}))$ is always ordered in the heap as it is stored. The heap elements are kept in order so that the final candidate with the maximum ED is always at the top of the heap [35,41].

The required control logic in CNTR unit for the SEE-LSD algorithm architecture is rather simple. The logic determines the next search level i and next admissible node n_1 for the next algorithm iteration based on the partial candidate, which was

extended in the TPU. If the extended candidate is a leaf node and $d(\mathbf{s}) < C_0$, the final candidate is stored to the memory unit and the sphere radius C_0 is possibly updated. The CNTR unit also terminates the search after L_{node} iterations.

The SEE-LSD algorithm architecture operates in sequential fashion a total of L_{node} iterations, where the parameter L_{node} should be selected as suitable to provide the desired performance. The latency of the algorithm iterations consists of the latency of the CNTR and the latency of the TPU or the memory unit. The TPU and memory unit operations are designed to be executed in parallel, where the TPU is typically the slower unit as it includes more operations and the memory unit is executed only seldom. In order to maximize the throughput of the SEE-LSD algorithm architecture, the TPU should be implemented with proper parallelism and pipelining. The parameter L_{node} can also be lowered to increase the throughput with the cost of decreased performance. The SEE-LSD algorithm architecture is as such scalable for system configurations with different number of transmit antennas N and different constellation Ω.

5.3.3.2 IR-LSD

The IR-LSD algorithm architecture is shown in Fig. 5.9 and includes a TPU with two calculation modules, a partial candidate memory unit, a final candidate memory unit, and a control logic (CNTR) unit. In each iteration, the TPU executes the tree pruning for two partial candidates, and the CNTR determines the partial candidate for the next iteration and the possible final candidate to be stored in the

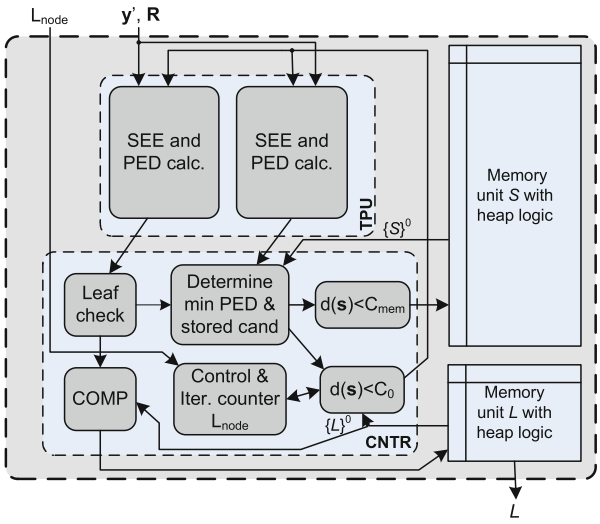

Fig. 5.9 A scalable architecture for the IR-LSD algorithm

memory unit. The problem of variable complexity is solved by applying an input variable L_{node}, which sets a maximum limit for the number of nodes to be pruned by the architecture. The IR-LSD algorithm architecture TPU is similar to the TPU in the SEE-LSD algorithm architecture with two similar candidate extension modules, which execute the tree pruning for the new selected candidate and the corresponding father candidate in parallel. The latency of the parallel units, i.e., the parallelism and pipelining choices, should be designed to be as similar as possible for efficient design.

There are two memory units in the IR-LSD architecture: the partial candidate memory set $\mathcal{S}$ and the final memory set $\mathcal{L}_{\text{F}}$. The memory units are designed as binary heap [35, 41] data structures, which keep the stored elements in order according to the selected cost metric. The partial candidate memory set $\mathcal{S}$ is implemented as min-heap, where the elements $\mathcal{N}(\mathbf{s}, d(\mathbf{s}), n_2, i)$ are ordered so that the candidate with the minimum PED is always sorted to be at the top of the heap. The final memory set $\mathcal{L}_{\text{F}}$, which is similar to the memory unit in the SEE-LSD architecture, is implemented as max-heap, where the stored final candidates $\mathcal{N}(\mathbf{s}, d(\mathbf{s}))$ are sorted according to the ED. The size of the partial candidate memory $\mathcal{S}$ is equal to L_{node} elements. In practice, ordering of the partial memory elements might become a limiting factor in the IR-LSD algorithm implementation with a large L_{node}. A technique called as memory sphere radius C_{mem} is applied to decrease the amount of memory access [31].

The control logic unit includes an iteration counter for the IR-LSD algorithm architecture and determines the candidates to be stored in the memory units and to be used in the search in the next algorithm iteration. The candidate to be used in the TPU unit in the next iteration is determined as the candidate with minimum PED from the extended candidates $\mathcal{N}_c$ and $\mathcal{N}_f$, and the minimum candidate in partial memory $\{\mathcal{S}\}^0$. If either one of the extended candidates $\mathcal{N}_c$ or $\mathcal{N}_f$ is selected for the next algorithm iteration, $\{\mathcal{S}\}^0$ remains in the memory. Thus, unnecessary memory access is minimized as the candidates $\mathcal{N}_c$ and $\mathcal{N}_f$ are not directly stored to the memory. The extended partial candidate(s) to be stored in $\mathcal{S}$ are also conditioned with C_{mem} to minimize memory access. If the extended candidate $\mathcal{N}_c$ is a leaf node and $d(\mathbf{s}) < C_0$, the final candidate is stored to the memory unit and the sphere radius C_0 is possibly updated.

The IR-LSD algorithm architecture and its timing are designed to minimize the latency in one algorithm iteration by introducing parallel operations. The straightforward data flow mapping of algorithm would first extend the new candidates, then store them in memory units, and finally determine the new candidate for the next iteration. However, the data flow can be designed more efficiently to reduce the latency of one algorithm iteration. After the TPU extends the partial candidates in the current iteration, the control logic unit determines the new candidate for the TPU at the next iteration and the stored candidates for the memory units from the current iteration. The TPU and memory units are then executed in parallel, which decreases the latency significantly compared to the straightforward mapping of the algorithm. In order to maximize the throughput of the IR-LSD algorithm architecture, the TPU and partial memory unit should be implemented with proper parallelism and

pipelining. Also the parameter L_{node} can be lowered to increase the throughput with the cost of decreased performance. The limit for the number of algorithm iterations L_{node} should be defined separately for different system configurations or according to the most complex supported configuration. A proper L_{node} value depends on the channel realization and on the search tree size, i.e., on the number of independent data streams and the constellation size $|\Omega|$.

5.3.4 VLSI Implementation Results

The SEE- and IR-LSD algorithm architectures were implemented for a 4×4 MIMO system with 16- and 64-QAM with the tools described in 5.2.2.2. The complexity results are given in area and in gate equivalents (GEs), where one GE corresponds to the area of a two-input drive-one NAND gate. The fixed-point word lengths were determined via computer simulations for a 4×4 MIMO–OFDM system and a maximum of 12 and 15 bits were found adequate for 16- and 64-QAM, respectively.

The SEE-LSD algorithm implementation is based on the architecture presented in Fig. 5.7. The SEE-LSD architecture TPU for 16-QAM was implemented with four parallel pipelined MULs in the first subunit and four parallel MULs in the latter subunit. The TPU for 64-QAM was implemented with four parallel pipelined MULs in the first subunit and eight parallel MULs in the latter subunit. Both algorithm implementations are done for output list size $N_{\text{cand}} = 15$. The synthesis results of the SEE-LSD algorithm implementation for the 0.18 µm CMOS technology are listed in Table 5.3. The IR-LSD algorithm implementation is based on the architecture presented in Fig. 5.9. The IR-LSD architecture TPU for 16-QAM was implemented, as in the SEE-LSD algorithm, with four parallel pipelined MULs in the first subunits and four parallel MULs in the latter subunits. The TPU for 64-QAM was implemented with four parallel pipelined MULs in the first subunit and eight parallel MULs in the latter subunit. Both algorithm implementations are done for output list size $N_{\text{cand}} = 15$. The memory unit $\mathcal{S}$ was implemented with dual port RAM to enhance the memory access. The maximum number of algorithm iterations is limited to 175 and 225 in the 16- and 64-QAM implementation, respectively. Thus, the memory unit size was 175×31 and 225×35 bits for the 16- and 64-QAM, respectively. The synthesis results of the IR-LSD algorithm implementation for the 0.18 µm CMOS technology are listed in Table 5.4.

Table 5.3 Synthesis results of the SEE-LSD algorithms for an SM system with $N = 4$

SEE-LSD	Area (mm^2)	kGEs	Latency	Power (mW)
16-QAM	0.13	10.6	13 cc/52 ns per it.	25
64-QAM	0.27	22.0	16 cc/64 ns per it.	38

Table 5.4 Synthesis results of the IR-LSD algorithms for an SM system with $N = 4$

IR-LSD	Area (mm^2)	kGEs	Latency	Power (mW)
16-QAM	0.31	25.4	14 cc/56 ns per it.	57
64-QAM	0.59	48.2	17 cc/68 ns per it.	90

5.3.4.1 Detection Rates

The results are applied to calculate detection rates of the algorithm implementations. The detection rate R_{det} denotes the amount of transmitted coded bits that the LSD algorithm implementation is able to detect in a certain time with a given complexity. The total detection rate R_{det} of the LSD algorithm implementation can be calculated as

$$R_{\text{det}} = \frac{NQ}{\Delta_{\text{tot}}} \text{bits/s}, \tag{5.9}$$

where Δ_{tot} corresponds to the throughput time of the LSD algorithm implementation. The throughput time for the sequential search algorithm implementations, the SEE-LSD algorithm and the IR-LSD algorithm, is defined as $\Delta_{\text{tot}} = \Delta_{\text{it}} L_{\text{avg}}^{\text{it}}$, where Δ_{it} is the latency per algorithm iteration and $L_{\text{avg}}^{\text{it}}$ is the average number of executed algorithm iterations. Thus, the achievable detection rate R_{det} depends on the defined maximum limit for visited nodes L_{node}, which should be properly selected to meet the desired FER target with a given channel realization and SNR γ. Implementation results of a K-best-LSD are also added for comparison [31]. It should be noted that the IR-LSD algorithm implementation checks two nodes in one algorithm iteration and that the K-best-LSD algorithm implementation detection rate is fixed as the algorithm search goes through a fixed number of nodes with variable performance depending on the channel realization and SNR. Also it should be noted that the implementation of multiple parallel LSD algorithms can be used to achieve a higher detection rate.

The detection rates of the LSD algorithm ASIC implementations for 16- and 64-QAM in different channel environments are listed in Table 5.5. The listed SNR range is selected as the operating range of the LSD based receiver with a given configuration and channel environment. The detection rates of the SEE-LSD algorithm and IR-LSD algorithm implementations are lower at low SNR as more algorithm iterations are required to achieve adequate performance. The detection rates at high SNR correspond to cases where the minimum number of iterations provides adequate performance. The LSD algorithm implementations have different performances and complexities, and, thus, we also compare the efficiency of the implementations. The comparison is done with an algorithm work factor W_{alg}, which is calculated as a multiplication product between the used resources in terms of GEs and the implementation throughput time per subcarrier Δ_{tot}, and a smaller value reflects a more efficient implementation [42, 43]. The algorithm work

Table 5.5 Detection rates of the LSD algorithm ASIC implementations in different channel environments

$R_{\text{det}}^{(\text{asic})}$ (dB)	IR-LSD alg. (Mbps)	SEE-LSD alg. (Mbps)	K-best-LSD alg. (Mbps)
16-QAM, UNC, $\gamma = [13-19]$	$[4.14, 31.7]$	$[1.07, 34.2]$	62.5
16-QAM, CORR, $\gamma = [21-26]$	$[1.70, 31.7]$	$[0.35, 34.2]$	62.5
64-QAM, UNC, $\gamma = [20-25]$	$[3.71, 39.2]$	$[1.12, 41.6]$	93.8
64-QAM, CORR, $\gamma = [30-35]$	$[1.62, 39.2]$	$[0.30, 41.6]$	93.8

Table 5.6 Performance and work factor numbers of the LSD algorithm ASIC implementations in different channel environments

		IR-LSD alg.	SEE-LSD alg.	K-best-LSD alg.
16-QAM, UNC	W_{alg}	$[0.097, 0.013]$	$[0.158, 0.005]$	0.030
$\gamma = [13-19]$ dB	Perf.	Max-log—0.6 dB	Max-log—0.8 dB	Max-log—0.4 dB
16-QAM, CORR	W_{alg}	$[0.239, 0.013]$	$[0.472, 0.005]$	0.030
$\gamma = [21-26]$ dB	Perf.	Max-log—0.5 dB	Max-log—0.5 dB	Max-log—1.2 dB
64-QAM, UNC	W_{alg}	$[0.311, 0.029]$	$[0.473, 0.013]$	0.050
$\gamma = [20-25]$ dB	Perf.	Max-log—1.2 dB	Max-log—1.2 dB	Max-log—0.9 dB
64-QAM, CORR	W_{alg}	$[0.715, 0.029]$	$[1.757, 0.013]$	0.050
$\gamma = [30-35]$ dB	Perf.	Max-log—0.7 dB	Max-log—0.7 dB	Max-log—2.0 dB

factor values of the LSD algorithm ASIC implementations for 16- and 64-QAM in different channel environments are listed in Table 5.6. Also the performances of the implementations relative to the max-log-MAP detector are listed in Table 5.6.

All of the LSD algorithm implementations have advantages in certain channel environments and SNR values. The K-best-LSD algorithm implementation achieves rather good performance in the uncorrelated channel with a fixed W_{alg}, but the performance suffers in highly correlated channels. The algorithm work factor W_{alg} is best in low SNR values, but the performance cannot be tuned with the channel as in the sequential search algorithms. The SEE-LSD algorithm implementation is the most efficient in high SNR values, but is the least efficient in low SNR because of the algorithm search strategy. The IR-LSD algorithm implementation is more efficient at low SNR compared to the SEE-LSD algorithm implementation and more efficient at high SNR compared to the K-best-LSD algorithm implementation. Both sequential search algorithm implementations perform much better compared to the K-best-LSD algorithm implementation in highly correlated channels with the cost of additional complexity. The performance of the sequential search algorithms can also be tuned with the penalty of additional complexity according to the requirements.

We also calculated the required parallel LSD algorithm resources with 0.18 µm CMOS technology for a downlink receiver in a 3GPP LTE standard transmission with 20 MHz bandwidth and with $N_{\text{used}} = 1{,}200$ subcarriers. We assume a 4×4

Table 5.7 The required LSD algorithm ASIC resources in the LSD detection of 3GPP LTE standard with 20 MHz BW and with $N_{\text{used}} = 1{,}200$ subcarriers

4×4 MIMO–OFDM, 16-QAM	Area (mm^2)	kGEs	Power (mW)
IR-LSD alg.	2.6–49.0	216–4,010	485–9,000
SEE-LSD alg.	1.0–100	83–8,140	198–19,200
K-best-LSD alg.	6.5	547	1,480

MIMO–OFDM system with 16-QAM, i.e., the LSD algorithm must be capable of the detection rate of 268.8 Mbps. The required 0.18 µm CMOS technology resources are scaled linearly from the LSD algorithm implementation results and are listed in Table 5.7. The required resources with IR-LSD algorithm and SEE-LSD algorithm implementations depend on the defined performance of the algorithms as discussed earlier in this section.

5.4 Trellis-Search Based MIMO Detection

In this section, we introduce a trellis-search based detection algorithm for iterative MIMO detection [44–46]. We use an unconstrained trellis structure as an alternative to the tree structure to represent the search space of a MIMO signal. We describe a trellis-based approximate Log-MAP algorithm as a replacement of the typically used Max-Log algorithm for iterative MIMO detection. We search the trellis to find a number of most likely paths for each trellis node and compute a log-sum of a number of exponential terms corresponding to a hypothesized transmitted bit value. Near-optimal performance can be achieved by choosing an appropriate number of surviving paths in the trellis-search process. The trellis-based detection algorithm is a very data-parallel algorithm because the searching operations at multiple trellis nodes can be performed simultaneously. The local search complexity at each trellis node is kept very low to reduce the overall processing time. Moreover, the trellis-based detector can support iterative MIMO detection by utilizing the a priori information from the outer channel decoder.

5.4.1 Trellis-Search Algorithm

The LLR computation requires calculations of two log-sums of $\frac{Q^{N_t}}{2}$ exponential terms. The brute-force implementation is too expensive. As a balanced trade-off between complexity and performance, we can use a reduced number (n) of exponential terms to approximate the Log-MAP algorithm as:

$$LLR(x_{k,b}) \approx \ln \sum_{n:x_{k,b}=+1} \exp\left(-\frac{1}{2\sigma^2} d(\mathbf{s})\right) - \ln \sum_{n:x_{k,b}=-1} \exp\left(-\frac{1}{2\sigma^2} d(\mathbf{s})\right), \quad (5.10)$$

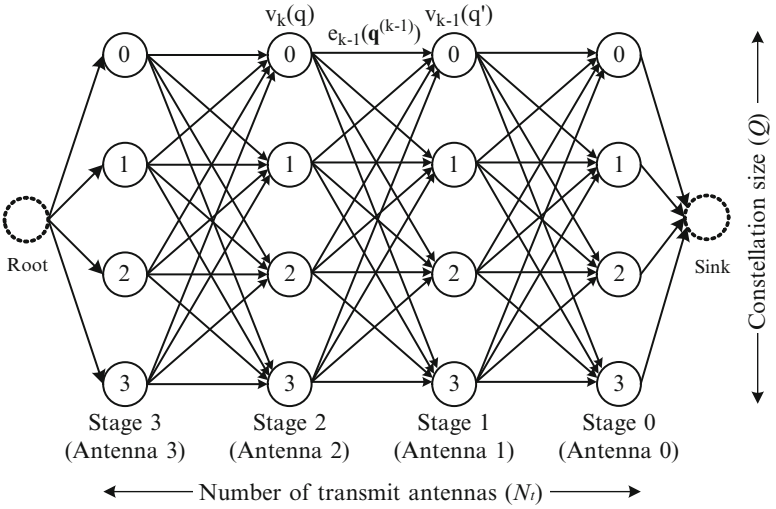

Fig. 5.10 Trellis model of a 4×4 4-QAM system

where the distance $d(\mathbf{s})$ is defined as:

$$d(\mathbf{s}) = \sum_{i=0}^{N_t-1} \left(|(\hat{\mathbf{y}})_i - (\mathbf{Rs})_i|^2 - \sigma^2 \sum_{j=0}^{B-1} x_{i,j} \cdot L_A(x_{i,j}) \right). \tag{5.11}$$

In the equation above, $\hat{\mathbf{y}} = \mathbf{Q}^H \mathbf{y}$, and $L_A(x_{i,j})$ is the a priori LLR for bit $x_{i,j}$. In order to implement (5.10), we must find n minimum distances $d(\mathbf{s})$ for each hypothesized transmitted data bit, i.e., $x_{k,b} = +1$ and $x_{k,b} = -1$. To realize this, we can use a trellis-search algorithm to find the n minimum distances.

5.4.2 *Trellis Model for Iterative MIMO Detection*

The search space of a MIMO signal can be represented with a compact trellis diagram. As an example, Fig. 5.10 shows the trellis diagram for a 4×4 4-QAM system. The trellis has N_t stages corresponding to N_t transmit antennas, and each stage contains Q different nodes corresponding to Q symbols of a complex constellation of the transmitted signal. In other words, the trellis is formed of columns representing the number of transmit antennas and rows representing values of a number of symbols with nodes at intersections. Each trellis node is physically mapped to a transmit symbol that belongs to a known modulation alphabet of the Q constellation symbols. Thus, any path through the trellis represents a possible

vector ($\mathbf{s}$) of transmitted symbols. Because of the upper triangular property of the matrix $\mathbf{R}$, the stages of the trellis are labeled in descending order. The trellis is fully connected, so there are Q^{N_t} number of different paths from the root node to the sink node. The nodes in stage k are denoted as $v_k(q)$, where $q = 0, 1, \ldots, Q-1$.

To compute the distance metric in (5.11) using the trellis model, we define a weight function $e_{k-1}(\mathbf{q}^{(k-1)})$ for each edge between node $v_k(q)$ in stage k and node $v_{k-1}(q')$ in stage $k-1$ as:

$$e_{k-1}(\mathbf{q}^{(k-1)}) = \left|\hat{y}_{k-1} - \sum_{j=k-1}^{N_t-1} R_{k-1,j} \cdot s_j\right|^2 - \sigma^2 \sum_{b=0}^{B-1} x_{k-1,b} \cdot L_A(x_{k-1,b}), \quad (5.12)$$

where $\mathbf{q}^{(k-1)} = [q_{N_t-1} \ \ldots \ q_k \ q_{k-1}]^T$ is the partial symbol vector, s_j is the complex-valued QAM symbol $s_j = QAM(q_j)$, B is the number of bits per constellation point, and $L_A(x_{k-1,b})$ is the a priori information for data bit $x_{k-1,b}$ provided by the outer channel decoder. In the first iteration, $L_A(x_{k-1,b})$ is not available and is set to 0. Note that the weight function not only depends on nodes $v_k(q)$ and $v_{k-1}(q')$, but also depends on all the nodes prior to node $v_k(q)$. In other words, depending on how we traverse the trellis, the weight function will get different values. We further define a path weight as the sum of the edge weights along the path. Then the distance metric as defined in (5.11) can be considered as a path weight, which can be computed recursively by adding up the edge weights along the path from the root node to the sink node. If we define a (partial) path metric d_k as the sum of the edge weights along this (partial) path, the path weight is then computed recursively as:

$$d_{k-1}(q') = d_k(q) + e_{k-1}(\mathbf{q}^{(k-1)}), \quad (5.13)$$

where $d_k(q)$ and $d_{k-1}(q')$ are the path weights associated with nodes $v_k(q)$ and $v_{k-1}(q')$, respectively, and $e_{k-1}(\mathbf{q}^{(k-1)})$ is the edge weight between node $v_k(q)$ and node $v_{k-1}(q')$.

In the trellis diagram, each trellis node $v_k(q)$ maps to a complex-valued symbol s_k such that each path from the root node to the sink node maps to a symbol vector $\mathbf{s}$. With the trellis model, we transform the MIMO detection problem into a per-node shortest paths problem, which is defined as follows. For each node $v_k(q)$ in the trellis diagram, we find a list of L most likely paths from the root node to the sink node over the node $v_k(q)$. The L most likely paths refer to the paths with the L shortest distances or the lowest L path weights. For each node, we only keep the L most likely paths and will discard all the other paths to reduce the complexity. The detection is performed layer by layer. In the trellis model, a layer corresponds to a stage in the trellis. In each stage k of the trellis, there are Q nodes, where each node corresponds to a constellation point. For each node $v_k(q)$ in stage k, $q = 0, 1, \ldots, Q-1$, we must find L shortest paths through the trellis, which are denoted as $\lambda_k^{(l)}(q)$, $l = 0, 1, \ldots, L-1$. Then, altogether QL candidates in each stage k of

the N_t stages of the trellis are used to compute the LLRs for data bits transmitted by antenna k as follows:

$$LLR(x_{k,b}) \approx \ln \sum_{(q,l):x_{k,b}=+1} \exp\Big(-\frac{1}{2\sigma^2}\lambda_k^{(l)}(q)\Big) - \ln \sum_{(q,l):x_{k,b}=-1} \exp\Big(-\frac{1}{2\sigma^2}\lambda_k^{(l)}(q)\Big). \quad (5.14)$$

With the trellis model, the detection problem now becomes a trellis-search problem. To detect a layer k, we need to search for L shortest paths for each node q in each stage k of the trellis diagram. The maximum theoretical value of the number L is Q^k, where $k = 0, 1, \ldots, N-1$ for the first stage, second stage, and etc., of the trellis. Practically, however, the number L should be kept small to reduce the complexity. The number L determines the detection performance: a larger L leads to better error performance. We will show later that even with a small L (such as $L = 2$ for $Q = 16$), the trellis-based detector can achieve good detection performance. To implement this algorithm, an exhaustive trellis-search approach would be very expensive. In order to reduce the search complexity, we use a greedy trellis-search algorithm that approximately finds the L shortest paths for each node in the trellis. In this search process, the trellis is first pruned by removing the unlikely paths. We refer to this pruning process as the "path reduction" process. In the path reduction process, the trellis is scanned from left to right, where each node retains the most likely L incoming paths using the local information it has so far. After the trellis is pruned, a second process, called the "path extension" process, is applied to extend the uncompleted paths so that each node will have L full paths through the trellis.

5.4.2.1 Path Reduction

Figure 5.11 illustrates a flow graph demonstrating a path reduction process. The path reduction process is configured to prune paths for each trellis node to a smaller number of surviving paths. The stages (columns) of the trellis are labeled in descending order, starting from stage $N_t - 1$ and ending with stage 0. Note that Fig. 5.11 illustrates only three successive stages, $k+1$, k, and $k-1$ among the N_t stages. As an example, we use a $Q = 4$, $L = 2$ case to explain the algorithm. In Fig. 5.11, each node receives $QL = 4 \times 2 = 8$ incoming paths from nodes in the previous stage of the trellis and, then, the $L = 2$ paths (the ones with the least cumulative path weights) are selected from the QL candidates. Next, the L survivors are expanded to the right so that each node will have the best QL outgoing paths forwarded to the next stage of the trellis. This process repeats until the end of the trellis. The path reduction process can effectively prune the trellis by keeping only L best incoming paths at each trellis node. As a result, each trellis node in the last stage of the trellis has L shortest paths through the trellis. However, other than the trellis nodes in the last stage, the path reduction process cannot guarantee that every trellis node will have L shortest paths through the trellis. These paths will be added as path extensions as described next.

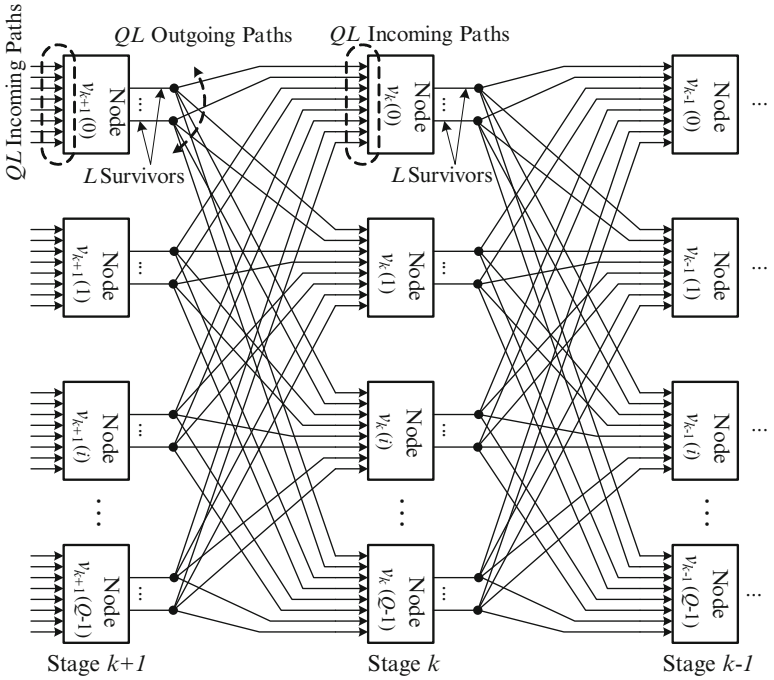

Fig. 5.11 Path reduction step

5.4.2.2 Path Extension

An objective of the trellis-based detection algorithm is to find L shortest paths for every node in the trellis. To achieve this goal, a path extension process is used after the path reduction process to fill in the missing paths for each trellis node. The goal is to extend the uncompleted paths so that each node will have L shortest paths through the trellis. The path extension is performed stage by stage (no path extension is required for the last stage), and node by node. Figure 5.12 is a flow graph demonstrating the path extension process. The path extension process is being demonstrated with respect to a node $v_k(i)$ in a stage k (i.e., the highlighted node in the figure). Note that all of the nodes in the same stage can be extended in parallel and independently.

As shown in Fig. 5.12, for a trellis node $v_k(i)$ (i.e., for the constellation point i in stage k), the path extension process first retrieves the $QL = 8$ outgoing path metrics computed in the path reduction step (at stage k), and then an extension process in stage $k-1$ is used to select the best $L = 2$ outgoing paths from $QL = 8$ candidates. Next, each of the $L = 2$ surviving paths is extended for the next stage of the trellis (stage $k-2$). Among the QL extended paths, only the best $L = 2$ paths are retained. This process repeats until the trellis has been completely traversed. As a result, the L shortest paths are obtained for node $v_k(i)$. Figure 5.12 shows a path extension

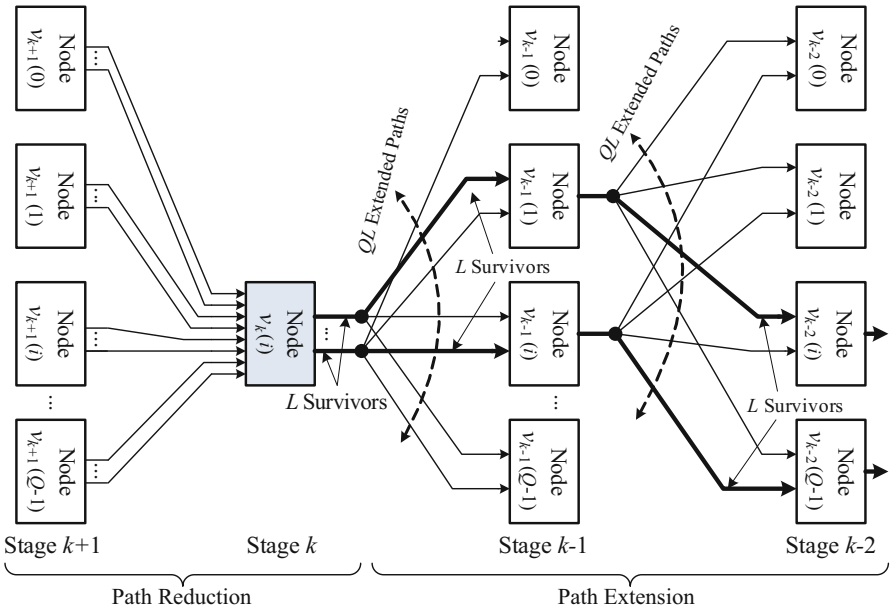

Fig. 5.12 Path extension step

process for one trellis node. The path reduction process is first performed until stage k of the trellis and next the path extension procedure is performed until the end of the trellis (stage 0). Note that the path extension process is to find the L best *outgoing* paths extending from a particular node.

5.4.2.3 LLR Computation

The most important feature of the trellis-based detection algorithm is that it will always guarantee that the bit LLR can be generated for every transmitted bit. For example, after the path reduction and the path extension processes are employed, every node $v_k(q)$ has successfully found L shortest paths or L minimum distances denoted as $\lambda_k^{(l)}(q)$, $l = 0, 1, \ldots, L-1$. We separate the LLR computation into two steps. A symbol reliability metric $\Gamma_k(q)$ is first computed for each node $v_k(q)$ as follows:

$$\Gamma_k(q) = \ln \sum_{l=0}^{L-1} \exp\Big(-\frac{1}{2\sigma^2}\lambda_k^{(l)}(q)\Big) = \mathop{\max}_{l}^{*}\Big(-\frac{1}{2\sigma^2}\lambda_k^{(l)}(q)\Big), \tag{5.15}$$

where the two-input $\max^*(\cdot)$ is defined as:

$$\mathop{\max}^{*}(a,b) \equiv \ln \sum (\exp(a) + \exp(b)) = \max(a,b) + \ln(1 + \exp(-|a-b|)). \tag{5.16}$$

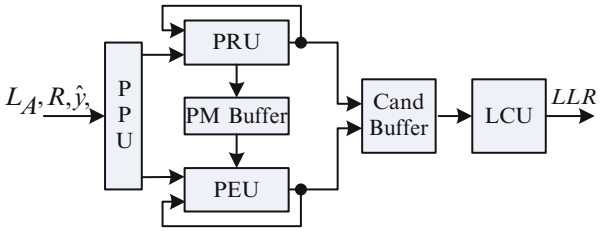

Fig. 5.13 VLSI architecture of the trellis-based iterative MIMO detector

Moreover, the n-input $\max^*(\cdot)$ for $n = 4, 8, 16$, etc., can be recursively computed based on the Jacobian algorithm. Then, the bit LLR is computed based on the symbol reliabilities $\Gamma_k(q)$:

$$\begin{aligned} LLR(x_{k,b}) &= \ln \sum_{q:x_{k,b}=+1} \exp(\Gamma_k(q)) - \ln \sum_{q:x_{k,b}=-1} \exp(\Gamma_k(q)) \\ &= \max^*_{q:x_{k,b}=+1} (\Gamma_k(q)) - \max^*_{q:x_{k,b}=-1} (\Gamma_k(q)). \end{aligned} \tag{5.17}$$

5.4.3 VLSI Architecture

Now we describe a high-speed VLSI architecture for the trellis-search based SfISfO MIMO detector. As a case study, we introduce a detector architecture with the surviving path number $L = 2$ for the 4×4 16-QAM system. Figure 5.13 shows the top level block diagram for the trellis-search MIMO detector. The detector consists of six main functional blocks: the path reduction unit (PRU), the path extension unit (PEU), the LLR calculation unit (LCU), the pre-processing unit (PPU), the path metric buffer (PM Buffer), and the candidate buffer (Cand Buffer). The PPU is used to pre-compute the initial path metrics and some constellation-dependent constant values that will be used by the PRU and the PEU. The PRU and the PEU are employed to implement the path reduction algorithm (cf. Fig. 5.11) and the path extension algorithm (cf. Fig. 5.12), respectively. The shortest path metrics found by the PRU and the PEU are stored in the Cand Buffer, which will then be used by the LCU to generate the LLR for each data bit based on (5.17). These blocks will be discussed in more detail in the following subsections.

Figure 5.14 shows the block diagram of PRU, which implements the path reduction algorithm. The PRU employs $Q = 16$ path calculation units (PCUs) and 16×2 minimum finder units (MFUs) to simultaneously process all the Q nodes in a trellis stage. This is a recursive architecture by reusing the hardware for processing nodes in different trellis stages. In Fig. 5.14, PCU i is used to compute the QL extended path metrics from node $v_k(i)$ to all the nodes in the next stage $k-1$. The extended path metrics are denoted as $\beta^{(l)}_{k-1}(i, j)$, where l is the surviving path

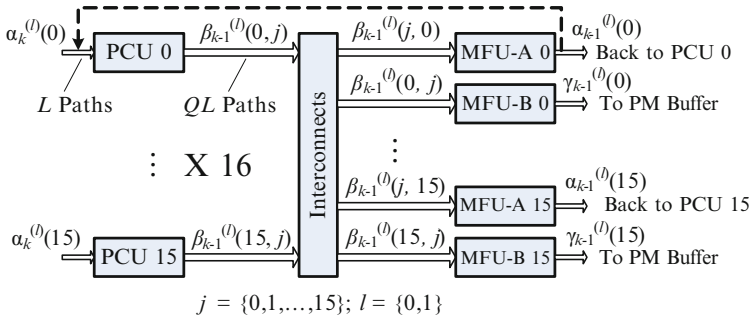

Fig. 5.14 Path reduction unit (PRU)

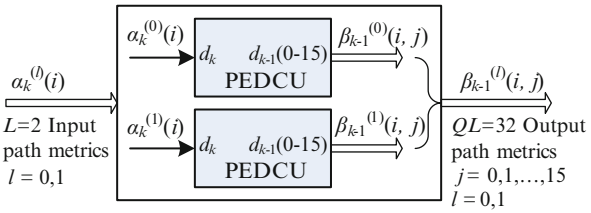

Fig. 5.15 Path calculation unit (PCU)

index ($l = 0, 1, \ldots, L-1$), i is the current node index, and j is the node index in the next stage ($j = 0, 1, \ldots, Q-1$). Next, the extended path metrics are gathered and sent to MFUs. In Fig. 5.14, MFU-A i is used to select the best L incoming paths to node $v_{k-1}(i)$, where the surviving path metrics are denoted as $\alpha_{k-1}^{(l)}(i)$, where $l = 0, 1, \ldots, L-1$. Then, these surviving paths are fed back to PCU i so that it can continue the processing for the next trellis stage. This operation is repeated until the trellis is completely traversed. MFU-B i is used to select the best L outgoing paths, denoted as $\gamma_{k-1}^{(l)}(i)$, from node $v_k(i)$ to any nodes in stage $k-1$. These best outgoing paths selected by MFU-B i will be stored into the path metric buffer (PM Buffer), which will be used later in the path extension process. Each PCU in Fig. 5.14 is used to compute $QL = 32$ path metrics in parallel. Figure 5.15 shows the block diagram of PCU which employs $L = 2$ PED calculation units (PEDCUs). For a given input path metric, or PED, d_k, one PEDCU needs to compute $Q = 16$ extended PEDs in parallel, denoted as $d_{k-1}(q)$, $q = 0, 1, \ldots, Q-1$. Figure 5.16 shows the hardware architecture for the PEDCU, which computes $Q = 16$ PEDs in parallel. Note that variables $R_{k-1,k-1}^2 |s_{k-1}(q)|^2$ and $\sigma^2 L_A(x_{k-1,b})$ are pre-computed in the PPU.

The MFU is used to select the best $L = 2$ path metrics from $QL = 32$ candidates. This type of $(32, 2)$ sorting can be done quickly by using a comparison tree. Note that the sorting cost of the trellis-based detector is much lower compared with the regular K-best detector which typically requires a larger (QK, K) sorting operation.

The PEU implements the path extension algorithm. As previously discussed, a path extension process is employed after the path reduction process to fill in the

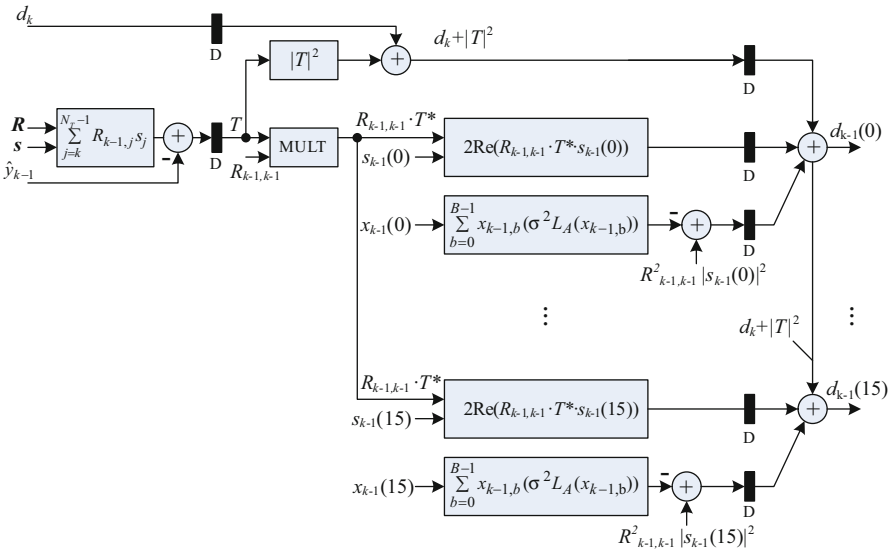

Fig. 5.16 Partial Euclidean distance calculation unit (PEDCU)

missing paths for each node so that every node will have L shortest paths through the trellis. The PEU has a very similar architecture to the PRU. The PEU employs $Q = 16$ PCUs and $Q = 16$ MFUs so that it can simultaneously extend Q nodes in a certain trellis stage. The PEU has a recursive architecture. In each iteration, PCU i calculates the QL extended path candidates based on the L input path metrics, and then, the MFU i selects the best L paths from these QL extended path candidates. The initial L input path metrics are retrieved from the PM Buffer, and, then, the PEU performs the path extension operation recursively.

5.4.4 VLSI Implementation Results

As a case study, we have developed a trellis-search based iterative MIMO detector ASIC module for a 4×4 16-QAM MIMO system. The fixed-point design parameters are summarized as follows. Each element in the $\mathbf{R}$ matrix is scaled by $\frac{1}{\sqrt{10N_t}} = \frac{1}{\sqrt{40}}$, and this scaled R is represented with 11 bits signed data S2.9 (2 integer bits with 9 fractional bits). The received signal y is represented with 11 bits signed data S5.6. The path metrics (PMs) are rounded to 13 bits between computational blocks. The LLR values are represented with 7 bit signed data S5.2. With this configuration, the fixed-point simulation result shows about $0.1 \sim 0.2$ dB performance degradation compared to a floating-point detector. The trellis-search detector has a pipelined architecture, where the pipeline stages for the PRU and PEU are $T = 4$. To maximize the throughput, we can feed four back-to-back MIMO

Table 5.8 VLSI implementation results for 4×4 trellis-search MIMO detector

Clock frequency	Throughput (1 iter.)	Core area	Gate count	Technology
320 MHz	1.7 Gbps	1.58 mm^2	1097K	65 nm

symbols in four consecutive cycles, e.g., at $t, t+1, t+2, t+3$ into the pipeline to fully utilize the hardware. The processing times for the path reduction process and the path extension process are both $3T = 12$ cycles, i.e., the iteration bound is 12 cycles. Thus, we can feed another four back-to-back MIMO symbols into the pipeline at $t+12, t+13, t+14, t+15$, and so forth. Furthermore, we can overlap the path reduction process with the path extension process to hide the processing delay. As a result, the maximum throughput of the detector is $\frac{4\times16\times fclk}{12} = \frac{16}{3} fclk$.

We have described the trellis-search detector with Verilog HDL and we have synthesized the design for a 1.08 V TSMC 65 nm CMOS technology using Synopsys Design Compiler. With a 320 MHz clock frequency, the detector can achieve a maximum throughput of 1.7 Gbps. Table 5.8 summaries the VLSI implementation results. From Table 5.8, one can observe that the trellis-search detector can achieve a very high data throughput (1.7 Gbps) while still maintaining a low area requirement (1.58 mm^2).

References

1. Hassibi B, Vikalo H (2005) On the sphere-decoding algorithm I. Expected complexity. IEEE Trans Signal Process 53(8):2806–2818
2. Paulraj A, Nabar RD, Gore D (2003) Introduction to space-time wireless communications. Cambridge University Press, Cambridge
3. Hochwald B, ten Brink S (2003) Achieving near-capacity on a multiple-antenna channel. IEEE Trans Commun 51(3):389–399
4. Hagenauer J (1997) The turbo principle: tutorial introduction and state of the art. In: Proceedings of the international symposium on turbo codes, Brest, France
5. Hagenauer J, Offer E, Papke L (1996) Iterative decoding of binary block and convolutional codes. IEEE Trans Inf Theory 42(2):429–445
6. Robertson P, Villebrun E, Hoeher P (1995) A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain. In: Proceedings of the IEEE international conference on communications, pp 1009–1013
7. Damen MO, Chkeif A, Belfiore J-C (2000) Lattice code decoder for space-time codes. IEEE Commun Lett 4(5):161–163
8. Murugan A, Gamal HE, Damen M, Caire G (2006) A unified framework for tree search decoding: rediscovering the sequential decoder. IEEE Trans Inf Theory 52(3):933–953
9. Anderson T (1984) An introduction to multivariate statistical analysis, 2nd edn. Wiley, New York
10. Anderson J, Mohan S (1984) Sequential coding algorithms: a survey and cost analysis. IEEE Trans Commun 32(2):169–176
11. Agrell E, Eriksson T, Vardy A, Zeger K (2002) Closest point search in lattices. IEEE Trans Inf Theory 48(8):2201–2214

12. Fincke U, Pohst M (1985) Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. Math Comput 44(5):463–471
13. Pohst M (1981) On the computation of lattice vectors of minimal length, successive minima and reduced basis with applications. ACM SIGSAM Bull 15:37–44
14. Viterbo E, Boutros J (1999) A universal lattice code decoder for fading channels. IEEE Trans Inf Theory 45(5):1639–1642
15. Damen MO, Gamal HE, Caire G (2003) On maximum–likelihood detection and the search for the closest lattice point. IEEE Trans Inf Theory 49(10):2389–2402
16. Myllylä M, Antikainen J, Juntti M, Cavallaro J (2007) The effect of LLR clipping to the complexity of list sphere detector algorithms. In: Proceedings of the annual Asilomar conference on signals, systems, and computers, Pacific Grove, 4–7 November 2007, pp 1559–1563
17. Wong K, Tsui C, Cheng RK, Mow W (2002) A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. In: Proceedings of the IEEE international symposium on circuits and systems, vol 3, Scottsdale, AZ, 26–29 May 2002, pp 273–276
18. Guo Z, Nilsson P (2006) Algorithm and implementation of the K-best sphere decoding for MIMO detection. IEEE J Sel Areas Commun 24(3):491–503
19. Barbero L, Thompson J (2008) Extending a fixed-complexity sphere decoder to obtain likelihood information for turbo-MIMO systems. IEEE Trans Vehicular Technol 57(5):2804–2814
20. Li M, Bougart B, Lopez E, Bourdoux A (2008) Selective spanning with fast enumeration: a near maximum-likelihood MIMO detector designed for parallel programmable baseband architectures. In: Proceedings of the IEEE international conference on communication, Beijing, China, 19–23 May 2008, pp 737–741
21. Guo Z, Nilsson P (2006) Algorithm and implementation of the K-best sphere decoding for MIMO detection. IEEE J Sel Areas Commun 24(3):491–503
22. Ketonen J, Myllylä M, Juntti M, Cavallaro JR (2008) ASIC implementation comparison of SIC and LSD receiver for MIMO-OFDM. In: Proceedings of the annual Asilomar conference on signals, systems, and computers, Pacific Grove, 25–29 October 2008, pp 1881–1885
23. Schnorr CP, Euchner M (1994) Lattice basis reduction: improved practical algorithms and solving subset sum problems. Math Program 66(2):181–191
24. Ketonen J, Juntti M, Cavallaro J (2010) Performance-complexity comparison of receivers for a LTE MIMO-OFDM system. IEEE Trans Signal Process 58(6):3360–3372
25. Myllylä M, Juntti M, Cavallaro JR (2007) Implementation aspects of list sphere detector algorithms. In: Proceedings of the IEEE global telecommunication conference, Washington, D.C., 26–30 November 2007, pp 3915–3920
26. Martin G, Smith G (2009) High-level synthesis: past, present, and future. IEEE Des Test Comput 26(4):18–25
27. Casseau E, Gal L, Bomel P, Jego C, Huet S, Martin E (2005) C-based rapid prototyping for digital signal processing. In: Proceedings of the European signal processing conference, Antalya, Turkey, 4–8 September 2005
28. Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z (2011) High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans Comput Aided Des Integr Circuits Syst 30(4):473–491
29. Calypto (2014) Catapult C synthesis overview. Technical report. http://calypto.com/en/products/catapult/overview
30. Ketonen J (2012) Equalization and channel estimation algorithms and implementations for cellular MIMO-OFDM downlink. Ph.D. dissertation, Acta Univ. Oul., C Technica 423, University of Oulu, Oulu
31. Myllylä M (2011) Detection algorithms and architectures for wireless spatial multiplexing in MIMO–OFDM systems. Ph.D. dissertation, Acta Univ. Oul., C Technica 380, University of Oulu, Oulu

32. Preyss N, Burg A, Studer C (2012) Layered detection and decoding in MIMO wireless systems. In: Conference on design and architectures for signal and image processing (DASIP), Karlsruhe, Germany, 23–25 October 2012, pp 1–8
33. Mohan S, Anderson JB (1984) Computationally optimal metric-first code tree search algorithms. IEEE Trans Commun 32(6):710–717
34. Dijkstra EW (1959) A note on two problems in connexion with graphs. In: Numerische Mathematik, vol 1. Mathematisch Centrum, Amsterdam, pp 269–271
35. Knuth D (1997) The art of computer programming. Volume 3: sorting and searching, 3rd edn. Addison-Wesley, Reading
36. Baro S, Hagenauer J, Witzke M (2003) Iterative detection of MIMO transmission using a list-sequential (LISS) detector. In: Proceedings of the IEEE international conference on communications, vol 4, pp 2653–2657
37. Xu W, Wang Y, Zhou Z, Wang J (2004) A computationally efficient exact ML sphere decoder. In: Proceedings of the IEEE global telecommunication conference, vol 4, 29 November – 3 December 2004, pp 2594–2598
38. Hagenauer J, Kuhn C (2007) The list-sequential (LISS) algorithm and its application. IEEE Trans Commun 55(5):918–928
39. Studer C, Burg A, Bolcskei H (2008) Soft-output sphere decoding: algorithms and VLSI implementation. IEEE J Sel Areas Commun 26(2):290–300
40. Burg A, Borgmanr M, Wenk M, Studer C, Bolcskei H (2006) Advanced receiver algorithms for MIMO wireless communications. In: Proceedings of the design, automation and test in Europe (DATE'06), vol 1, March 2006, 6 pp
41. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms. MIT Press, Cambridge
42. Ullman J (1994) Computational aspects of VLSI. Computer Science Press, Rockville
43. Bajwa RS, Owens R, Irwin M (1994) Area time trade-offs in micro-grain VLSI array architectures. IEEE Trans Comput 43(10):1121–1128
44. Sun Y, Cavallaro JR (2009) High throughput VLSI architecture for soft-output MIMO detection based on a Greedy graph algorithm. In: ACM great lakes symposium on VLSI design, May 2009, pp 445–450
45. Sun Y, Cavallaro JR (2012) High-throughput soft-output MIMO detector based on path-preserving trellis-search algorithm. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(7):1235–1247
46. Sun Y, Cavallaro JR (2012) Trellis-search based soft-input soft-output MIMO detector: algorithm and VLSI architecture. IEEE Trans Signal Process 60(5):2617–2627

Chapter 6
Stochastic Decoders for LDPC Codes

François Leduc-Primeau, Vincent C. Gaudet, and Warren J. Gross

6.1 Introduction

Low density parity check (LDPC) codes have now been established as one of the leading channel codes for approaching the channel capacity in data storage and communication systems. Compared to other codes, they stand out by their ability of being decoded with a message-passing decoding algorithm that offers a large degree of parallelism, which offers interesting possibilities for simultaneously achieving a large coding gain and a high data throughput.

Exploiting all the available parallelism in message-passing decoding is difficult because of the logic area required for replicating processing circuits, but also because of the large number of wires required for exchanging messages. The use of stochastic computing was proposed as a way of achieving highly parallel LDPC decoders with a smaller logic and wiring complexity.

Current decoding algorithms based on stochastic computing do not outperform standard algorithms on all fronts, but they generally offer a significant advantage in average processing throughput normalized to circuit area.

We start this Chapter by providing an overview of LDPC codes. Readers familiar with the topic can safely skip to Sect. 6.3, where we present the stochastic number representation and the concept of stochastic computing. Section 6.4 then describes several LDPC decoding algorithms that perform all their computations using the stochastic representation. It is also possible to use the stochastic representation

F. Leduc-Primeau • W.J. Gross (✉)
McGill University, Montreal, QC, Canada
e-mail: francois.leduc-primeau@mail.mcgill.ca; warren.gross@mcgill.ca

V.C. Gaudet
University of Waterloo, Waterloo, ON, Canada
e-mail: vcgaudet@uwaterloo.ca

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_6

for only a subset of the computations. Section 6.5 presents an algorithm that uses that approach and presents methods for efficiently converting numbers between a conventional (deterministic) representation and a stochastic representation. Finally, Sect. 6.6 presents a stochastic approach for decoding non-binary LDPC codes.

6.2 Overview of LDPC Codes

In this section, we review the aspects of LDPC codes that are required to explain the stochastic decoding algorithms. We start in Sect. 6.2.1 by describing the structure of the codes, and Sect. 6.2.2 then presents the standard decoding algorithm, called the Sum–Product algorithm (SPA).

6.2.1 Structure

LDPC codes are part of the family of linear block codes, which are commonly defined using a parity-check matrix H of size $m \times n$. The codewords corresponding to H are the column vectors $\mathbf{x}$ of length n for which $H \cdot \mathbf{x} = \mathbf{0}$, where $\mathbf{0}$ is the zero vector. LDPC codes can be binary or non-binary. For a binary code, the elements of H and $\mathbf{x}$ are from the Galois Field of order 2, or equivalently $H \in \{0,1\}^{m \times n}$ and $\mathbf{x} \in \{0,1\}^n$. Non-binary LDPC codes are defined similarly, but the elements of H and $\mathbf{x}$ are taken from higher order Galois Fields. The rate r of a code expresses the number of information bits contained in the codeword divided by the code length. Assuming H is full rank, we have $r = 1 - \frac{m}{n}$.

A block code can also be equivalently represented as a bipartite graph. We call a node of the first type a *variable* node (VN), and a node of the second type a *check* node (CN). Every row i of H corresponds to a check node c_i, and every column j of H corresponds to a variable node j. An edge exists between c_i and v_j if $H(i,j)$ is non-zero. The two equivalent representations are illustrated in Fig. 6.1.

The key property that distinguishes LDPC codes from other linear block codes is that their parity-check matrix is sparse (or "low density"), in the sense that each row and each column contains a small number of non-zero elements. Furthermore this number does not depend on n. In other words, increasing the code size n also increases the sparsity of H. The number of non-zero elements in a row of H is equal to the number of edges incident on the corresponding check node and is called the check node degree, denoted d_c. Similarly the number of non-zero elements in a column is called the variable node degree and denoted d_v.

When all the rows of H have the same degree d_c and all the columns have the same degree d_v, we say that the code is part of the family of regular (d_v, d_c) codes. Code families are important for analyzing the error correction performance of an LDPC code. When n is sufficiently large, two codes taken from the same family will have the same error correction performance with high probability when decoded

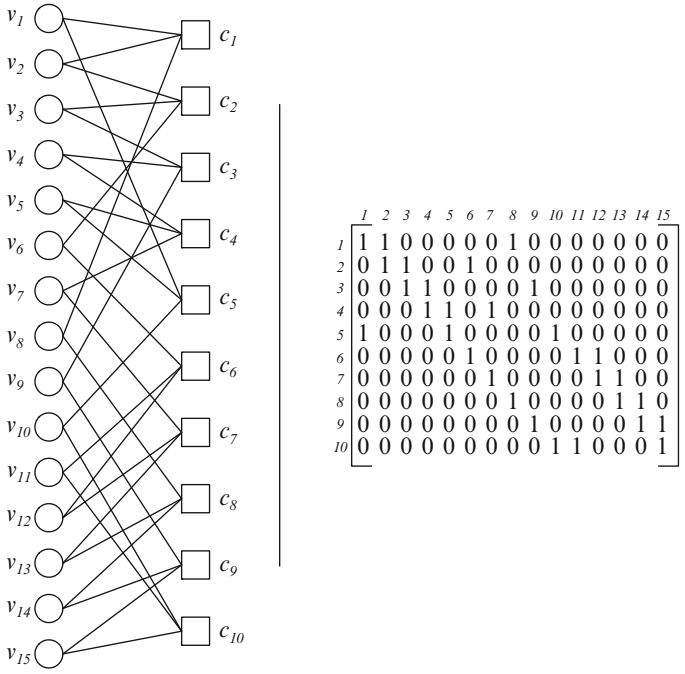

Fig. 6.1 *Left*: Graphical representation of a short binary LDPC code with $n = 15$ and $m = 10$. *Circles* represent variable nodes and *squares* represent check nodes. *Right*: Parity-check matrix representation of the same code

with the SPA. For a large n it is therefore possible to predict the performance of a code by instead analyzing the average performance of the corresponding family of codes, which is a much simpler task. Families of irregular codes can also be defined by specifying a VN degree distribution and a CN degree distribution. For example, we can define a family of irregular codes with rate $\frac{1}{2}$ by specifying that 40 % of the VNs have degree 3 and 60 % have degree 4, while 20 % of the CNs have degree 12 and 80 % have degree 6. The reader is advised to consult a reference such as [15] for more information on the construction and analysis of LDPC codes.

6.2.2 Decoding

LDPC codes can be decoded using a variety of message-passing algorithms that operate by passing messages on the edges of the code graph. These algorithms are interesting because they have a low complexity per bit while also offering a high level of parallelism. If the graph contains no cycles, there exists a message-passing algorithm that yields the maximum-likelihood estimate of each transmitted

bit, called the SPA. In practice, all good LDPC codes contain cycles, and in that case the SPA is not guaranteed to generate the optimal estimate of each symbol. Despite this fact, the SPA usually performs very well on graphs with cycles, and experiments have shown that an LDPC code decoded with the SPA can still be used to approach the channel capacity [4]. The SPA can be defined in terms of various likelihood metrics, but when decoding binary codes, the log likelihood ratio (LLR) is preferred because it is better suited to a fixed-point representation and removes the need to perform multiplications. Suppose that p is the probability that the transmitted bit is a 1 (and $1-p$ the probability that it is a 0). The LLR metric Λ_i is defined as

$$\Lambda_i = \ln\left(\frac{1-p}{p}\right).$$

Algorithm 1 describes the SPA for binary codes (the SPA for non-binary codes is described in Sect. 6.6.1). The algorithm takes LLR priors as inputs and outputs an estimate of each codeword bit. If the modulated bits are represented as $x_i \in \{-1,1\}$ and transmitted over the additive white Gaussian noise channel, the LLR priors Λ_i corresponding to each codeword bit $i \in \{1,2,\ldots,n\}$ are obtained from the channel output y_i using

$$\Lambda_i = \frac{-2y_i}{\sigma^2},$$

where σ^2 is the noise variance. The algorithm operates by passing messages on the code graph. We denote a message passed from a variable node i to a check node j as $\eta_{i,j}$, and from a check node j to a variable node i as $\theta_{j,i}$. Furthermore, for each variable node v_i we define a set V_i that contains all the check node neighbors of v_i, and similarly for each check node c_j, we define a set C_j that contains the variable node neighbors of c_j. The computations can be described by two functions: a variable node function $\mathrm{VAR}(S)$ and a check node function $\mathrm{CHK}(S)$, where S is a set containing the function's inputs. If we let $S = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_d\}$, the functions are defined as follows:

$$\mathrm{VAR}(S) = \sum_{i=1}^{d} \Lambda_i \tag{6.1}$$

$$\mathrm{CHK}(S) = \operatorname{arctanh}\left(\prod_{i=1}^{d} \tanh(\Lambda_i)\right). \tag{6.2}$$

The algorithm performs up to L iterations, and stops as soon as the bit estimate vector $\hat{\mathbf{x}}$ forms a valid codeword, that is $H \cdot \hat{\mathbf{x}} = \mathbf{0}$.

input : $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_n\}$
output: $\hat{\mathbf{x}} = [\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n]$
begin
  $\theta_{j,i} \leftarrow 0, \forall i, j$
  for $t \leftarrow 1$ *to* L **do**
    for $i \leftarrow 1$ *to* n **do** `// VN to CN messages`
      foreach $j \in V_i$ **do**
        $\eta_{i,j} \leftarrow \text{VAR}(\{\Lambda_i\} \cup \{\theta_{a,i} : a \in V_i\} \setminus \{\theta_{j,i}\})$
    for $j \leftarrow 1$ *to* m **do** `// CN to VN messages`
      foreach $i \in C_j$ **do**
        $\theta_{j,i} \leftarrow \text{CHK}(\{\eta_{a,j} : a \in C_j\} \setminus \{\eta_{i,j}\})$
    for $i \leftarrow 1$ *to* n **do** `// Compute the decision vector`
      $\Lambda'_i \leftarrow \text{VAR}(\{\Lambda_i\} \cup \{\theta_{a,i} : a \in V_i\})$
      if $\Lambda'_i \geq 0$ **then** $\hat{x}_i \leftarrow 0$
      else $\hat{x}_i \leftarrow 1$
    Terminate if $\hat{\mathbf{x}}$ is a valid codeword
  Declare a decoding failure

Algorithm 1: Sum–Product decoding of an LDPC code using LLR messages

6.3 Stochastic Computing

Stochastic computing was originally studied by Gaines [6] and Poppelbaum [12]. At the time, the motivation was to allow realizing digital circuit implementations of systems that were too complex to be implemented using a conventional number representation, by taking advantage of the fact that many computations can be performed on stochastic streams using very simple circuits. The downside of stochastic computing is that it has a limited precision, but the combination of low complexity and low precision is well suited to a number of applications. For example, stochastic computing has been proposed for the circuit implementation of neural networks [3], which can benefit from massive parallelism and can often tolerate low precision computation. Additionally, with the continuous shrinking of transistors in CMOS integrated circuits, process variations as well as random fluctuations in operating parameters have now become important limiting factors for the speed and energy efficiency of circuits. Because of its random nature, stochastic computing is naturally fault-tolerant, and there has been renewed interest in using it to achieve fault-tolerant circuit implementations [14]. A comprehensive survey of applications for which stochastic computing has been considered can be found in [1].

6.3.1 The Stochastic Stream Representation

To explain stochastic streams, it is useful to make a parallel with analog signals. Let us consider a discretized version of a real-valued analog signal that we denote $a[t]$, for $t \in \{1, 2, 3, \ldots\}$. A stochastic stream $x[t]$ can be thought of as a random sequence

that is a coarse quantization of $a[t]$, but where the randomness is used to ensure that the quantization is unbiased, so that the expected value of $x[t]$ is always $a[t]$. If we denote the expected value of a random variable X as $\mathbb{E}[X]$, this can be written as $\mathbb{E}[x[t]] = a[t]$.

Since the motivation of stochastic computing is reducing the complexity of the computing circuits, stochastic streams are usually binary-valued. In that case they can be communicated using a single wire in the circuit, just like analog signals. A binary stochastic stream is defined as a sequence of independent random variables $x[t]$, distributed such that $\mathbb{E}[x[t]] = a[t]$. Note that since $x[t]$ is binary, its distribution is fully determined by its expected value. Unless otherwise mentioned, the term *stochastic stream* refers to a binary stochastic stream. If we label the two values of a binary stochastic stream as 0 and 1 we have $x[t] \in \{0,1\}$, and therefore the range of values that can be represented is $0 \leq \mathbb{E}[x[t]] \leq 1$. In order to represent analog signals that are valued over a different range, we can simply map $a[t]$ to a signal $a'[t]$ such that $a'[t] \in [0,1]$. If the mapping is one-to-one, it can be inverted at the output of the computation to restore the initial range. We call the sequence $\mathbb{E}[x[t]]$ the expectation sequence of the stochastic stream $x[t]$.

An important aspect of any number representation is its precision. Suppose that a real number a is represented as $\hat{a}$. The precision can be defined as the worst error of $\hat{a}$ with respect to a. As a point of comparison, we consider the precision of representing a as a fixed-point number, since this is the most common approach in conventional signal processing systems. Suppose that we want to represent a real number $a \in [0,1]$ as an n-bit fixed-point number $\hat{a}$. We obtain $\hat{a}$ using

$$\hat{a} = \frac{\text{ROUND}(a \cdot 2^n)}{2^n},$$

where the rounding is to the nearest integer. Since $|\hat{a} - a| \leq 1/2^n$, we say that the precision is $1/2^n$.

We can now compare this with the precision of a stochastic stream $x[t]$ where t runs from 1 to n. Let us first consider a simple case where the expectation sequence $a[t]$ does not vary in time, that is $\mathbb{E}[x[t]] = a[t] = c$, with $c \in [0,1]$. The sequence $a[t]$ can be reconstructed from $x[t]$ in an optimal way by using the mean estimator

$$\hat{a}[t] = \frac{1}{t} \sum_{i=1}^{t} x[i]. \tag{6.3}$$

Since the stochastic stream is generated randomly, the precision of a sequence of length n is also random. To simplify, let us assume that the length of the sequence is known in advance, and that the sequence is generated deterministically to get the best precision possible, which is achieved by choosing the sequence such that $\sum_{i=1}^{n} x[i] = \text{ROUND}(c \cdot n)$. In this case we have

$$|\hat{a}[t] - a[t]| = \left| \frac{\text{ROUND}(c \cdot t)}{t} - c \right| \leq \frac{1}{2t}. \tag{6.4}$$

The precision of the stochastic stream improves with time, since the estimation error $|\hat{a}[t] - a[t]|$ goes to 0 for all c, as t goes to infinity. However, to obtain the same precision as a fixed-point representation with n bits, we must wait until $t = 2^{n-1}$. If $n \leq 2$, the two representations have equivalent precision, but as n increases, the stochastic representation needs exponentially more bits to achieve the same precision. Why then use stochastic streams? The hope is that the gains made in the complexity of the computation circuits are more important than the loss in precision. Also, a nice consequence of the randomness of stochastic streams is that the desired precision does not need to be established in advance, but instead can be chosen dynamically by running the computation for a shorter or longer time. Nonetheless, it is clear that stochastic streams are only suitable for computations that require relatively low precision.

If $a[t]$ varies in time, the optimal estimator will be different from (6.3), but it is easy to see that in that case the estimation error depends not only on time but also on the rate of change of $a[t]$ (the smallest error being obtained when $a[t]$ is constant).

6.3.2 Computation Circuits

A stochastic stream $x[t]$ can be generated from a real sequence $a[t] \in [0,1]$ by comparing it with a random threshold T having a uniform distribution over $[0,1]$:

$$x[t] = \begin{cases} 0 & \text{if a[t]} < \text{T}, \\ 1 & \text{otherwise.} \end{cases} \tag{6.5}$$

In some cases, computations can be performed on stochastic streams using very simple circuits. Unless otherwise specified, circuits for stochastic computation are designed by assuming that the input streams are mutually independent, which is the case if they are generated using independent threshold variables. Figure 6.2 shows two basic computing circuits. The first circuit is an AND gate. Suppose that A and B are independent binary random variables with $\Pr(A = 1) = p_A$ and $\Pr(B = 1) = p_B$. Then $\Pr(C = 1) = p_A p_B$, and therefore the AND gate performs a multiplication. The second circuit is a multiplexer. Similarly, let $\Pr(D = 1) = p_D$, $\Pr(E = 1) = p_E$, $\Pr(Z = 1) = p_Z$, then $\Pr(F = 1) = (1 - p_Z)p_D + p_Z p_E$, and therefore the circuit performs a convex combination of streams D and E, with the weights determined by p_Z. Note that it is not possible to directly perform the addition of two streams D and E, since the domain of $p_D + p_E$ is $[0,2]$. However, a normalized addition can be implemented by using $p_Z = \frac{1}{2}$.

Several other circuits have been proposed to implement more complex functions (see [1] for more examples).

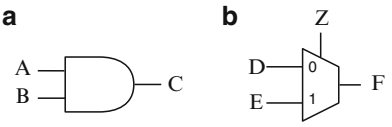

Fig. 6.2 Stochastic computing circuits for (**a**) multiplication and (**b**) convex combination

6.4 Fully Stochastic Decoders

This section discusses LDPC decoding algorithms in which all computations are performed in the stochastic domain, which we refer to as fully stochastic decoders. Other algorithms that only perform a subset of the computations in the stochastic domain are described in Sects. 6.5 and 6.6. Section 6.4.1 introduces the fundamental circuits used in a stochastic LDPC decoder. Following this, Sects. 6.4.2 and 6.4.3 describe two different extensions of the simple stochastic algorithm that both allow decoding practical LDPC codes.

6.4.1 A Simple Stochastic Decoder

LDPC decoders have the potential to achieve a high throughput because each of the n codeword bits can be decoded in parallel. However, the length of the codes used in practice is on the order of 10^3, going up to 10^5 or more. This makes it difficult to make use of all the available parallelism while still respecting circuit area constraints. One factor influencing area utilization is of course the complexity of the VAR and CHK functions to be implemented, but because of the nature of the message-passing algorithm, the wires that carry messages between processing nodes also have a large influence on the area, as was identified early on in one of the first circuit implementations of an SPA LDPC decoder [2].

The need to reduce both logic and wiring complexity suggests that stochastic computing could be a good approach. The use of stochastic computation for the message-passing decoding of block codes was first proposed by Gaudet and Rapley [7]. The idea was prompted by the realization that the two SPA functions VAR and CHK had very simple stochastic implementations when performed in the probability domain. Let us first consider the CHK function. In the LLR domain, the function is given by (6.2), which in the probability domain becomes

$$\mathrm{CHK}(p_1, p_2, \ldots, p_d) = \frac{1 - \prod_{i=1}^{d}(1-2p_i)}{2}. \tag{6.6}$$

The implementation of this function in the stochastic domain is simply an exclusive-OR (XOR) gate. That is, if we have independent binary random variables $X_1, X_2, \ldots, X_d$, each distributed such that $\Pr(X_i = 1) = p_i$, then taking

$$Y = X_1 + X_2 + \cdots + X_d \mod 2 \tag{6.7}$$

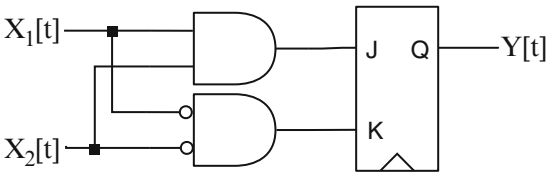

Fig. 6.3 Stochastic computation circuit for the two-input VAR function in the probability domain

yields $\Pr(Y=1) = \mathrm{CHK}(p_1, p_2, \ldots, p_d)$. This result is not as surprising as it might seem. Indeed, the modulo-2 sum is exactly the constraint that must be satisfied by the codeword bits involved in this check node operation. Using stochastic streams instead of codeword bits is akin to performing a Monte-Carlo simulation to find the probability associated with an unknown bit connected to this check node.

A circuit for computing the VAR function with two inputs was also presented in [7]. In the probability domain, the LLR function of (6.1) with $d = 2$ is given by

$$\mathrm{VAR}(p_1, p_2) = \frac{p_1 p_2}{p_1 p_2 + (1-p_1)(1-p_2)}. \tag{6.8}$$

In the stochastic domain, this function can be computed approximately using the circuit shown in Fig. 6.3. The JK flip-flop becomes 1 if its J input is 1, and 0 if its K input is 1. Otherwise, it retains its previous value. This implementation is different in nature from the one used for the CHK function, since it contains a memory. The behavior of the circuit can be analyzed by modeling the output Y as a Markov chain with states $Y = 0$ and $Y = 1$. Suppose that the stochastic streams $X_1[t]$ and $X_2[t]$ are generated according to the expectation sequences $p_1[t]$ and $p_2[t]$, respectively, and let the initial state be $Y[0] = s_o$. Then, at time $t = 1$, we have

$$\mathbb{E}[Y[1]] = \Pr(Y[1]=1) = \begin{cases} p_1[1]p_2[1] & \text{if } \mathrm{s_o} = 0, \\ p_1[1] + p_2[1] - p_1[1]p_2[1] & \text{if } \mathrm{s_o} = 1. \end{cases}$$

None of the expressions above are equal to $\mathrm{VAR}(p_1[1], p_2[1])$, and therefore the expected value of the first output of the circuit is incorrect, irrespective of the starting state. However, if we assume that the input streams are independent and identically distributed (i.i.d.) with $p_1[t] = p_1$ and $p_2[t] = p_2$, it is easy to show that the Markov chain converges to a steady-state such that

$$\lim_{t\to\infty} \mathbb{E}[Y[t]] = \mathrm{VAR}(p_1, p_2). \tag{6.9}$$

To build a circuit that will compute the VAR function for more than two inputs, we can make use of the fact that the VAR function can be distributed arbitrarily, which can easily be seen by considering the equivalent LLR-domain formulation in (6.1). For example we have $\mathrm{VAR}(p_1, p_2, p_3) = \mathrm{VAR}(\mathrm{VAR}(p_1, p_2), p_3)$.

Stochastic decoders built using these circuits were demonstrated for very small codes, but they are unable to decode realistic LDPC codes. The reason is that (6.9)

is not sufficient to guarantee the accuracy of the variable node computation, since we do not know that the input streams are stationary or close to stationary. In graphs with cycles, low precision messages can create many fixed points in the decoder's iterative dynamics that would not be there otherwise. This was noted in [21], and the authors proposed to resolve the precision issue by adding an element called a *supernode*, which takes one stochastic stream as input and outputs another stochastic stream. It interrupts the feedback path by using a constant expectation parameter to generate the output stochastic stream. Simultaneously, it estimates the mean of the incoming stochastic stream. The decoding is performed in several iterations, and an iteration is completed once a stochastic stream of length ℓ has been transmitted on every edge of the graph. Once the iteration completes, the expectation parameter in each supernode is updated with the output of the mean estimator. Because the expectation parameter is kept constant while the stochastic streams are being generated, the precision can be increased by increasing ℓ.

While the supernode approach works, it requires large values of ℓ to achieve sufficient precision, and therefore a lot of time for transmitting the ℓ bits in each iteration. However, it is not necessary for the expectation parameter of a stochastic stream to be constant. Any method that can control the rate of change of the expectation sequences will allow avoiding fixed points in the decoding algorithm created by insufficient precision. In particular, this can be achieved by using low-pass filters, some of which are described in Sect. 6.4.2.

6.4.2 Stochastic Decoders Using Successive Relaxation

Now that we have explained the basic concepts used to build stochastic decoders, we are ready to present stochastic decoding algorithms that are able to decode practical LDPC codes. Most such algorithms make use of a smoothing mechanism called Successive Relaxation. In Sect. 6.4.2.1, we explain how Successive Relaxation was introduced into the domain of LDPC decoding algorithms, and how it is used in stochastic decoders. Then, in Sect. 6.4.2.2, we present circuit implementations for the stochastic computation of the variable node function. Using stochastic streams to represent likelihood information can sometimes cause precision issues. The main problem lies in the inability of accurately representing probability values very close to 0 or 1. Section 6.4.2.3 discusses this range problem and how it can be mitigated. Finally, Sect. 6.4.2.4 summarizes the performance of a fully stochastic decoder when decoding the LDPC code standardized by the IEEE 802.3an standard for 10 Gbps Ethernet networking.

6.4.2.1 The Role of Successive Relaxation

Message-passing LDPC decoders are iterative algorithms. We can express their iterative progress by defining a vector $\mathbf{x}_o$ of length n containing the information

received from the channel, and a second vector $\mathbf{x}[t]$ of length n_e containing the messages sent from each variable node to its neighboring check nodes at iteration t, where n_e is the number of edges in the graph. The standard SPA decoder for an LDPC code is an iterative algorithm that is memoryless, by which we mean that the messages sent on the graph edges at one iteration only depend on the initial condition, and on the messages sent at the previous iteration. As a result, the decoder's progress can be represented as follows:

$$\mathbf{x}[t] = h(\mathbf{x}[t-1], \mathbf{x}_o),$$

where $h()$ is a function that performs the check node and variable node message updates, as described in Algorithm 1.

In the past, analog circuit implementations of SPA decoders have been considered for essentially the same reasons that motivated the research into stochastic decoders. Since these decoders operate in continuous time, a different approach was needed to simulate their decoding performance. The authors of [8] proposed to simulate continuous-time SPA by using a method called *successive relaxation* (SR). Under SR, the iterative progress of the algorithm becomes

$$\mathbf{x}[t] = (1-\beta) \cdot \mathbf{x}[t-1] + \beta \cdot h(\mathbf{x}[t-1], \mathbf{x}_o), \tag{6.10}$$

where $0 < \beta \leq 1$ is known as the relaxation factor. As $\beta \to 0$, the simulated progress of the decoder approaches a continuous-time (analog) decoder. However, the most interesting aspect of this method is that it can be used not only as a simulator, but also as a decoding algorithm in its own right, usually referred to as Relaxed SPA. Under certain conditions, Relaxed SPA can provide significantly better decoding performance than the standard SPA.

In stochastic decoders, SR cannot be applied directly because the vector of messages $\mathbf{x}[t]$ is a binary vector, while $\mathbf{x}[t]$ obtained using (6.10) is not if $\beta < 1$. However, if we want to add low-pass filters to a stochastic decoder, we must add memories that can represent the expectation domain of the stochastic streams. Suppose that we associate a state memory with each edge, and group these memories in a vector $\mathbf{s}[t]$ of length n_e. Since the expectation domain is the probability domain, the elements of $\mathbf{s}[t]$ are in the interval $[0, 1]$. Stochastic messages can be generated from the edge states by comparing each edge state to a random threshold, as described in (6.5). We can then rewrite (6.10) as a mean tracking filter, where $\mathbf{s}[t]$ is the vector of estimated means after iteration t, and $\mathbf{x}[t]$, $\mathbf{x}_o[t]$ are vectors of stochastic bits:

$$\mathbf{s}[t] = (1-\beta) \cdot \mathbf{s}[t-1] + \beta \cdot h(\mathbf{x}[t-1], \mathbf{x}_o[t-1]). \tag{6.11}$$

The value of β controls the rate at which the decoder state can change, and since $\mathbb{E}[\mathbf{x}[t]] = \mathbf{s}[t]$, it also controls the precision of the stochastic representation.

6.4.2.2 Circuit Implementations of the VN Function

We will first consider stochastic variable node circuits with two inputs $X_1[t]$ and $X_2[t]$. As previously, we denote by $p_1[t]$ and $p_2[t]$ the expectation sequences associated with each input stream. Let E be the event that $X_1[t] = X_2[t]$. We have that

$$\mathbb{E}[X_1[t] \,|\, E] = \mathbb{E}[X_2[t] \,|\, E] = \text{VAR}(p_1[t], p_2[t]),$$

where VAR() is defined in (6.8). Therefore, one way to implement the variable node function for stochastic streams is to track the mean of the streams at the time instants when they are equal. As long as $\Pr(E) > 0$, a mean tracker can be as close as desired to $\text{VAR}(p_1[t], p_2[t])$ if the rate of change of $p_1[t]$, $p_2[t]$ is appropriately limited. If the mean tracker takes the form of (6.11), this corresponds to choosing a sufficiently small β.

The first use of relaxation in the form of (6.11) was proposed in [17], where the relaxation (or mean tracking) step is performed in the variable node, by using a variable node circuit that is an extension of the original simple circuit shown in Fig. 6.3. In the original VN circuit, each graph edge had a corresponding 1-bit flip-flop. This flip-flop can be extended to an ℓ-bit shift-register, in which a "1" is shifted in if both inputs $X_1[t]$ and $X_2[t]$ are equal to 1, and a "0" is shifted in if both inputs are equal to 0. When a new bit is shifted in, the oldest bit is discarded.

Let us denote the number of "1" bits in the shift-register by $w[t]$, and define the current mean estimate as $s[t] = w[t]/\ell$. If we make the simplifying assumptions that the bits in the shift register are independent from $X_1[t]$ and $X_2[t]$, and that when a bit is added to the shift register, the bit to be discarded is chosen at random, then it is easy to show that the shift-register implements the successive relaxation rule of (6.10) *in distribution*, with $\beta = \Pr(E)/\ell$, in the sense that

$$\mathbb{E}[s[t]] = \left(1 - \frac{\Pr(E)}{\ell}\right) \cdot s[t-1] + \frac{\Pr(E)}{\ell} \cdot \text{VAR}(p_1[t-1], p_2[t-1]).$$

When the variable node degree is large, it was suggested in [18] to implement the variable node function using a computation tree with two levels. Let us denote the computation performed by the first level circuit as VARST_1 and by the second level circuit as VARST_2. For example, the circuit for a degree-6 VN can be implemented as

$$\text{VARST}(x_o, x_1, \ldots, x_5) = \text{VARST}_2(\text{VARST}_1(x_1, x_2, x_3), \text{VARST}_1(x_0, x_4, x_5)),$$

where x_o is the current stochastic bit corresponding to the channel information, and $x_1, x_2, \ldots, x_5$ are the stochastic bits received from the neighboring check nodes. The corresponding circuit is shown in Fig. 6.4. When using such a two-level implementation, it is proposed in [18] to use small shift-registers for the first level, and a large one for the second level.

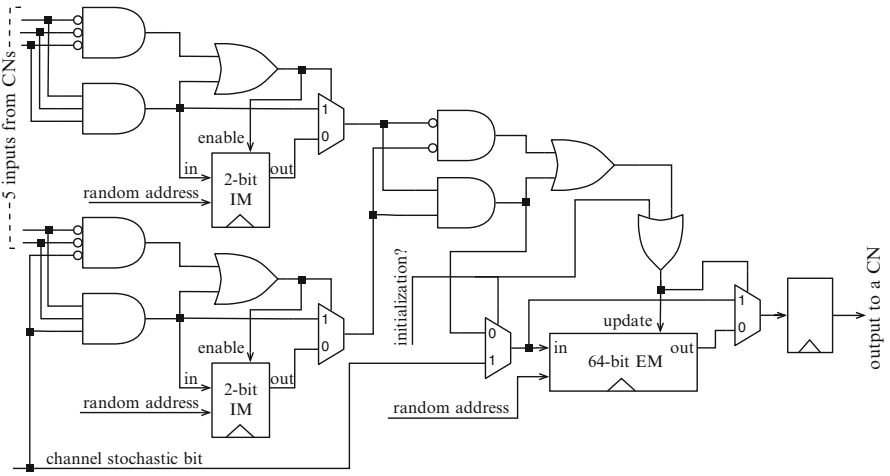

Fig. 6.4 Degree-6 stochastic VN circuit corresponding to one output [18]

Using a shift register is interesting from a pedagogical point of view because it uses a stochastic representation for the mean estimate. However, there are several reasons that discourage its use in decoders. First, the choice of relaxation factor β is tied to the precision of the mean estimate, which prevents from freely optimizing the value of β. Second, because the mean is represented stochastically, storing a high precision estimate requires a lot of register bits, which are costly circuit components. Lastly, the relaxation factor β is not constant, since $\beta = \Pr(E)/\ell$. This can complicate the analysis and design of the decoder.

For these reasons, a better approach to performing the mean tracking, proposed in [19], is to directly represent the mean estimate $s[t]$ as a fixed-point number. The VN computation with inputs X_1, X_2 can now be implemented as

$$s[t+1] = \begin{cases} (1-\beta)\cdot s[t] + \beta & \text{if } \mathrm{X}_1 = \mathrm{X}_2 = 1, \\ (1-\beta)\cdot s[t] & \text{if } \mathrm{X}_1 = \mathrm{X}_2 = 0, \\ s[t] & \text{otherwise.} \end{cases}$$

If β is chosen as a negative power of 2, the update can be implemented with only bit shifts and additions.

6.4.2.3 Tweaking the Probability Domain Representation

The stochastic computations used in decoding LDPC codes are expressed in the probability domain because that greatly simplifies the computation of the check node function, which becomes a simple XOR gate, as stated in (6.7). It was found

that when running SPA in the LLR domain, the performance is adversely affected if the representable range of LLR values is too limited. The probability domain representation requires a large number of bits to represent large LLR magnitudes, and representing the probability as a stochastic stream requires a further exponential increase in the number of bits used, as was discussed in Sect. 6.3.1. To illustrate this, consider a stationary stochastic stream of length t. The smallest non-zero probability value that it can represent is $1/t$, and therefore the largest LLR value $\Lambda_{\max}$ less than infinity that it can represent is

$$\Lambda_{\max} = \ln\left(\frac{1-\frac{1}{t}}{\frac{1}{t}}\right) = \ln(t-1).$$

For example, the ability to distinguish an LLR value of 2 from larger values requires a stationary stream of length $t \geq 9$, while distinguishing an LLR of 4 from larger values requires $t \geq 56$.

One way to improve the representable range is to note that when decoding an LDPC code, we are not interested in approximating the true a-posteriori probabilities of each transmitted bit, but simply of identifying which codeword is more likely to have been transmitted. Therefore, we can try scaling down the prior LLR values to increase the representation precision for large LLR magnitudes (at the expense of decreasing the representation precision for values near zero). Suppose that we apply a scaling factor α to all LLR values. The minimal length of a stationary stream that will distinguish $\Lambda_{\max}$ from larger values becomes

$$t \geq \lceil e^{\alpha \Lambda_{\max}} + 1 \rceil$$

Continuing the previous example, if we set $\alpha = \frac{1}{2}$, distinguishing an LLR value of $\Lambda_{\max} = 2$ now requires only $t \geq 4$, and $t \geq 9$ for $\Lambda_{\max} = 4$. This LLR scaling approach was initially proposed in [17] under the name Noise Dependent Scaling, where it was shown that it is necessary to choose the right scaling factor in order to achieve a low error rate when the channel signal-to-noise ratio is large.

6.4.2.4 Benchmark Using the IEEE 802.3an Standard

The IEEE 802.3an 10GBASE-T standard (10 Gbps Ethernet networking over CAT-6 copper cables) uses an LDPC code of length 2,048 and rate 0.841 for error correction. Because of its industrial relevance and high throughput requirements, this code became a popular benchmark for LDPC decoders. A stochastic decoder for this code that uses the techniques presented in Sects. 6.4.2.1–6.4.2.3 was described in [20]. In addition, this decoder uses a heuristic circuit optimization called "MTFM" where a single relaxation filter is used simultaneously by the d_v outputs of a variable node circuit.

The main strength of this stochastic decoder compared to other decoder implementations is that it achieves a high average decoding throughput, especially when normalizing for circuit area. Its main inconvenient is that its worst-case decoding time is significantly larger. This worst case can be dealt with by adding buffering at the input, but many applications have stringent latency requirements that prevent such buffering.

6.4.3 The "Delayed" Stochastic Decoder

The simple stochastic decoder discussed in Sect. 6.4.1 is interesting for the very low complexity of the circuits required, but unfortunately it is unable to decode practical codes because the precision of the messages exchange is in most cases insufficient to ensure convergence. The relaxation-based approach presented in Sect. 6.4.2 resolves this issue by controlling the rate of change of the expectation sequences associated with each stochastic stream, allowing the decoder to converge. The "delayed" stochastic (DS) decoder introduced in [11] takes a different approach to resolving the convergence problem, with the objective of eliminating a large portion of the register circuits required in relaxation-based decoders.

Instead of controlling the precision of the stochastic streams, the DS decoder propagates "freeze" flags that indicate whether a stochastic message is reliable or not. If a check node receives a freeze flag, it propagates the flag to its neighbors and the messages sent by this check node are ignored in the variable nodes.

6.4.3.1 Simple DS Decoder

In its simplest form, the DS decoder uses the same variable node function as the simple stochastic decoder of Sect. 6.4.1. In the first iteration, the variable nodes send a stochastic bit generated from the channel probability. In following iterations, the variable nodes receive messages from the check nodes, and the reliability of each output of a VN is determined by the presence of an agreement among input bits. If there is no agreement for generating a given VN output, a "freeze" flag is propagated on that output.

6.4.3.2 Some Heuristics for Improved Convergence

In the simple DS algorithm, the proportion of messages that are deemed reliable was observed to be insufficient in many cases, resulting in very slow convergence. Two modifications to the algorithm were proposed to improve the decoding time. First, small memories called "internal memories" (IM) are added to the variable node circuit to provide some averaging of the stochastic streams. The resulting VN circuit used to generate one output then becomes identical to the one in Fig. 6.4, but

without the 64-bit edge-memory (EM). Instead of sampling from the EM, the DS circuit outputs a "freeze" flag. Second, a special condition is added to further reduce the number of "freeze" flags being propagated. When a majority of a VN's outputs would be sending "freeze" flags, the VN circuit instead enters a special state where no "freeze" flags are sent, and the output messages are sampled from the IMs.

6.4.3.3 Benchmark Using the IEEE 802.3an Standard

A hardware decoder for the code specified in the IEEE 802.3an standard is reported in [11]. The results show that the decoding latency of the DS algorithm is longer than for the relaxation-based decoder presented in Sect. 6.4.2. The latency can be made equivalent at the cost of reducing the coding gain by 0.2 dB. However, the DS decoder occupies a much smaller area: 3.93 mm^2 instead of 6.38 mm^2 (in a 90 nm technology). The average throughput of the DS decoder is also 2.8 times higher.

These hardware results show that it is possible for a stochastic decoder to converge on practical LDPC codes even when the circuit contains a very small amount of memory. However more work must be done to improve the worst-case convergence time and the coding gain.

6.5 Mixing Stochastic and Conventional Computations

It is a common occurrence in computer architecture that the complexity of a computation can be reduced by changing the representation domain of the data. For example, applying a Fourier transform can simplify a complex convolution operation into a simpler multiplication, while operating with a logarithmic representation simplifies multiplications into additions. Besides the issue of numerical precision which must also be considered, the decision to use an alternate representation in an implementation must take into account the other computations that are required by the algorithm, and the cost of converting one representation domain to another.

This situation is found in the SPA. As was presented in Sect. 6.4.1, performing the check node computation in the stochastic domain results in an exact implementation (in distribution) of the SPA check node function that has a very low complexity. On the other hand, the stochastic variable node computation can only become exact if the input streams are stationary, which is a case of limited interest since it implies that the decoder is not making any progress. Among exact implementations of the variable node computation, the simplest is obtained by using the LLR domain, for which the computation is simply an addition.

In this section, we present an algorithm called the Relaxed Half-Stochastic (RHS) algorithm [10] that uses the stochastic domain for the check node computation and the LLR domain for the variable node computation, and which includes mechanisms to efficiently convert the values between the two domains.

The stochastic domain has the potential of greatly simplifying some types of computations, but there are also situations where it is more likely to increase the complexity, for example when a computation requires a high precision. The approach presented here could therefore also provide opportunities for other algorithms that would benefit from using stochastic computation in only a portion of the algorithm.

6.5.1 The RHS Algorithm

The structure of the RHS algorithm, like all other algorithms presented in this chapter, is very similar to that of the SPA, described in Algorithm 1. In fact, part of the variable node computation is identical to the SPA, since it operates in the LLR domain. When describing the algorithm we assume that messages are exchanged according to the so-called *flooding* schedules. The steps described can be trivially modified to use other schedules.

Like the SPA, the RHS algorithm takes as input a set of n LLR values $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_n\}$ that correspond to the n bits received from the channel. However, it is easier to describe the algorithm from the perspective of individual variable nodes and check nodes. Still following the SPA described in Algorithm 1, the first iteration begins by generating the LLR values $\eta_{i,j}$, where the index i runs over the variable nodes, while j runs over the neighboring check nodes of the variable node. When considering a particular variable node i we can simplify the notation to η_j. In RHS, we call the η_j values *intermediate VN outputs*. These intermediate VN outputs are then converted to stochastic messages and sent to neighboring check nodes. The check node computation is performed in the stochastic domain, and stochastic messages are sent back from check nodes to variable nodes, which completes the iteration.

The main interest of the algorithm lies in the mechanism used to convert to and from the stochastic domain. This is described in Sects. 6.5.2 and 6.5.3.

6.5.2 Domain Conversion: LLR to Stochastic

The RHS algorithm uses an extended version of stochastic streams where each stream element consists of a binary vector of length k. Therefore, when a variable node sends a message to a check node, it transmits k bits, either serially or in parallel. Let us denote such a message as $\mathbf{X} = [X_1, X_2, \ldots, X_k]$, where each X_i is an independent binary random variable. In order to use the simple check node function common to all stochastic LDPC decoders, we define the expectation parameter in the probability domain. The fact that the k bits in a message vector are generated simultaneously allows exploring various generation rules, but one way to define $\mathbf{X}$ is to have independent and identically distributed bits, such that $\mathbb{E}[X_1] = \mathbb{E}[X_2] = \cdots = \mathbb{E}[X_k] = p_j$. The expectation parameter p_j is simply the LLR-domain intermediate output η_j converted to the probability domain:

$$p_j = \frac{1}{e^{\eta_j} + 1}.$$

One way to generate the stochastic bits is to compare p_j to a random threshold, uniformly distributed over the probability domain. However, the computation of p_j can be avoided by instead converting the threshold to the LLR domain. Let Z be a uniform random variable over the open interval $(0,1)$. An LLR threshold T is obtained using $T = \ln((1-Z)/Z)$. We then generate a stochastic bit for each $i \in [1,k]$ by comparing the intermediate output with a threshold sampled from T:

$$X_i = \begin{cases} 0 & \text{if } \eta_i > T, \\ 1 & \text{otherwise.} \end{cases} \tag{6.12}$$

In the circuit implementation, a realization of Z can be generated easily using a linear feedback shift-register (LFSR) circuit, and the conversion to the LLR domain can be approximated accurately using a priority encoder circuit and a few additional logic gates.[1]

As discussed in Sect. 6.4.1, the stochastic check node circuit composed of XOR gates implements the SPA check node function provided that the stochastic inputs are independent. This requirement determines which thresholds in the decoder must be independent. Any two messages that are sent to different check nodes can be generated with the same threshold. On the other hand, each of the k bits in the message vector must be independent. Therefore, $k \cdot d_c$ independent thresholds are required in a complete decoder, where d_c is the check node degree. Since $kd_c \ll n$, the complexity of the random threshold generator circuit does not have a large impact on the complexity of the decoder.

6.5.3 Domain Conversion: Stochastic to LLR

A check node is connected to d_c variable nodes, and therefore at every decoding iteration it receives d_c messages and outputs d_c messages. However, each output message is generated from only $d_c - 1$ messages, following the extrinsic information principle at the center of the SPA.

The RHS check node function therefore takes $d_c - 1$ stochastic vectors $\mathbf{X}_i$ as inputs, each of length k. Let $i \in [1, d_c - 1]$ be an index running over the check node inputs, and $j \in [1,k]$ an index running over the bits in each vector. Therefore we can denote an individual bit inside a vector as $X_{i,j}$. To simplify the presentation, we consider the generation of one of the d_c check node output messages. Using XOR gates, the check node function computes an output vector $\mathbf{Y} = [Y_1, \ldots, Y_k]$ as

[1] A detailed example of a circuit implementation can be found in [9].

$$Y_j = \sum_{i=1}^{d_c-1} X_{i,j} \mod 2,$$

for $j \in [1,k]$.

This output vector $\mathbf{Y}$ is then transmitted back to a neighboring variable node. At this point, we need to use the knowledge of how the vectors $\{\mathbf{X}_1,\ldots,\mathbf{X}_{d_c}\}$ were generated to interpret $\mathbf{Y}$. Using the generation rule described in 6.5.2, the expectation of $\mathbf{Y}$ lies in the probability domain, and since the bits are i.i.d., the maximum-likelihood estimate $\hat{m}$ of the probability represented by $\mathbf{Y}$ is

$$\hat{m} = \frac{1}{k}\sum_{j=1}^{k} Y_j. \tag{6.13}$$

We define a set $\mathscr{M}$ such that $\hat{m} \in \mathscr{M}$. The set $\mathscr{M}$ contains all the possible messages that can be received by a variable node in a single iteration. Under (6.13), we have $\mathscr{M} = \{0, \frac{1}{k}, \frac{2}{k}, \ldots, 1\}$, for a total of $k+1$ distinct messages.

For small k values, the precision of the message received is too crude for the iterative decoder to converge, and therefore we are interested in tracking the mean of the messages received over the iterations. Let $\hat{m}[t]$ be a message received on an input of a variable node at iteration t, and let $s[t]$ be the mean probability estimate at the end of iteration t, with $s[0] = \frac{1}{2}$. For $t \geq 1$, we can apply the relaxation filter to generate an updated estimate $s[t]$:

$$s[t] = (1-\beta)s[t-1] + \beta\hat{m}[t]. \tag{6.14}$$

However, this estimate $s[t]$ is in the probability domain, whereas we are interested in converting the stochastic message to the LLR domain. For small values of k, we can design a tracking filter operating in the LLR domain by studying the transfer function corresponding to each message in $\mathscr{M}$. The LLR domain tracking then simply amounts to choosing the right transfer function according to the received message $\hat{m}$:

$$\Lambda[t] = f(\Lambda[t-1]; \hat{m}[t]),$$

where $\Lambda[t]$ is the LLR estimate after iteration t, and $f(\Lambda;\hat{m})$ is the transfer function corresponding to message $\hat{m}$. We obtain the *ideal* transfer functions by expressing (6.14) in the LLR domain, which yields

$$\Lambda[t] = \ln\left(\frac{e^{\Lambda[t-1]} + \beta(1-\hat{m}[t](e^{\Lambda[t-1]}+1))}{1-\beta(1-\hat{m}[t](e^{\Lambda[t-1]}+1))}\right).$$

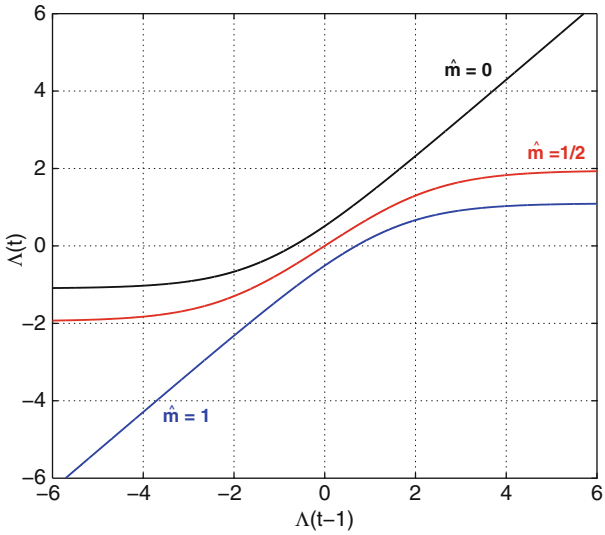

Fig. 6.5 LLR tracker transfer functions for $\mathscr{M} = \{0, \frac{1}{2}, 1\}$ and $\beta = \frac{1}{4}$

However, using the ideal functions directly in the circuit implementation would result in overly complex circuits. Instead, we must approximate each transfer function using simple linear functions, or alternatively using look-up tables, leaving the logic synthesis tool to optimize the circuit. An implementation should also exploit the symmetry of the transfer functions. We have that $f(\Lambda;\mu_i) = -f(-\Lambda;\mu_{k+2-i})$, where μ_i is the i-th largest message in $\mathscr{M}$.

For example, Fig. 6.5 shows the transfer functions for $\mathscr{M} = \{0, \frac{1}{2}, 1\}$ (hence $k=2$) and $\beta = \frac{1}{4}$. These functions can be approximated with good accuracy by the following piecewise-linear functions:

$$f(\Lambda;0) \approx \begin{cases} \Lambda + b & \text{if } \Lambda \geq \mathrm{d}, \\ d & \text{if } \Lambda < \mathrm{d}, \end{cases} \tag{6.15}$$

$$f(\Lambda;\tfrac{1}{2}) \approx \begin{cases} a\Lambda & \text{if } -\mathrm{c} \leq \Lambda \leq \mathrm{c}, \\ -c & \text{if } \Lambda < -\mathrm{c}, \\ c & \text{if } \Lambda > \mathrm{c}, \end{cases} \tag{6.16}$$

where the parameters a, b, c, d are chosen based on the value of β. The $f(\Lambda;1)$ function is obtained using the symmetry rule. Using these approximations, it is possible to design a very simple circuit that will convert the stochastic messages into LLR values, while also implementing the relaxation filter.

6.5.4 Benchmark Using the IEEE 802.3an Standard

Hardware implementation results of an RHS decoder for the LDPC code included in the IEEE 802.3an standard were reported in [9]. Compared to the other stochastic decoders presented in this chapter, the decoder based on the RHS algorithm occupies a larger area. Implemented in a 65 nm CMOS technology, the RHS decoder uses 4.41 mm^2, while using a quadratic scaling law, the equivalent area for the relaxation-based fully stochastic decoder of Sect. 6.4.2 is 3.33 mm^2, and that of the delayed stochastic decoder of Sect. 6.4.3 is 2.05 mm^2. In return for the increased area, the RHS decoder reduces the worst-case latency by 3.6 times, while simultaneously providing an additional coding gain of 0.2 dB with respect to the relaxation-based stochastic decoder, and 0.4 dB with respect to the delayed stochastic decoder. Finally, its average throughput is 2.6 times larger than the relaxation-based decoder, and similar to that of the delayed stochastic decoder.

6.6 Stochastic Decoders for Non-Binary LDPC Codes

Non-binary LDPC codes were shown to outperform their binary counterpart at an equivalent bit length [5], and furthermore are particularly interesting for channels that exhibit bursty error patterns, which are prominent in applications such as data storage and wireless communication. Unfortunately, they are also difficult to decode. Stochastic computation is one approach that has been explored to reduce the complexity of the decoding algorithm.

6.6.1 Message-Passing Decoding

In a non-binary code, codeword symbols can take any value from the Galois Field (GF) of order q. The field order is usually chosen as a power of 2, and in that case we denote the power as p, that is $2^p = q$. The information received about a symbol can be expressed as a probability mass function (PMF) that, for each of the q possible values of this symbol, indicates the probability that it was transmitted, given the channel output. For a PMF U, we denote by $U[\gamma]$ the probability corresponding to symbol value $\gamma \in \mathrm{GF}(q)$. Decoding is achieved by passing messages representing PMFs on the graph representation of the code, as in message-passing decoding of binary codes. However, when describing the algorithm, it is convenient to add a third node type called a *permutation* node (PN), which handles part of the computation associated with the parity-check constraint. The permutation nodes are inserted on every edge in the graph, such that any message sent from a VN to a CN or from a CN to a VN passes through a permutation node, resulting in a tripartite graph.

At every decoding iteration, a variable node v receives d_v PMF messages from neighboring permutation nodes. A PMF message sent from v to a permutation node p at iteration t, denoted by $U_{vp}^{(t)}$, is given by

$$U_{vp}^{(t)} = \text{NORM}\left(L_v \times \prod_{p' \neq p} U_{p'v}^{(t-1)} \right), \tag{6.17}$$

where L_v is the channel PMF, and NORM() is a function that normalizes the PMF so that all its probabilities sum to 1. A PN p receives a message $U_{vp}^{(t)}$ from a VN and generates a message to a CN c by performing

$$U_{pc}^{(t)}[\gamma h_p] = U_{vp}^{(t)}[\gamma], \ \ \forall \gamma \in \text{GF}(q),$$

where h_p is the element of matrix H that corresponds to VN v (which specifies the column) and CN c (which specifies the row). A CN c receives d_c messages from permutation nodes and generates messages by performing

$$U_{cp}^{(t)} = \bigstar_{p' \neq p} U_{p'c}^{(t)}, \tag{6.18}$$

where $\bigstar$ is the convolution operator. Finally, a message sent by a CN also passes through a PN, but this time the PN performs the inverse operation, given by

$$U_{pv}^{(t)}[\gamma h_p^{-1}] = U_{cp}^{(t)}[\gamma], \ \ \forall \gamma \in \text{GF}(q),$$

where h_p^{-1} is such that $h_p \times h_p^{-1} = 1$.

Among the computations described above, the multiplications required in (6.17) are costly to implement, but (6.18) has the highest complexity, since the number of operations required scales exponentially in the field order q and in the CN degree d_c.

6.6.2 Stochastic Decoding Algorithms

Several ways of applying stochastic computing to the decoding of non-binary LDPC codes are proposed in [16]. Among these, the only algorithm that can decode any code, without restrictions on q or d_v, is the non-binary version of the RHS algorithm. Just like in the binary version presented in Sect. 6.5, the non-binary RHS decoder uses the stochastic representation for the check node computation only.

A stochastic message sent to a check node consists of a single symbol $X \in \text{GF}(q)$, which can be represented using p bits. In comparison, messages exchanged in the SPA consist of complete PMF vectors requiring $qQ = 2^p Q$ bits, where Q is the number of bits used to represent one probability value in the PMF.

Therefore, stochastic algorithms significantly reduce the routing complexity of a circuit implementation.

For binary codes, the stochastic check node function is an addition over GF(2), which requires $d_c(d_c-1)$ XOR gates for generating all d_c outputs. For non-binary codes, it is an addition over GF(q). Assuming that q is a power of two, the addition can still be implemented using only XOR gates, requiring $pd_c(d_c-1)$ gates to generate all outputs. The stochastic check node function is therefore much less complex than the SPA one. Furthermore, its complexity scales only logarithmically in q.

Unlike binary codes, non-binary regular LDPC codes with $d_v = 2$ can be good codes [13], and the low variable node degree has the advantage of reducing the decoder complexity. A specialized stochastic algorithm called "NoX" is proposed in [16] for $d_v = 2$ codes. It significantly reduces the decoder's complexity by extending the update method based on transfer functions used in the binary RHS algorithm. The binary RHS algorithm achieves a low complexity by combining the mean tracking of stochastic streams with the conversion to the LLR domain, and using low-complexity approximations of the resulting transfer functions (see Sect. 6.5.3). The NoX algorithm for $d_v = 2$ non-binary codes also includes the VN computation in these transfer functions. This is rendered manageable by the fact that for $d_v = 2$, the VN function in (6.17) simplifies to $U_{vp}^{(t)} = \text{NORM}(L_v \times U_{p'v}^{(t-1)})$, where p and p' are the two PNs connected to the VN.

6.6.3 Results

When considering the average number of iterations required for decoding as well as the number of operations required to perform one iteration, the NoX algorithm has about the same complexity as a version of the SPA that uses the Fast Fourier Transform to reduce the complexity of the CN update. The main reason explaining that NoX does not outperform SPA is that the iterative progress of NoX is slower than SPA. This could be acceptable since the complexity of one iteration of NoX is also much less. However, a side effect of the slower progress is that larger memories are required for storing the current state of the VN circuits, which in turn increases the complexity of an update. It is likely that the complexity of NoX can be reduced by increasing the amount of information exchanged in one iteration, just like the parameter k of the binary RHS algorithm (Sect. 6.5) can be increased to transmit more information in each message, while still relying on the low-complexity stochastic CN function.

References

1. Alaghi A, Hayes JP (2013) Survey of stochastic computing. ACM Trans Embed Comput Syst 12(2s):92:1–92:19. doi:10.1145/2465787.2465794
2. Blanksby AJ, Howland CJ (2002) A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder. IEEE J Solid-State Circuits 37(3):404–412
3. Brown BD, Card HC (2001) Stochastic neural computation I: computational elements. IEEE Trans Comput 50(9):891–905
4. Chung SY, David Forney J, Richardson TJ, Urbanke R (2001) On the design of low-density parity-check codes within 0.0045dB of the Shannon limit. IEEE Commun Lett 5(2):58–60
5. Davey M, MacKay D (1998) Low-density parity check codes over gf(q). IEEE Commun Lett 2(6):165–167. doi:10.1109/4234.681360
6. Gaines BR (1967) Stochastic computing. In: Proceedings of the spring joint computer conference, 18–20 April 1967. ACM, New York, pp 149–156
7. Gaudet V, Rapley A (2003) Iterative decoding using stochastic computation. Electron Lett 39(3):299–301. doi:10.1049/el:20030217
8. Hemati S, Banihashemi A (2006) Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes. IEEE Trans Commun 54(1):61–70. doi:10.1109/TCOMM.2005.861668
9. Leduc-Primeau F, Raymond AJ, Giard P, Cushon K, Thibeault C, Gross WJ (2012) High-throughput LDPC decoding using the RHS algorithm. In: Proceedings of 2012 conference on design & architectures for signal & image processing (DASIP)
10. Leduc-Primeau F, Hemati S, Mannor S, Gross WJ (2013) Relaxed half-stochastic belief propagation. IEEE Trans Commun 61(5):1648–1659. doi:10.1109/TCOMM.2013.021913.120149
11. Naderi A, Mannor S, Sawan M, Gross W (2011) Delayed stochastic decoding of LDPC codes. IEEE Trans Signal Process 59(11):5617–5626. doi:10.1109/TSP.2011.2163630
12. Poppelbaum WJ, Afuso C, Esch JW (1967) Stochastic computing elements and systems. In: Proceedings of the fall joint computer conference, AFIPS '67 (Fall), 14–16 November 1967. ACM, New York, pp 635–644. doi:10.1145/1465611.1465696. http://doi.acm.org/10.1145/1465611.1465696
13. Poulliat C, Fossorier M, Declercq D (2008) Design of regular (2,d/sub c/)-ldpc codes over gf(q) using their binary images. IEEE Trans Commun 56(10):1626–1635. doi:10.1109/TCOMM.2008.060527
14. Qian W, Li X, Riedel M, Bazargan K, Lilja D (2011) An architecture for fault-tolerant computation with stochastic logic. IEEE Trans Comput 60(1):93–105. doi:10.1109/TC.2010.202
15. Richardson T, Urbanke R (2008) Modern coding theory. Cambridge University Press, Cambridge
16. Sarkis G, Hemati S, Mannor S, Gross W (2013) Stochastic decoding of LDPC codes over GF(q). IEEE Trans Commun 61(3):939–950. doi:10.1109/TCOMM.2013.012913.110340
17. Sharifi Tehrani S, Gross W, Mannor S (2006) Stochastic decoding of LDPC codes. IEEE Commun Lett 10(10):716–718
18. Sharifi Tehrani S, Mannor S, Gross W (2008) Fully parallel stochastic LDPC decoders. IEEE Trans Signal Process 56(11):5692–5703. doi:10.1109/TSP.2008.929671
19. Sharifi Tehrani S, Naderi A, Kamendje GA, Mannor S, Gross WJ (2009) Tracking forecast memories in stochastic decoders. In: Proceedings of IEEE international conference on acoustics, speech, and signal processing (ICASSP)
20. Sharifi Tehrani S, Naderi A, Kamendje GA, Hemati S, Mannor S, Gross WJ (2010) Majority-based tracking forecast memories for stochastic LDPC decoding. IEEE Trans Signal Process 58(9):4883–4896. doi:10.1109/TSP.2010.2051434
21. Winstead C, Gaudet VC, Rapley A, Schlegel CB (2005) Stochastic iterative decoders. In: International symposium on information theory, pp 1116–1120

Chapter 7
MP-SoC/NoC Architectures for Error Correction

Carlo Condo, Maurizio Martina, and Guido Masera

7.1 Introduction

In the last year several standards for both wired and wireless communications have been proposed. Indeed, modern terminals, such as smartphones and tablets, are equipped with different modules for ubiquitous access to internet. As argued in [38], flexibility has become a fundamental property of architectures for digital baseband processing, as it allows to support different operating modes more efficiently than simply placing several modules together and turn them on or off. Unfortunately, the high throughputs to be sustained make actual flexibility implementation a challenging task, especially in the context of channel code decoder architectures, where often decoding algorithms are both complex and iterative.

High throughput imposes to have parallel architectures made of several processing elements (PEs) connected via and appropriate communication backbone. The design of an optimized architecture for a single code is a well-established problem and has been largely investigated in the past. On the contrary, depending on the amount of required flexibility, the problem of designing efficient architectures becomes increasingly challenging. In this sense we can have two main classes of flexibility: (a) architectures that support only one family of codes (such as turbo codes or LDPC codes) or (b) architectures that support more families of codes (e.g., both turbo and LDPC codes). In order to achieve interoperability among different standards, the second class is the most interesting one. Indeed it contains three sub-classes of particular interest: (1) architectures that support different codes within a standard (e.g., both turbo and LDPC codes for the WiMAX standard), (2) architectures that support different codes within more standards (e.g., both turbo

C. Condo • M. Martina • G. Masera (✉)
Politecnico di Torino, Corso Duca degli Abruzzi, 24 Torino, Italy
e-mail: Carlo.Condo@polito.it; Maurizio.Martina@polito.it; Guido.Masera@polito.it

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_7

and LDPC codes for WiFi, WiMAX and LTE standards), (3) architectures that support different codes and that are *future proof* or *fully flexible*, which means that these architectures can accommodate any code, provided that the amount of available resources is enough. This last case is the most challenging one as adding flexibility can lead to less optimized solutions with respect to cases (1) and (2). Nevertheless, all the aforementioned cases require to design both flexible PEs and flexible communication structures.

Flexibility in the PEs can be achieved adding some multiplexers to select among a fixed set of alternatives or via programmable architectures such as Application Specific Instruction-set Processors (ASIPs). On the other hand, flexibility in the communication structure can be obtained resorting to crossbars, shuffling-networks, or borrowing results from the general NoC paradigm.

7.2 Flexibility in the Communication Structure

Some works in the literature propose flexible and efficient communication structures either for turbo [16] or LDPC codes [4] relying on crossbars, or shuffling-networks. Unfortunately, these solutions are usually tailored around specific characteristics of some particular classes of codes, thus not being used or extended in the case of a general approach. Stemming from the NoC paradigm [12], Neeb et al. [36] proposed an interesting NoC-based approach to enable flexible and efficient interconnection among P processing elements (PE) in parallel turbo decoder architectures. According to [43] this approach, where the network structure is used to connect PEs belonging to the same Intellectual Property (IP), is referred to as intra-IP NoC. The literature shows that the intra-IP NoC approach has been mainly studied in the context of (1) parallel turbo decoder architectures [25, 27, 30], (2) semiparallel LDPC code decoder architectures [14,31,43], (3) flexible turbo/LDPC code decoder architectures [6, 31]. In the following we will assume that each PE is made of a processing core and a memory (see Fig. 7.1a), where the processing core implements LLR (Logarithmic Likelihood Ratio) computation/updating operations and the memory is used both for the storage of data coming from the network and as an input buffer for the processing core.

IntraIP-NoCs tailored around iterative channel decoder architectures exhibit some specific features that are summarized in the following. Since the data block is partitioned into P subblocks and each PE is assigned one subblock, all nodes exchange nearly the same amount of data. Moreover, idle times in the PEs are reduced to maximize the throughput. As a consequence, the nodes exchange the data at a fairly constant rate. Besides, due to the random nature of interleavers for turbo codes and $\mathbf{H}$ matrices for LDPC codes, the pattern of connection among PE has little adjacency. One of the main consequences of these properties is that the injected traffic load tends to be uniform both in time and space. Furthermore, since decoding algorithms are iterative, minimizing the latency of the single iteration is of paramount importance to achieve high throughput. Thus, simple routing algorithms

and routing circuits have to be designed. To this purpose the designer can take advantage of the uniform traffic load and the homogenous nature of the nodes. It is worth noting that, as the amount of packets injected into the network is known and depends on P and on the data block size N_f, flow control is not required. Nevertheless, it is important to define the packet injection rate r, namely the number of packets injected in the network by a PE in a clock period. This parameter can simultaneously model two phenomena: (1) the use of different clock frequencies for the PEs and the network (usually the maximum clock frequency of PEs is lower than the one used for the network). (2) the fact that a PE may not produce a valid packet at each clock cycle. Finally, if the permutation laws of the turbo code interleavers and the **H** matrices of the LDPC codes considered by a decoder are known, then the traffic patterns can be derived by off-line analysis. This leads to the so called *zero-overhead* networks introduced in [43] and further developed in [25,30], where all the routing information is precalculated by the means of a simulator, as the one in [24], leading to significant simplification in the architecture of the nodes.

The structure of the packets is common to all the proposed architectures, namely each packet is made of a header and a payload. The header contains routing information, such as the identifier of the destination PE that, for P processing elements, requires $\log_2(P)$ bits. The payload contains both a refined LLR and the memory location where the LLR has to be stored. Even if the number of bits used to represent LLRs impacts on the bit-error-rate performance of a code, 8 bits is a typical value. On the other hand, the memory location is represented on $\lceil \log_2(N_f/P) \rceil$ bits.

The NoC-based approaches proposed in the literature for iterative channel decoder architectures can be divided into two main categories depending on the type of network they employ: (1) indirect networks [30,31] (2) direct networks [6,14,25]. In the following both approaches are described.

7.2.1 Indirect Networks

The most popular indirect network topologies proposed in the literature for iterative channel decoder architectures are Multistage Interconnection Networks (MINs) [10]. MINs rely on the cascade of several switch stages, each of which is often referred to as router in the literature of NoC-based iterative channel decoder architectures. On the other hand, PEs are not part of the routers, but they serve as both the input and the output of the network (see Fig. 7.1a). Examples of such networks are Clos, Benes, Omega, and Butterfly networks [10]. The number of stages and the number of switches per stage changes with the topology, as an example the Butterfly network is made of $\log_2(P)$ stages where each stage relies on $P/2$ switches.

It is worth noting that the degree of connectivity obtained using these networks is larger than that achievable with shared bus structures, but it is smaller than the one reachable with crossbars. As a consequence, when two or more paths connecting

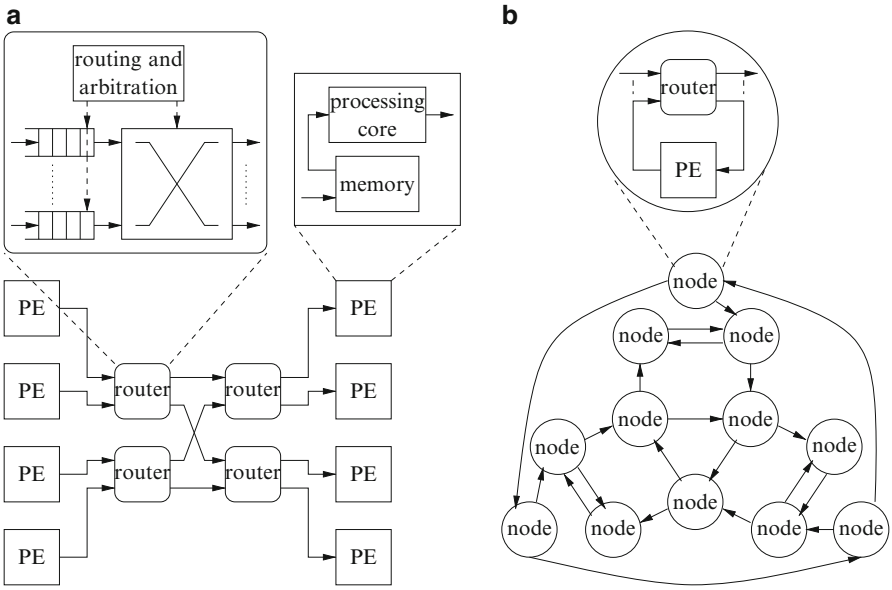

Fig. 7.1 Network and node architecture: (**a**) indirect network example, (**b**) direct network example

source/destination PEs cross, conflicts arise. A simple solution to handle conflicts is to include two FIFO queues in the router architecture. Thus, each router relies on a 2×2 switch, two input FIFOs for storing conflicting packets and a routing and arbitration block that serves the input queues with a round-robin policy and generates all the required control signals as depicted in the top part of Fig. 7.1a.

The most significant results in the literature concerning indirect networks relate to Butterfly and Benes networks [27,30]. The Butterfly network features logarithmic diameter and a highly scalable recursive structure. Moreover, routing in the Butterfly network is very simple as the bits of the destination PE are used to select the output port at each stage of the network. On the other hand, this network has no path diversity, namely there exist only one path connecting two PEs; as a consequence, conflicts can arise frequently.

An alternative solution to partially solve conflicts is the Benes network. The Benes network has a larger path diversity than the Butterfly network, even if its diameter is about two times the diameter of the Butterfly network. However, conflicts are avoided with the Benes network only if the source/destination pattern is a permutation of the PE identifiers. Unfortunately, this is not always the case of turbo and LDPC codes. An interesting solution to solve this problem is the one described in [30], where a time-division-multiple-access architecture is proposed. The idea is to exploit the deterministic characteristics of the traffic to assign a time slot to each packet, that is the cycle when the packet will be injected into the network. As a consequence, the network inputs are scheduled such that at each cycle there

is only one packet per output port. Then, the routing information is precalculated off-line and stored in the packet header, leading to an architecture belonging to the zero-overhead class.

The analysis presented in [30] refers to $r = 0.2$ to avoid network congestion for both Butterfly and Benes networks. Moreover, implementation results shown in [27] on a 130 nm standard cell technology for $P = 16$ show that the Benes network performs better than the Butterfly one both in terms of occupied area and maximum achievable frequency/throughput. The Butterfly network occupies 0.75 mm^2 and achieves a clock frequency of 345 MHz corresponding to a throughput of 138 Mb/s. On the other hand, the Benes network occupies only 0.48 mm^2 (−35 %) and achieves a clock frequency of 381 MHz (+10.4 %) and a throughput of 152 Mb/s (+10 %).

7.2.2 Direct Networks

Direct networks analyzed in the literature [14, 25, 31] rely on P nodes whose architecture can be seen as a generalization of the one proposed for indirect networks. Indeed, each node contains one PE, D input, and D output connections from/to the network, where D is the topology degree, and a routing and arbitration block (see Fig. 7.1b). Thus, each node relies on $D+1$ input FIFOs and a $(D+1) \times (D+1)$ switch.

In [31] binary de-Bruijn topologies are proposed to design a flexible interconnection structure for an LDPC and a turbo/LDPC code decoder respectively. The binary de-Bruijn network is indeed very appealing to support the communications of a multiprocessor turbo/LDPC decoder as it features good path diversity properties. In addition, de-Bruijn networks have logarithmic diameter that leads to small latencies, and a recursive structure that makes them highly scalable. The analysis presented in [31] highlights that flexibility comes at the expense of a larger area with respect to indirect networks. Indeed, scaling the results to a 130 nm standard cell technology for $P = 16$ to compare with indirect network results, we obtain that the binary de-Bruijn network requires 0.64 mm^2 for a 244 MHz clock frequency, that is about +33 % of area and −36 % of clock frequency.

The works in [14, 25] extend the analysis to other topologies. Both [14, 25] analyze 2D-mesh topologies, showing interesting speed-up and scalability results. Moreover, in [25] several topologies are compared for network degrees ranging from 2 to 4. The analysis corroborates the results presented in [31] by showing that logarithmic topologies as the (generalized) de-Bruijn and Kautz ones are the most suited to achieve both flexibility and high performance. Moreover, experimental results presented in [25] for a wide number of cases prove that zero-overhead architectures require less area with respect to the ones where hardware resources for the routing algorithm are employed. Considering $P = 16$, $r = 0.33$ and a clock frequency of 200 MHz, zero-overhead architectures for generalized Kautz topologies achieve in the best case a throughput of about 102 Mb/s with an area of 0.74 mm^2 for $D = 2$ and about 103 Mb/s with 0.92 mm^2 for $D = 4$.

7.3 Review of Flexible Decoding Architectures

In this section, the state of the art on flexible channel decoders is briefly reviewed for the two most interesting classes mentioned in Sect. 7.1, namely architectures supporting different codes within more standards (multi-standard architectures) and architectures that in principle accommodate any code (fully flexible architectures).

7.3.1 Multi-Standard Architectures

While the same type of code can be used in various standards, code size, rate, and construction method can vary greatly between one application and the other. A first step towards flexibility in channel decoders is then guaranteeing support for a single type of code, taking in account the code parameters employed in different standards.

A flexible turbo code decoder architecture is devised in [19], where a decoder targeting both 3GPP-LTE and Mobile WiMAX standards is proposed. The different nature of the considered turbo codes (single-binary in 3GPP-LTE and double-binary in Mobile WiMAX) is tackled by means of bit-to-symbol and symbol-to-bit conversions [17], addressed later in Sect. 7.4.1 of this chapter, that allow almost complete memory sharing. Moreover, a novel dual-mode interleaver is introduced that reduces the overhead relative to the implementation of the native ARP [15] and QPP [45] interleavers respectively. The resulting multi-standard decoder can reach up to 186 Mb/s with moderate complexity.

Multi-standard support is achieved in [1] by means of partially parallel turbo code decoder based on ASIPs. Support is given for 3GPP-LTE, WiMAX, and DVB-RCS standards, reaching a maximum throughput of 170 Mb/s and yielding good efficiency. Different ASIPs are used for the in-order and interleaved phases of the decoding process, and the complexity and latency are kept in check via smart information exchange networks and pipeline idle time minimization.

The flexible LDPC decoder described in [43] is one of the first work to consider a Network-on-Chip (NoC) as a possible interconnection structure. Together with the design of processing elements, the design of the application-specific NoC is carried out in detail: it is shown that many of the characteristics of general purpose NoCs are not necessary, thus reducing the overhead commonly associated with complex interconnections. The complete decoder is arranged on a toroidal mesh topology.

A decoder with extremely good error correction capabilities is shown in [39]. It is designed targeting LDPC convolutional codes that can be obtained from quasi-cyclic LDPC codes. Again, the regular structure of these codes allows for easy design of largely parallel structure. Up to 2 Gb/s are obtained with a frequency of 100 MHz.

7.3.2 *Fully Flexible Architectures*

Few recent works have focused on extending the concept of flexible decoders not only to multiple codes, but also to multiple code types, providing complete support for whole standards.

The work in [33] describes the design of a multi-standard turbo/LDPC decoder based on ASIPs and the sharing of memories between the two code types. Each ASIP has two separate datapaths, one for each decoding mode: eight ASIPs are instantiated and connected via a simple but flexible interconnection network that can be reconfigured when switching decoding mode to adapt to the different communication patterns. Also in [2] is presented an ASIP-based decoder that includes convolutional code decoding as well. This work is characterized by extremely small area occupation and high achievable frequency that helps to meet most throughput requirements for the considered standards.

The works presented in [11, 35, 42] exploit commonalities between turbo and LDPC decoding to design a unified architecture for multiple standards. By viewing an LDPC code as a series of turbo codes [23], the BCJR algorithm can be applied to both code types. The shared datapath and memories result in an overall area much lower than separate dedicated decoder implementations.

7.4 Improving the Efficiency of NoC-Based Decoders

It has been shown in the previous section that NoCs are extremely flexible interconnection structures, able to guarantee connectivity among all the nodes of the network. NoC-based decoders can in fact support up to numerous standards at the same time [2, 7, 11, 41]. However, flexibility comes at the cost of increased interconnection complexity and additional latencies that impact heavily on both throughput and energy consumption, reducing the overall decoder efficiency. NoC-induced latencies in particular can be a major obstacle in extending the support of a decoder to multiple standards, whereas excessive energy consumption and area occupation limit the set of applications in which the decoder can be employed. To reduce the impact of NoCs on the decoder efficiency, various techniques can be applied.

7.4.1 *Energy Reduction Techniques*

The area occupation of a NoC usually consists of 20–40 % of the total decoder area [9, 26] that results in a proportional impact on the power consumption. It is consequently desirable to reduce the complexity of the NoC as much as possible. A set of interesting techniques is considered in [26], aimed at limiting the width of

the channels between the nodes of NoC-based turbo code decoders by reducing the number of concurrently transmitted extrinsic metrics and the number of quantization bits of each metric.

Let us define $\lambda^{ext}[u]$ as the extrinsic information associated to symbol u, obtained at the output of the SISO decoder at each half-iteration. All $\lambda^{ext}[u]$ must be sent through the NoC according to the interleaving rule: the width of each NoC channel must accommodate, depending on the nature of the considered code, the transmission of one or more concurrent extrinsic metrics. In Double-Binary turbo codes like the ones used in WiMAX, where each symbol u is composed of two bits, the transmitted $\lambda^{ext}[u]$ is an array of three elements. However, following the architecture presented in [18], it is possible to switch between symbol-level $\lambda^{ext}[u]$ and bit-level $\lambda^{ext}[A]$ and $\lambda^{ext}[B]$. The width of the transmitted packet is thus reduced by 1/3, together with the width of the memories in which $\lambda^{ext}[u]$ is stored. The consequent area reduction is much more consistent than the increment brought by the bit-to-symbol and symbol-to-bit conversion units, necessary since the BCJR algorithm implemented in SISOs requires symbol-level metrics. The conversion operation, however, introduces an overall Bit Error Rate (BER) performance loss of about 0.2 dB. A further step towards the reduction of the area occupation of the NoC and of the decoder in general can be taken by addressing the quantization of $\lambda^{ext}[A]$ and $\lambda^{ext}[B]$. Applying the Pseudo-Floating-Point (PFP) representation suggested in [37], it is possible to reduce the quantization without incurring in significant performance degradation. The idea is based on the fact that bits within the representation of extrinsic metrics play a different role in the decoding process according to their weight, as highlighted also in [40, 44]. Analyzing the binary representation of $\lambda^{ext}[A]$ and $\lambda^{ext}[B]$ from the most significant bit to the least significant bit, it is possible to detect the first zero-to-one or one-to-zero transition. This signals the starting bit of the significant part of the extrinsic metric. Finally, an equal number of bits is assigned to the significant parts of both $\lambda^{ext}[A]$ and $\lambda^{ext}[B]$, alongside a common shift factor used at reception to reconstruct the extrinsic values. It is show in [26] that the joint application of both methods can reduce the total NoC channel width of more than 50 % that, depending on the router architecture and decoder structure, can save up to 40 % of the total NoC area.

In iteratively decoded codes like turbo and LDPC codes, the energy required for the decoding of a frame is directly proportional to the number of performed iterations: limiting their number is then an effective way of limiting the energy consumption. A few Early Stopping Criteria (ESCs) can be found in the literature for turbo codes [13, 28], but many more have been proposed for LDPC codes, e.g., [3, 22, 29]. Since LDPCs have a straightforward method of identification of correct decoding, existing ESCs focus on the early identification of situations in which the decoding is going to fail. This is usually achieved via observation of the evolution of metrics throughout different iterations. Most ESCs offer limited flexibility, and their performance can vary substantially when applied to different codes. The Multi-Standard-ESC (MSESC) proposed in [8] has been designed to adapt on-the-fly to code parameters and channel conditions, thus being particularly fit for flexible multi-standard decoders. During each iteration, by comparing the

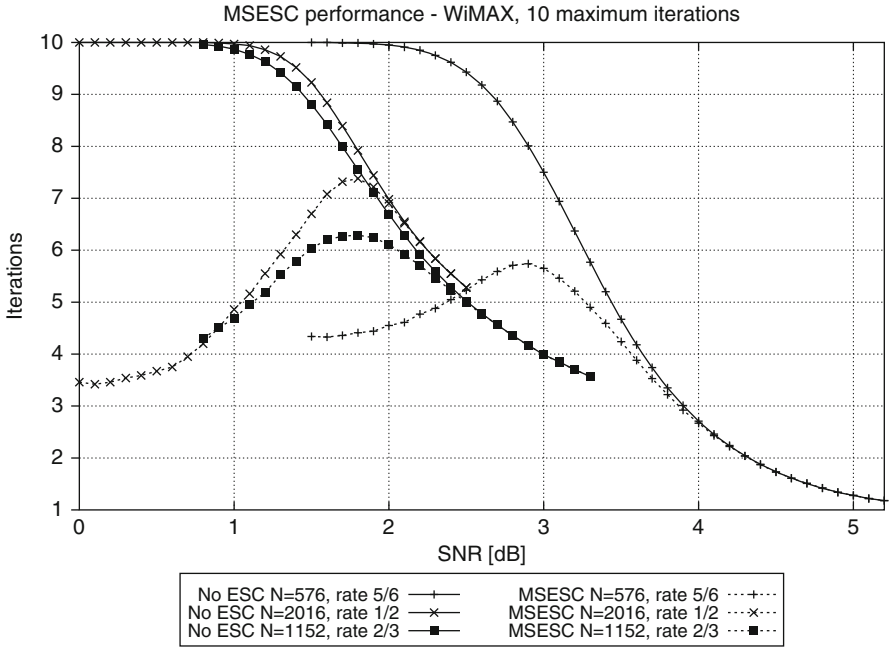

Fig. 7.2 MSESC performance for WiMAX codes

Check-Node-Mean-Magnitude metric [3] and the syndrome value against three thresholds $T1$, $T2$, and $T3$, MSESC decides on the stopping of the decoding process, effectively identifying both impossible decoding and insufficient available iterations for successful decoding. The thresholds can be computed as follows:

$$T1 = \mathrm{M} \cdot 2^{-6} \quad T2 = \mathrm{M} \cdot 2^{bits_f} \quad T3 = \mathrm{M} \cdot 2^{-5} = T1 \cdot 2 \tag{7.1}$$

where M is the number of rows in the LDPC parity check matrix and $bits_f$ is the number of fractional bits assigned to the representation of LLRs. The dependency of $T1$, $T2$, and $T3$ on code parameters and design choices allows MSESC to maximize the number of saved iterations without affecting the BER performance with a very wide range of codes. Figure 7.2 shows the number of performed iterations for the decoding of three WiMAX codes against the Signal-to-Noise Ratio (SNR), with and without MSESC. The codes are characterized by different block lengths and rates, but MSESC maintains its effectiveness with all of them. At low SNR successful decoding is improbable, and after very few iterations the decoding is stopped, once impossible decoding is identified. The performed iterations start to rise with the SNR, reaching a peak in the early waterfall region, where successful decoding is likely but still requires a high number of iterations. Finally, at high SNR, early impossible decoding is very rare, and MSESC is almost always deactivated. The implementation of MSESC reported in [8] shows that the simple logic involved

in the on-the-fly computation of threshold values guarantees a very small power consumption overhead that is compensated by the reduced number of iterations. Compared to decoders implementing no ESCs or alternative solutions, MSESC shows energy saving ranging from 10 to 90 %.

7.4.2 Latency Reduction Techniques

The latency introduced by NoCs with respect to less flexible interconnection structures is unsustainable for most decoders, due to the strict throughput requirements of many communication standards. In fact the required throughput imposes an upper bound on the duration of the decoding process and, consequently, on delivery time of each message injected in the NoC: high message injection rates, traffic, and collisions often result in late messages. These, even in small percentages, can be disastrous for the decoder BER performance.

A common choice to avoid late messages is to stall the decoder between decoding phases, waiting for the delivery of all information. Unfortunately, a straightforward implementation of this approach severely limits the achievable throughput. A possible solution has been studied in [26, 32] for turbo codes and [5] for LDPC codes, where the reliability of the exchanged information is evaluated through threshold-based measures. In [32], reliable information is characterized by a small enough difference between a-priori and extrinsic information, while in [5] reliable LLRs are those with a large enough magnitude. In both cases, if the information is deemed reliable it is not exchanged anymore, reducing the traffic on the NoC and possibly the total delivery time. This technique can be particularly effective in presence of large networks with light traffic patterns, where up to 20 % throughput gains have been observed.

Stalling the decoder, however, is often unfeasible, especially in decoders with heavy traffic loads, in which the transmission times are already long and additional latency cannot be sustained. A possible approach in these situations can be the artificial reduction of the rate of packet injection r in the NoC by setting the frequency of the network at a multiple of that of the processing elements, that is reducing the value of r. While effective, this solution is extremely expensive in terms of power consumption. A set of alternative methods have been studied in [9], aimed at the reduction and optimization of NoC traffic. The Hard Importance (HI) method is very similar to the threshold-based reliability measures devised in [32] and [5], while the Soft Importance (SI) allows to discard low-importance packets in case they collide once they have been injected in the NoC. Both techniques offer significant advantages in heavy traffic conditions, reducing the percentage of late messages and the NoC switching activity. The remaining late messages, however, still cause severe performance degradation in the majority of cases. Traffic optimization is based on the fact that an estimate of the number of clock cycles available for message delivery can be sent together with the information and can be updated in the NoC. This field is used as a priority indicator: most urgent messages

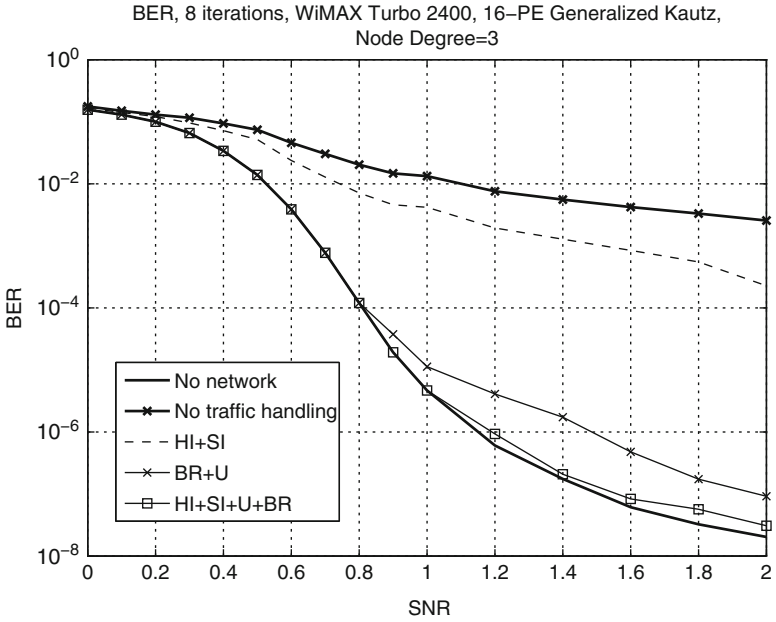

Fig. 7.3 BER performance of WiMAX code on NoC with combinations of traffic reduction and optimization

are favored in case of collisions. Exploiting this Urgency (U) priority field, NoC buffers can be changed from FIFOs to urgency-ordered buffers (Buffer Reordering, BR) so that urgent messages are always the first to be served in routers. These two traffic optimization methods are extremely effective in guaranteeing on-time message delivery. Figure 7.3 plots the BER performance of a WiMAX turbo code mapped on a 16 PE Kautz NoC, under the influence of different traffic handling techniques. The "No network" represents the achievable performance, with 0 % late updates, while "No traffic handling" has been obtained without the application of any of the described methods. It can be noticed how traffic reduction alone (HI+SI) is not able to bring the BER to acceptable levels. On the contrary, very good results are shown by traffic optimization (BR+U) and even more performance improvement is observed when all four techniques work together. Implementation of the four techniques on the decoder presented in [7] results in a 13 % area overhead and a 15 % power consumption reduction, thanks to the lower NoC frequency [9].

7.5 Dynamic Reconfiguration

Extensive research has been conducted in the last few years on flexible multi-mode and multi-standard decoding architectures, and large efforts have been spent to limit the impact of flexibility on achievable throughput and dissipated energy. However,

the issue of dynamic reconfiguration has been often neglected in this context. Surprisingly enough, most of flexible decoders available in the open literature [2, 11, 25, 27, 27, 31, 35, 42, 43] efficiently work on a wide range of codes, but do not support the run-time switch from one code to another.

Very few contributions have considered the fundamental requirement of rapidly reconfiguring the architecture for a new decoding task and the related overheads in terms of area and energy [7,20,21,34]. Change of decoding mode, standard, or code parameters requires not only hardware support, but also memory initialization and specific controls; moreover, since in many standards a code switch can be issued as early as one data frame ahead, reconfiguration time is also a major need.

In this section, a detailed analysis of the reconfiguration issue is carried out and the different kinds of reconfiguration are described, together with alternative solutions and trade-offs.

7.5.1 The Reconfiguration Task

We can distinguish between three levels of dynamic reconfiguration, associated to growing levels of flexibility and complexity:

1. intra-standard reconfiguration,
2. inter-standard reconfiguration,
3. reconfiguration to a new code.

The first case does not refer to a change of standard, but simply to a change of communication mode: the decoder has to switch between two codes of the same family, belonging to the same standard. New and old codes can have different length and code rate, but they typically share many other elements, such as, the decoding algorithm and the iteration control. The similarity between the two codes can be exploited to simplify the configuration process and reduce the amount of data to be updated. In the second case, the switch is between two codes with potentially large differences with respect to each other: in addition to code length and rate, other important features may be changed, such as, the decoding algorithm or the permutation law. The reconfiguration process tends to be more complex and it involves a larger amount of data. The last case is of interest for fully flexible or future-proof decoders, which are able to dynamically adapt to any code in the considered families (e.g., turbo and LDPC codes). This flexibility is also extended to codes not yet known at the time of decoder design, provided that the length is within limits that depend on the size of the internal memories allocated to store received frames.

The actual complexity of the dynamic reconfiguration and its impact on the overall decoder in terms of additional occupied area and dissipated energy depend on the amount of internal data to be updated when switching to a new code and on the available time to complete this operation.

In modern wireless communication systems, a code switch can be issued as early as one data block ahead and the decoder must be reconfigured for the new data block while it is still running on the previous one. Therefore, a time efficient reconfiguration technique is mandatory. The worst-case reconfiguration latency can be simply expressed as

$$L_{rec} = \frac{N_f R}{T} \tag{7.2}$$

where N_f is the bit length of the current block, R is the code rate, and T is the specified throughput. It can be easily seen from (7.2) that L_{rec} tends to become very short in the case of short block, low code rate, and high throughput. Considering current wireless communication standards, it has a typical value of a few µs. As an example, in the 3GPP LTE standard, 188 different information block sizes are specified, ranging from 40 to 6,144, the lowest code rate is $1/3$ and the throughput can be as high as 450 Mbit/s for the down-link; the peak throughput is specified for the largest block, while it scales for shorter blocks. Using the largest block, (7.2) gives a latency of 13.6 µs. The requirement on L_{rec} is expected to become even more severe in LTE Advanced, which specifies a throughput as high as 1 Gbit/s.

Estimating the amount of data to be updated at the reconfiguration of the decoder is rather difficult, as it is heavily dependent on the specific adopted architecture and on the kind of addressed reconfiguration.

An FPGA based decoder is intrinsically reconfigurable, as the running decoder can be stopped at any time and a new design can be loaded into the programmable device to support a completely different code. Current FPGA technology allows for run-time partial configuration, which means that the decoder structure can be modified or extended, while it is serving the current decoding task. These dynamic reconfiguration techniques offer a large potential flexibility, ranging from the simple switch between two similar codes, decoded by architectures that share a high percentage of hardware resources, up to the load of a complete new decoder, running a different decoding algorithm. However, FPGA configuration is still a slow and energy inefficient process, which involves the uploading of tens of Mbytes, therefore it is hardly acceptable for flexible decoding applications, with severe speed and power consumption constraints.

Large flexibility is also offered by ASIP-based decoders, which are basically customized processors with specialized datapath and instruction set. In this kind of decoder, the dynamic reconfiguration can be viewed as a context switch, where instruction memory and internal registers are dynamically loaded with new contents when the switch to a new code is requested. In [21], the amount of configuration data for a multi-ASIP decoder is evaluated as equal to about 1 kbits, due to both processor instructions (around $2/3$ of the total) and parameters ($1/3$). Proper hardware resources are necessary to support the context switch (e.g., data busses, shadow memory, and registers).

As a further alternative, which was explored in [21] and [7], the single processing element can be implemented in the form of a parameterized dedicated architecture,

which only needs a few configuration bits, as there is no instruction memory . If the decoder flexibility is limited to a small number of standards and both interleaving laws and parity check matrices are algorithmically generated by means of simple parameterized hardware components, then the whole configuration becomes very easy, fast, and involves a reduced amount of data.

Finally, the decoder can be based on an application specific NoC, with parameterized processing elements and routing elements [7]. This fully flexible solution supports the third level of configuration mentioned above and allows for dynamic switching between any couple of codes, including codes with a structure of either interleaver or parity check matrix that cannot be generated algorithmically, or is not known at the design time. In this case, the amount of data that need to be updated is significantly larger, because it includes the management of the routing. Basically, for each generated LLR, its destination node in the NoC must be available to properly forward the LLR. As a consequence, when the decoder is configured for a size N code, the number of bits to be uploaded tends to grow as $O(N\lceil log_2 N\rceil)$, where N is the codes size and $\lceil log_2 N\rceil$ bits are used to represent each location address. For example, to support the LTE codes, more than 150,000 reconfiguration bits are necessary to update the NoC routing.

From these rough evaluations, one can see that the reconfiguration process implies quite a large throughput of data moved to the decoder. Therefore, an efficient organization of the configuration process is of utmost importance to confine the related complexity and energy overheads.

7.5.2 Reconfiguration of ASIP Based Decoders

The problem of dynamically reconfiguring an ASIP based turbo decoder and the development of an efficient hardware implementation is addressed in [20, 21, 34]. It is assumed that each received block is associated to a specific configuration and that the loading of the configuration for a new block is performed during the processing of the current block.

An already available multi-mode and multi-standard turbo decoder [33] is initially considered and a set of modifications are applied in order to enable the dynamic switch between different codes. The original decoder architecture is organized around two sets of ASIPs interconnected via a Butterfly Network on Chip, where each set corresponds to a component decoder. Each ASIP unit includes ten pipeline stages and it is associated to several memories, used to store input channel LLR values, extrinsic information, state metric, instructions, and configuration data. The required interleaving/deinterleaving addresses are generated algorithmically and make use of a set of parameters which depend on the specific code to be supported and therefore are part of the configuration. From the complete list of configuration data, the information to be updated at each code switch is classified into four categories:

1. component decoder dependent parameters,
2. identical parameters for all ASIPs,
3. ASIP dependent parameters,
4. parameters required only by the last ASIP of a component decoder (for tail bits).

A low latency configuration process is then developed exploiting both multicast and broadcast mechanisms. Two multicast operations are used to update the first class of parameters, one multicast operation is necessary for parameters shared among all ASIPs, and multiple unicast transfers are exploited to upload the ASIP dependent parameters. Moreover, the ASIP configuration load has been significantly reduced by adopting a unified program, which works for all considered codes, including double-binary turbo codes. Finally, the internal configuration memory has been extended to store multiple configurations at the same time: this allows to save time in the switch between frequently used codes.

The decoder has been enriched with a dynamic reconfiguration infrastructure that supports unicast, multicast, and broadcast transfers. The adopted solution consists of a master-slave 26 bit bus-based structure. A master unit initially receives the configuration data, which are then moved to one or multiple slave units, based on the required type of transfer. A dedicated unit allows to select at run-time the target destination ASIPs. Finally, configuration data are moved from the slave units to their final destinations.

This configuration infrastructure and the related protocol controller have been implemented using both FPGA and ASIC technologies. The worst case latency is lower than 10 µs in the case of FPGA implementation, while 1 µs is reached with a 65 nm ASIC technology, allowing for a 500 MHz clock frequency. The corresponding area overhead is around 2 % of the decoder.

The described solution allows to handle the configuration process in two different ways. In the first approach, the configuration data is generated off-line for all possible codes and stored in a global memory. At every code switch, a configuration manager simply reads the new configuration data from the global memory and send it to the decoder, by means of the dedicated infrastructure. This static solution is functionally simple, but requires a large global memory to support a large set of codes and it is not compatible with any extension. The second approach is dynamic and allows for the run-time generation of new configurations. The single bus reconfiguration architecture, applied to a multi-ASIP decoder, is shown in Fig. 7.4a, where a reconfiguration interface (RI) connects each the shared bus to each ASIP and the reconfiguration data can be provided by either a memory or a reconfiguration manager.

Although the described reconfiguration technique is tailored around a specific ASIP based decoder that only supports turbo codes in WiMAX and LTE standards, it can be easily extended to include more codes. However, since the reconfiguration infrastructure uses a shared bus, this solution is expected to become less efficient if the number of ASIPs or PEs to be configured in the decoder is significantly larger than shown in [21].

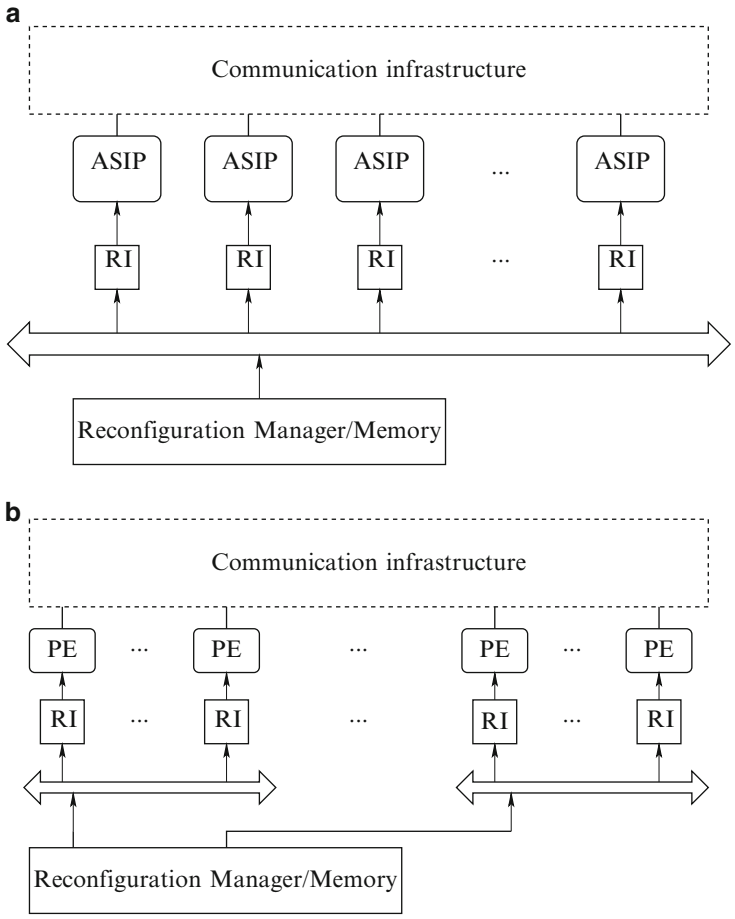

Fig. 7.4 Reconfiguration architectures: (**a**) single bus architecture, (**b**) multiple bus architecture

7.5.3 Reconfiguration of NoC Based Decoders

For the decoding architecture described in Sect. 7.3.2, the reconfiguration task affects the content of the location memory, which stores the destination addresses for each processed message, and a few parameters, such as check node degrees and the SISO window size. We assume that the whole set of reconfiguration data are saved in a distributed configuration memory (CM), allocated in each PE of the decoder. As reconfiguration must occur within the decoding of the previous block, the NoC interconnects cannot be exploited to update the CM in each processing element. Instead, N_b buses are dedicated to the configuration task, and each bus serves P/N_b PEs. The use of multiple buses allows to increase the bandwidth and to handle larger amount of reconfiguration data. This high level architecture is shown

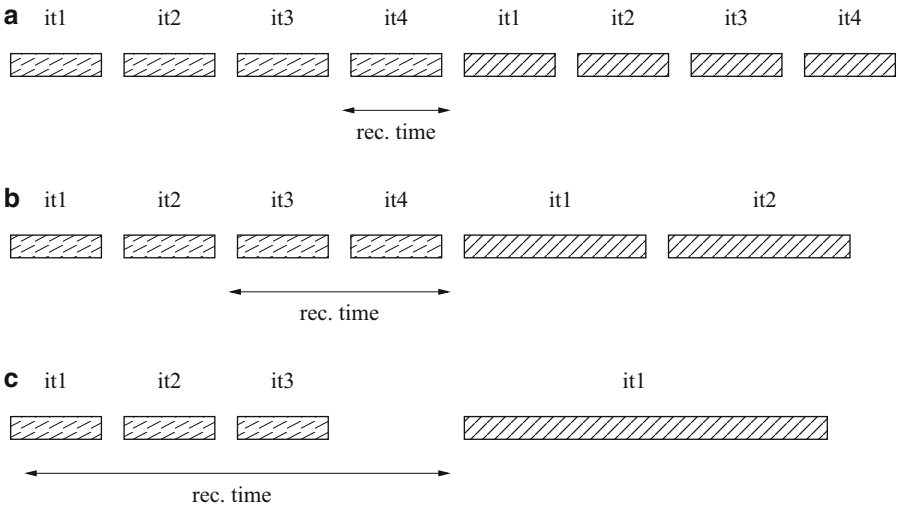

Fig. 7.5 Examples of reconfiguration. Reconfiguration can be overlapped with a single decoding iteration (**a**), with two iterations (**b**), or with a number of iterations (**c**)

in Fig. 7.4b, where multiple buses (N_b) concurrently update the decoding PEs and reconfiguration data are taken from a memory or from a manager, as in part (a) of the figure.

The CM is managed as a circular buffer, where two sets of configuration data can be present at the same time: the one necessary for the decoding of the current block, and a new one, to be used when the decoder will switch to a new code. Proper read and write pointers allow to separate old and new configurations. Limiting the size B of the CM is of interest, because it impacts on area and energy consumption. To this purpose, multiple strategies can be adopted. The first idea is to partially overlap the configuration process with the decoding of one or more blocks (N_o) of the current code. A second option is to overlap an additional part of the configuration process with the first decoding iteration of the newly loaded code. A final option is skipping one or multiple iterations (N_{sk}) while decoding the last received block: the saved time can be used to complete the loading of CM before starting the decoding of the new block. Differently from the other solutions, the last one has an impact on the error correction performance, because of the skipped iterations.

A few examples of reconfiguration are shown in Fig. 7.5, where distinct patterns are used to indicate blocks decoded with different codes. In part (a) of the Figure, the switch to a new code is considered, after four decoding iterations with the previous code: as new and old codes have similar size, the reconfiguration time is close to the time length of a single iteration. Therefore, in this case, $N_o = 1$ and $N_{sk} = 0$. In part (b), the destination code is longer and reconfiguration time needs to be extended to cover more than one iteration. Finally, in part (c), the destination code is much longer than the current code, therefore, one decoding iteration is skipped ($N_{sk} = 1$).

In general, given a set of standards and turbo or LDPC codes to be supported, the reconfigurable decoder can be designed by selecting a proper combination of the four mentioned parameters (the number of busses N_b, the buffer size B, the number of overlapping blocks N_o, and the number of skipped iterations N_{sk}), with the purpose of minimizing the overall complexity and matching the required latency at the same time, with minimum impact on the error correction performance.

Intuitively, the global latency decreases with larger N_b, because more PEs can be reconfigured at the same time. Moreover, increasing B reduces the need for skipped iterations and overlapping blocks. When the destination code in the reconfiguration process is larger than the current one, the uploading of CM may require either larger N_o or larger N_{sk}.

A complete analysis of both intra- and inter-standard reconfiguration is presented in [7], where several codes from a large variety of standards (WiFi, DVB-RCS, WiMAX, CMMB, DTMB, 3GPP-LTE, and HPAV) are considered. Starting from the architecture proposed in [6], an extended NoC based fully flexible decoder is obtained with shared memory and interconnect resources. Moreover configuration busses, memories, and control logic have been included to support dynamic switch between different codes. The decoder has been synthetized with a 90 nm CMOS standard cell technology, with 22 PEs running at 200 and 170 MHz when configured to support LDPC and turbo decoding respectively; 300 MHz is the target clock frequency for the NoC. The whole set of turbo and LDPC codes included in the mentioned standards are fully supported and achievable throughput meets the specifications until ten iterations for LDPC codes and eight for turbo codes. Post place & route estimated area is 3.42 mm^2 and peak dissipated power is 120 mW.

The reconfiguration process has been tested on every possible couple of codes in the seven considered standards. From this large set of experiments, the following conclusions can be drawn.

- The intra-standard reconfiguration is possible with no need for skipped iterations ($N_{sk} = 0$) in all standards, except LTE, which requires $N_{sk} > 0$ in 6.8 % of possible switches between the 188 specified codes.
- Inter-standard reconfiguration is also possible with no skipped iterations when the new code is not belonging to LTE, CMMB, or DTMB standards.
- Inter-standard reconfigurations towards LTE, CMMB, or DTMB requires $N_{sk} > 0$ in a percentage of possible cases which ranges between 5 and 77 %. Moreover, for all considered cases, $N_{sk} \leq 3$.

The impact of the reconfiguration process and particularly of the skipped iterations on the decoder performance has been assessed by means of BER simulations, taking into account the probability that a reconfiguration is required as a consequence of channel fading. Based on different fading scenarios, the channel conditions may change at a rate f_{ch} between 10 and 150 Hz [7]. The actual reconfiguration probability can be calculated as

$$P_{rec} = \frac{f_{ch} RN}{T} \tag{7.3}$$

where the ratio T/N is the number of coded frames received per unit time. P_{rec} is shown to range between 0.25 and 0.3 % in presence of a fast moving receiver, while it remains under 0.15 % in the other cases. The effect of $N_{sk} \leq 3$ on BER performance in such conditions is negligible and lower than 0.05 dB for all considered codes and standards.

Acknowledgements This work has been partially supported by the Newcom# project.

References

1. Al-Khayat R, Baghdadi A, Jezequel M (2012) Architecture efficiency of application-specific processors: a 170Mbit/s 0.644mm2 multi-standard turbo decoder. In: 2012 International symposium on system on chip (SoC), pp 1–7. doi:10.1109/ISSoC.2012.6376368
2. Alles M, Vogt T, Wehn N (2008) FlexiChaP: a reconfigurable ASIP for convolutional, turbo, and LDPC code decoding. In: 2008 5th International symposium on turbo codes and related topics, pp 84 –89
3. Cai Z, Hao J, Sethakaset U (2008) Efficient early stopping method for LDPC decoding based on check-node messages. In: Proceedings of Asilomar conference on signals, systems and computers, pp 466–469. doi:10.1109/ACSSC.2008.5074448
4. Chen X, Lin S, Akella V (2010) QSN-a simple circular-shift network for reconfigurable quasi-cyclic LDPC decoders. IEEE Trans Circuits Syst II 57(10):782–786
5. Condo C, Masera G (2011) A flexible NoC-based LDPC code decoder implementation and bandwidth reduction methods. In: 2011 Conference on design and architectures for signal and image processing (DASIP), pp 1 –8
6. Condo C, Martina M, Masera G (2012) A network-on-chip-based turbo/ldpc decoder architecture. In: Design, automation and test in Europe conference and exhibition, pp 1525–1530
7. Condo C, Martina M, Masera G (2013) VLSI implementation of a multi-mode turbo/LDPC decoder architecture. IEEE Trans Circuits Syst I 60(6):1441–1454
8. Condo C, Baghdadi A, Masera G (2014) Energy-efficient multi-standard early stopping criterion for low-density-parity-check iterative decoding, Communications, IET. 8(12):2171, 2180. doi:10.1049/iet-com.2013.0869
9. Condo C, Baghdadi A, Masera G (to appear) Reducing the dissipated energy in multi-standard turbo and LDPC decoders. Circuits Syst Signal Process
10. Duato J, Yalamanchili S, Ni L (2003) Interconnection networks: an engineering approach. Morgan Kaufmann, Los Altos
11. Gentile G, Rovini M, Fanucci L (2010) A multi-standard flexible turbo/LDPC decoder via ASIC design. In: International symposium on turbo codes & iterative information processing, pp 294–298
12. Guerrier P, Greiner A (2000) A generic architecture for on-chip packet-switched interconnections. In: Design, automation and test in Europe conference and exhibition, pp 250–256
13. Hagenauer J, Offer E, Papke L (1996) Iterative decoding of binary block and convolutional codes. IEEE Trans Inf Theory 42(2):429–445. doi:10.1109/18.485714
14. Hu W-H, Chen C-Y, Bahn JH, Bagherzadeh N (2012) Parallel low-density parity check decoding on a network-on-chip-based multiprocessor platform, Computers & Digital Techniques, IET. 6(2): 86, 94. doi:10.1049/iet-cdt.2010.0177
15. IEEE Std 802.16, part 16: air interface for fixed broadband wireless access systems (2004)
16. Kim B, Yoo I, Park IC (2013) Low-complexity parallel QPP interleaver based on permutation patterns. IEEE Trans Circuits Syst II 60(3):162–166

17. Kim JH, Park IC (2008) A 50Mbps double-binary turbo decoder for WiMAX based on bit-level extrinsic information exchange. In: IEEE Asian solid-state circuits conference, pp 305–308. doi:10.1109/ASSCC.2008.4708788
18. Kim JH, Park IC (2009) Bit-level extrinsic information exchange method for double-binary turbo codes. IEEE Trans Circuits Syst II 56(1):81–85
19. Kim JH, Park IC (2009) A unified parallel radix-4 turbo decoder for mobile WiMAX and 3GPP-LTE. In: IEEE custom integrated circuits conference, 2009 (CICC '09), pp 487–490. doi:10.1109/CICC.2009.5280790
20. Lapotre V, Murugappa P, Gogniat G, Baghdadi A, Diguet JP, Bazin JN, Hubner M (2013) A reconfigurable multi-standard asip-based turbo decoder for an efficient dynamic reconfiguration in a multi-asip context. In: 2013 IEEE computer society annual symposium on VLSI (ISVLSI), pp 40–45. doi:10.1109/ISVLSI.2013.6654620
21. Lapotre V, Murugappa P, Gogniat G, Baghdadi A, Hubner M, Diguet JP (2013) Stopping-free dynamic configuration of a multi-asip turbo decoder. In: 2013 Euromicro conference on digital system design (DSD), pp 155–162. doi:10.1109/DSD.2013.24
22. Li J, You XH, Li J (2006) Early stopping for LDPC decoding: convergence of mean magnitude (CMM). IEEE Commun Lett 10(9):667–669. doi:10.1109/LCOMM.2006.1714539
23. Mansour M, Shanbhag N (2002) Turbo decoder architectures for low-density parity-check codes. In: IEEE Global Telecommunications Conference, 2002 (GLOBECOM '02), vol 2, pp 1383–1388. doi:10.1109/GLOCOM.2002.1188425
24. Martina M (2010) Turbo NOC: network on chip based turbo decoder architectures. Downloadable at www.vlsilab.polito.it/~martina
25. Martina M, Masera G (2010) Turbo NOC: a framework for the design of network-on-chip-based turbo decoder architectures. IEEE Trans Circuits Syst I 57(10):2776–2789
26. Martina M, Masera G (2013) Improving network-on-chip-based turbo decoder architectures. J Signal Process Syst 73(1):83–100
27. Martina M, Masera G, Moussa H, Baghdadi A (2011) On chip interconnects for multiprocessor turbo decoding architectures. Elsevier Microprocess Microsyst 35(2):167–181
28. Moher M (1993) Decoding via cross-entropy minimization. In: IEEE global telecommunications conference, 1993 (GLOBECOM '93), including a communications theory mini-conference. Technical program conference record, IEEE in Houston, vol 2, pp 809–813. doi:10.1109/GLOCOM.1993.318192
29. Mohsenin T, Shirani-Mehr H, Baas B (2011) Low power LDPC decoder with efficient stopping scheme for undecodable blocks. In: Proceedings of IEEE international symposium on circuits and systems
30. Moussa H, Muller O, Baghdadi A, Jezequel M (2007) Butterfly and Benes-based on-chip communication networks for multiprocessor turbo decoding. In: Design, automation and test in Europe conference and exhibition, pp 654–659
31. Moussa H, Baghdadi A, Jezequel M (2008) Binary de Bruijn on-chip network for a flexible multiprocessor LDPC decoder. In: Design automation conference, pp 429–434
32. Muller O, Baghdadi A, Jezequel M (2006) Bandwidth reduction of extrinsic information exchange in turbo decoding. Electron Lett 42(19):1104–1105. doi:10.1049/el:20062209
33. Murugappa P, Al-Khayat R, Baghdadi A, Jezequel M (2011) A flexible high throughput multi-ASIP architecture for LDPC and turbo decoding. In: Design, automation and test in Europe conference and exhibition, pp 1–6
34. Murugappa P, Lapotre V, Baghdadi A, Jezequel M (2013) Rapid design and prototyping of a reconfigurable decoder architecture for qc-ldpc codes. In: 2013 International symposium on rapid system prototyping (RSP), pp 87–93. doi:10.1109/RSP.2013.6683963
35. Naessens F, Bougard B, Bressinck S, Hollevoet L, Raghavan P, Van der Perre L, Catthoor F (2008) A unified instruction set programmable architecture for multi-standard advanced forward error correction. In: Proceedings of IEEE workshop on signal processing systems, pp 31–36. doi:10.1109/SIPS.2008.4671733

36. Neeb C, Thul MJ, Wehn N (2005) Network-on-chip-centric approach to interleaving in high throughput channel decoders. In: IEEE international symposium on circuits and systems, pp 1766–1769
37. Park SM, Kwak J, Lee K (2008) Extrinsic information memory reduced architecture for non-binary turbo decoder implementation. In: IEEE vehicular technology conference, 2008 (VTC Spring 2008), pp 539–543
38. Polydoros A (2008) Algorithmic aspects of radio flexibility. In: IEEE international symposium on personal, indoor and mobile communications, pp 1–5
39. Sham CW, Chen X, Lau F, Zhao Y, Tam W (2013) A 2.0 Gb/s throughput decoder for QC-LDPC convolutional codes. IEEE Trans Circuits Syst I Regul Pap 60(7):1857–1869. doi:10.1109/TCSI.2012.2230506
40. Singh A, Boutillon E, Masera G (2008) Bit-width optimization of extrinsic information in turbo decoder. In: 2008 5th International symposium on turbo codes and related topics, pp 134–138. doi:10.1109/TURBOCODING.2008.4658686
41. Studer C, Benkeser C, Belfanti S, Huang Q (2011) Design and implementation of a parallel turbo-decoder ASIC for 3GPP-LTE. IEEE J Solid State Circuits 46(1):8–17
42. Sun Y, Cavallaro JR (2010) A flexible LDPC/Turbo decoder architecture. J Signal Process Syst 64(1):1–16
43. Vacca F, Moussa H, Baghdadi A, Masera G (2009) Flexible architectures for LDPC decoders based on network on chip paradigm. In: Euromicro conference on digital system design, pp 582–589
44. Vogt J, Ertel J, Finger A (2000) Reducing bit width of extrinsic memory in turbo decoder realisations. Electron Lett 36(20):1714–1716. doi:10.1049/el:20001177
45. 3gpp ts 36.211, evolved universal terrestrial radio access (e- utra): physical channels and modulation, version 8.4.0 (2008)

Chapter 8
ASIP Design for Multi-Standard Channel Decoders

Purushotham Murugappa, Amer Baghdadi, and Michel Jezequel

8.1 Flexibility Requirement in Channel Decoder Design

Mobile wireless connectivity is a key feature of a growing number of devices, which will count soon in tens of billions, from laptops, tablets, cell phones, cameras, and other portable devices. The variety of applications and traffic types will be significantly larger than today and will result in more diverse requirements. These applications are driving the creation of new transmission techniques and design architectures that push the boundaries to achieve high throughput, low latency, area, and power efficient implementations.

Channel coding is one of the key techniques that enable reliable high throughput data transfer through unreliable wireless channels. However, as a large variety of channel coding options and flavors are specified in existing and emerging digital communication standards, there is an increasing need for flexible implementations. In fact, several powerful error correction techniques exist today, each suitable for specific application parameters (frame size, transmission channel, signal-to-noise ratio, bandwidth, etc.). Considering the emerging multi-mode and multi-standard applications, as well as the increasing interest for Software Defined Radio (SDR) and Cognitive Radio (CR) applications, combination of multiple error correction techniques becomes mandatory. Table 8.1 shows a representative set of mobile wireless standards to highlight their differences in data rates and channel encoding schemes. The most commonly used error correcting codes in these standards are convolutional codes (CC), turbo codes (SBTC: single-binary turbo codes and DBTC: double-binary turbo codes), and low-density parity-check (LDPC) codes.

P. Murugappa • A. Baghdadi (✉) • M. Jezequel
Institut Mines-Telecom, Telecom Bretagne, CNRS Lab-STICC,
Technopôle Brest-Iroise, 29238 Brest, France
e-mail: Purushotham.Murugappa@telecom-bretagne.eu; Amer.Baghdadi@telecom-bretagne.eu;
Michel.Jezequel@telecom-bretagne.eu

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_8

Table 8.1 Representative set of mobile wireless standards and related channel codes and parameters

Standard	Codes	Rates	States	Frame size (bits)	Throughput (Mbps)
UMTS	CC	1/4..1/2	256	.. 504	<1
	SBTC	1/3	8	.. 5,114	.. 2
HSDPA	SBTC	1/2–3/4	8	.. 5,114	.. 14.4
CDMA2000	CC	1/6 .. 1/2	256	.. 744	<1
	SBTC	1/5–1/2	8	.. 20,730	.. 2
IEEE-802.11n (WiFi)	CC	1/2..3/4	64	.. 4,095	.. 450
	LDPC	1/2–5/6	-	.. 1,620	.. 450
IEEE802.16e (WiMAX)	CC	1/2–5/6	64	.. 864	.. 75
	DBTC	1/2–3/4	8	.. 4,800	.. 75
	LDPC	1/2–5/6	-	.. 1,920	.. 75
DVB-S2	LDPC	1/4–9/10	-	.. 64,800	.. 90
DVB-RCS	DBTC	1/3–6/7	8	.. 1,728	.. 2
3GPP-LTE	SBTC	0.33–0.95	8	.. 6,144	.. 150

In this context, and at the receiver side, it is well known that channel decoding is one of the most computation, communication, and memory intensive, and thus, power-consuming component. Channel decoder design has been extensively investigated during the last few years and several implementations have been proposed. Some of these implementations succeeded in achieving high throughput for specific standards through the adoption of highly dedicated architectures that work as hardware accelerators. However, these implementations do not take into account flexibility and scalability issues. Particularly, this approach implies the allocation of multiple separated hardware accelerators to realize multi-standard systems, which often result in poor hardware efficiency. Furthermore, it implies long design-time which is no more compatible with the severe time-to-market constraints and the continuous development of new standards and applications.

More recently, several contributions have been proposed targeting flexible, yet high throughput, implementations of channel decoders. The flexibility varies from supporting different modes of a single communication standard to the support of multi-standards multi-modes applications. Other implementations have even proposed to increase the target flexibility to the support of different channel coding techniques. As a matter of fact, a knowledge gap is growing quickly in the last few years between the need for flexibility in the digital base-band processing segment of modern communication systems, and the actual availability of flexible while efficient hardware support to the quest for reconfigurability. The main reason that determines this growing gap is related to the poor area and energy efficiency of flexible solutions proposed till now and the huge increase of non-recurrent engineering (NRE) costs in the production of dedicated integrated circuits for specific applications (ASIC) with new semiconductor technologies.

Towards the target of filling the above mentioned gap, it becomes crucial to define and develop efficient and high performance flexible channel decoder architecture models for emerging and future digital communication systems. The need of optimal solutions in terms of performance, area, and power consumption is increasing and cannot be neglected against flexibility. In common understanding, a blind approach towards flexibility results in some loss in optimality. The objective of recent initiatives in this context is to unify flexibility-oriented and optimization-oriented approaches. The main goal is to deliver enablers and building block solutions in order to derive, for a specific application need, the best balance between a highly flexible solution and a specifically optimized one.

This chapter illustrates the use of Application-Specific Instruction-set Processor (ASIP) models and tools as one of the main recent design approaches towards this target, enabling the designer to scale and freely tune the flexibility/performance trade-off as required by the considered application requirements. Related contributions are emerging rapidly seeking to improve the resulting architecture efficiency in terms of performance/area and in addition to increase the flexibility support. The chapter starts by introducing the ASIP design approach and the recently available methodologies and tools. Then we illustrate the application of this design approach through two ASIP design examples for multi-standard turbo decoding with different architecture alternatives and design objectives. Considering the increased flexibility requirement for the support of multiple types of channel codes, the chapter presents then a short review of related state-of-the-art contributions to illustrate the trend towards the use of ASIP-based design approach in this context. The chapter ends with a summary to highlight the main conclusions and to introduce few related research perspectives.

8.2 ASIP Design Approach

In traditional design of flexible hardware architectures, the flexibility is incorporated by the expert designer through the use of initialization parameters loaded from a configuration memory or input ports of the design. The architecture description involves manual design of a finite state machine (FSM) that controls the different design units of the pipeline taking into account the various supported parameters. But when the number of flexibility parameters increases, the design and validation of such parametrized control logic become more and more complicated.

On the other hand, instruction-set based processors provide inherently high flexibility in terms of control logic design through software programmability. Their architectures have evolved dramatically in the last couple of decades [1] from microprogrammed FSMs to dynamically reordered parallel pipelines with multiple levels of parallelism and optimization techniques. More recently, the trend to design customized instruction-set processors has been made successful given the development of new design methodologies and tools. Such tools enable the designers to specify a customizable processor in weeks rather than months.

Two main approaches can be distinguished in this regard [1]. The first allows to configure and extend a predefined core or architectural skeleton with application-specific hardware resources [2–4]. Additional instructions can be defined with corresponding functional pipelines and application-specific register files or memory interfaces. The second approach allows to support architects in the effort to design from scratch a completely custom processor with its complete tool chain (compiler, simulator, etc.). Ideally, it uses an architectural description language (ADL) to describe the architecture of the design from which all tools and the synthesizable description of the core can be generated [5, 6]. All these approaches fall under the name of customizable processors and often are referred as ASIP for Application-Specific Instruction-set Processors (ASIPs).

Designing an efficient ASIP requires a deep analysis of the target flexibility parameters and algorithm variants in order to devise the adequate algorithm and architecture choices that enable efficient hardware resource allocation and sharing. The application-specific instruction-set can then be derived accordingly. The general ASIP design methodology comprises the following four steps: (1) analysis of target application and underlined algorithms with respect to flexibility requirements, (2) derivation of algorithm/architectural choices for the target flexibility and parallelism degree, (3) design of basic building blocks of the ASIP with efficient resource usage and sharing, and (4) design of the complete architecture of the ASIP including the dedicated instruction-set, datapath, pipeline stages, memory banks, and I/O interfaces.

One of the most mature ASIP development tools is Processor Designer which was first commercialized by the startup LISATek and then acquired by CoWare in 2003 and later by Synopsys in 2010. It was developed with a simulator-centric view [1] and uses a C-like language (LISA) for design description of programmable architectures and their peripherals and interfaces. The language syntax provides a high flexibility to describe the instruction set of various parallelism techniques with explicit modeling of both data-path and control. The usage of a centralized description of the processor architecture ensures the consistency of the Instruction-Set Simulator (ISS), software development tools (compiler, assembler, and linker, etc.), and RTL (Register Transfer Level) implementation, minimizing the verification and debug effort. The benefits of using this design approach are illustrated in the rest of this chapter through examples related to multi-standard channel decoders design.

8.3 ASIP-Based Decoders for Turbo Codes

This section illustrates the ASIP design flow for the implementation multi-standard turbo decoders. After a brief introduction on the hardware-efficient decoding algorithm for turbo codes, a short review of the related state-of-the-art implementations is given. Then, two ASIP design examples with different architecture alternatives and design objectives are presented together with performance analysis and comparisons.

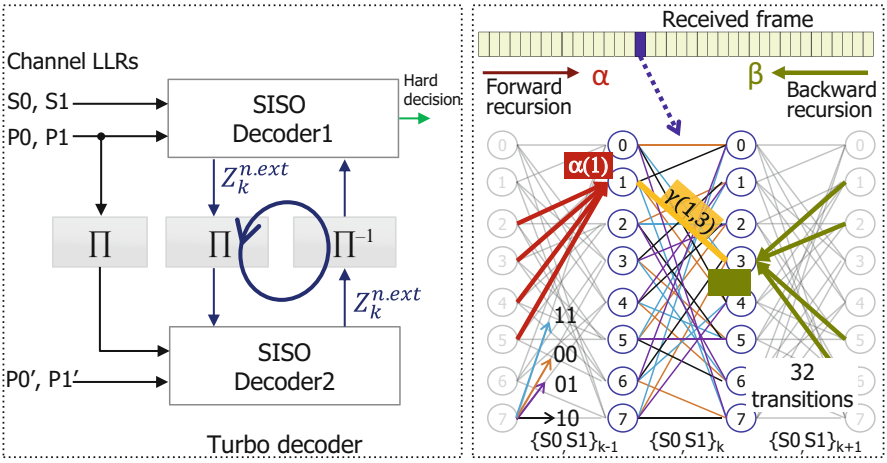

Fig. 8.1 Typical turbo codes decoder structure and an 8-states DBTC trellis

8.3.1 Hardware-Efficient Decoding Algorithm

Typical turbo decoder structure consists of two Soft Input Soft Output (SISO) component decoders iteratively exchanging extrinsic information via an interleaving (Π) and deinterleaving (Π^{-1}) functions as illustrated in Fig. 8.1. The SISO decoders often implement the max-log-MAP algorithm [7], which represents the hardware-efficient version of the maximum a posteriori (MAP) decoding using the BCJR algorithm known to achieve the minimal symbol error probability. In order to explain briefly the underlined max-log-MAP computations, let us consider the 8-state DBTC code of WiMAX standard, represented by its trellis in Fig. 8.1. For each received double binary symbol $\{S0,S1\}_k$, the SISO decoder computes first the branch metrics $\gamma_k(s',s)$ which represent the probability of a transition to occur between two trellis states (s': starting state, s: ending state). Then the SISO decoder runs the forward and backward recursions over the trellis (Fig. 8.1). The forward state metrics $\alpha_k(s)$ of the kth symbol are computed recursively using those of the $(k-1)$th symbol and the branch metrics of the corresponding trellis section. Similarly for the backward state metrics $\beta_k(s)$ which corresponds however to the backward recursion (traversing the trellis in the reverse direction).

$$\alpha_k(s) = \max_{s'}(\alpha_{k-1}(s) + \gamma_k(s',s))$$
$$\forall(s',s = 0,1,..7) \tag{8.1}$$

$$\beta_k(s) = \max_{s'}(\beta_{k+1}(s) + \gamma_k(s',s))$$
$$\forall(s',s = 0,1,..7) \tag{8.2}$$

Finally, the extrinsic information Z_k^{ext} of the kth symbol is computed for all possible decisions $(00,01,10,11)$ using the forward state metrics, the backward state metrics, and the extrinsic part of the branch metrics as formulated in the following expressions:

$$Z_k^{apos}(d(s',s)=x) = \max_{(s',s)/d(s',s)=x}(\alpha_{k-1}(s)+\gamma_k(s',s)+\beta_k(s)) \quad \forall(x=00,01,10,11) \tag{8.3}$$

$$Z_k^{ext}(d(s',s)=x) = Z_k^{apos}(d(s',s)=x) - \gamma_k^{int_x}(s',s) \quad \forall(x=00,01,10,11) \tag{8.4}$$

The extrinsic information can be normalized by subtracting the minimum value in order to reduce the related storage and communication requirements, thus only three extrinsic information values should be exchanged for each symbol.

$$Z_k^{n.ext}(d(s',s)=x) = Z_k^{ext}(d(s',s)=x) - min(Z_k^{ext}(d(s',s)=x)) \quad \forall(x=00,01,10,11) \tag{8.5}$$

Executing one forward–backward recursion on all symbols of the received frame in the natural order completes one half iteration. A second half iteration should be executed in the interleaved order to complete one full turbo decoding iteration. Once all the iterations are completed (usually 6–7 iterations), the turbo decoder produces a hard decision for each symbol $Z_k^{hard\ dec.} \in (00,01,10,11)$.

For SBTC, the use of the trellis compression (Radix-4) [8] represents an efficient parallelism technique and allows for efficient resource sharing with a DBTC SISO decoder, as two single binary trellis sections (two bits) can be merged into one double binary trellis section.

8.3.2 State-of-the-Art on Turbo Codes Decoders

Since the discovery of turbo codes in 1993 [9], considerable amount of research works has been targeting practical VLSI implementations of turbo decoders. Using mainly the low complexity max-log-MAP algorithm, many contributions have been proposed targeting diverse design objectives in terms of area efficiency, energy efficiency, flexibility, scalability, and high throughput. Among the initial efforts in this context we can cite the examples of [10–13]. Authors of [14] present an overview of the implementation aspects related to turbo decoding architectures

discussing the issues of quantization and iteration stopping criteria. Several joint algorithmic/architecture optimization techniques have been investigated in [15]. One of the first ASIC implementations was presented in [16] achieving a throughput of 50 Mbps with ten decoding iterations and an operating frequency of 1 GHz.

Turbo codes have been since widely adopted in wireless communication standards like CDMA2000, 3GPP, LTE, WiMAX, DVB-RCS, etc. Many implementations have succeeded to meet the low throughput requirements of the early standards (e.g., CDMA2000 and 3GPP) using advanced DSP architectures [17–19] or customizable processors [20]. However, the scalability of such implementations is limited by the block interleavers specified in these standards which cause memory access contentions when targeting higher sub-block parallelism degree. The work presented in [21], targeting LTE, allows multiple SISO decoders (1, 2, 4, or 8) to concurrently process frame sub-blocks and integrates a three-stage network to connect the multiple memory and SISO decoder modules. Implemented in 90 nm CMOS technology, the design achieves a throughput of 129 Mbps with eight iterations and occupies an area of 2.1 mm^2 while exhibiting a power consumption of 219 mW and supporting the maximum specified frame size of 6,144 bits. Targeting Gbps throughputs, a recent work [22] has proposed an LTE compliant turbo decoder architecture with 32 parallel SISO decoders. A throughput of 2.15 Gbps is achieved with an on chip area of 7.1 mm^2 using 65 nm CMOS technology. Other works have proposed the additional support of DBTC specified in WiMAX standard. As an example, the work in [23] presents an architecture which supports all DBTC parameters specified in WiMAX and 18 frame sizes of the 188 specified in LTE. A high area efficiency is achieved by supporting only those frame sizes with interleaving properties that can be easily mapped to the extrinsic exchange paths of DBTC. Another example is the parameterized architecture of [24] which supports both turbo codes modes (DBTC and SBTC) and achieves a high throughput of 187 Mbps with eight parallel MAP decoders.

Recently, ASIP design approaches have been explored in this application context. Such an architecture model enables the designer to freely tune the flexibility/performance trade-off and thus to meet the increasing flexibility requirement efficiently. Furthermore, the well-established design methodology and the mature available tools enable short design-time. The rest of this section will illustrate this trend through the presentation of two ASIP design examples with different architecture alternatives and design objectives.

8.3.3 TurbASIP: Highly Flexible ASIP

One of the first proposed ASIPs for flexible turbo decoding has been presented in [25]. The main design objective for this ASIP, namely TurbASIP, was to explore the effectiveness of the newly proposed ASIP-design tools in terms of quality of the generated HDL code and flexibility limitations when targeting this class of applications. To that end, the target flexibility was set very high to investigate

the support of any convolutional code trellis of DBTC and SBTC. The number of trellis states, however, is limited to 8 states in DBTC mode and 16 states in SBTC mode, as typically adopted in existing wireless communication standards. Another design objective for this ASIP was the investigation and the exploitation of the various parallelism techniques available for turbo decoding. Most of the available parallelism techniques [26] have been exploited. This includes:

1. Metric level parallelism, which concerns the processing of all metrics involved in the decoding of each received symbol inside a SISO decoder. It exploits the inherent parallelism of the trellis structure (as the same operations related to the computation of γ, α, β and the extrinsic information (γ^{ext}) should be repeated for all the trellis transitions) and the parallelism of the MAP computations (parallel computation of the metrics α, β, and extrinsic information γ^{ext}).
2. SISO decoder level parallelism, which exploits the use of multiple SISO decoders, each processing a sub-block of the same frame in natural or interleaved orders. At this level, parallelism can be applied either on sub-blocks and/or on component decoders (shuffled decoding).
3. Turbo decoder level parallelism, which simply duplicates whole turbo decoder to process iterations and/or frames in parallel. Nevertheless, this parallelism level is too area-expensive (all memories and computation resources are duplicated) and presents no gain in frame decoding latency, and thus it was not considered.

8.3.3.1 Overview of TurbASIP Architecture

Considering these design objectives, adequate algorithm/architectural choices have been derived to meet the target flexibility and parallelism degree. This step is followed by the design of basic building blocks of the ASIP with efficient resource usage and sharing. Figure 8.2 presents the overall architecture of TurbASIP.

The architecture exploits the metric level parallelism through the use of two hardware recursion units to process concurrently the forward and backward computations following a butterfly scheduling scheme. Each recursion unit includes a state metric calculation matrix with 32 *adder* nodes and 8 max operator nodes to compute all trellis transitions related metrics in parallel. It includes in addition the required hardware resources to compute branch metrics and the hard decision with multiple registers to store intermediate results. The target high flexibility degree is supported through the use of high number of multiplexers enabling to handle any convolutional code trellis structure. A main design choice in the ASIP architecture for turbo decoding concerns the memory organization. Besides the quantization aspects which impact the width of the memory words, the parallelism degree imposes the use of multiple memory banks and the flexibility requirement (supported frame lengths) impacts directly the size of these banks. Figure 8.2 illustrates how particular is the memory organization for such application and consequently how adequate is an ASIP design approach compared to general-purpose instruction-set processor

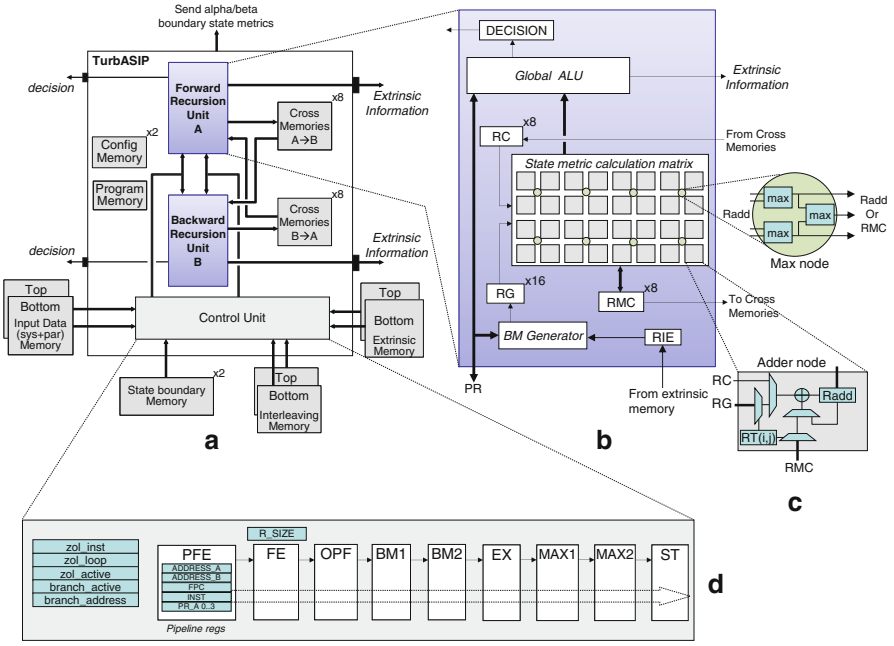

Fig. 8.2 TurbASIP architecture: (**a**) overview, (**b**) recursion unit, (**c**) adder node, (**d**) pipeline

architectures. Besides the *Input memories* (duplicated due to the parallelism implied by the choice of butterfly scheduling scheme) several memories are required: *Extrinsic memories* to store extrinsic messages exchanged between SISO decoders, *Interleaving memories* to store interleaving addresses, *Cross memories* to store intermediate metrics between the two executed forward/backward recursions, *State boundary memories* to store boundary state metrics for metric initialization between neighboring sub-blocks, *Program memory* to store program instructions, and *Config memory* to store parameters related to the supported codes (trellis definition, block size, number of iterations, etc.).

Exploiting the second level of parallelism, sub-block parallelism and shuffled decoding, is done through the instantiation of multiple ASIPs in each component decoder to process concurrently the received frame, partitioned in sub-blocks, in natural and interleaved order. The ASIP should feature the required input/output interfaces and the flexibility to support any interleaving rules and parallelism degree is met through the use of application-specific network-on-chip (NoC) architectures [27]. The efficiency of this parallelism level and related analysis of the architecture scalability are addressed in [26].

8.3.3.2 Pipeline Architecture and Instruction-Set

The devised pipeline structure for TurbASIP is illustrated in Fig. 8.2. The first three stages correspond to instruction address generation, instruction fetch, and instruction decode. The branch metrics are calculated in the two pipeline stages *BM1* and *BM2*. The *EX* stage executes the adder nodes to compute the 32 state metric LLRs as defined in (8.1) and (8.2). The *MAX1* stage executes the reconfigurable max operators and computes the α and β state metric LLRs by taking the maximum of the *RADD* registers column wise. During extrinsic generation phase, the max operators are reconfigured to calculate the maximum of the four *RADD* registers along the horizontal direction of the state metric calculation matrix. Thus, two partial a posteriori LLR values are generated per row. The final a posteriori LLRs are generated by the max operators in the *MAX2* pipeline stage that calculates the maximum of the eight partial a posteriori LLRs to produce the four a posteriori symbol LLRs γ^{apos}. The last stage of the pipeline (*ST*) generates the extrinsic information from a posteriori LLRs and the intrinsic LLRs γ^{int} as given by (8.4).

Appropriate instruction-set is designed according to the devised algorithm/architecture choices and target flexibility described above. An excerpt of assembly code for the first iteration of the turbo decoding execution is presented in Listing 8.1. It starts by initializing the ASIP configuration registers setting the mode by reading the trellis configuration registers (instruction *SET_CONF*). Next, the size in symbols of half of the window (SET_SIZE), the scaling factor (SET_SF), number of windows in the sub-block (SET_WINDOW_N), and the initial window counter (SET_WINDOW_ID) are set. The value 6 used by the *SET_SF* command allows to scale the input extrinsic LLRs by 0.875 before computing the branch metrics. The initial window boundary state registers of the first window (RMC) are initialized to be uniform, i.e., all states are initialized to same equiprobable value. Zero overhead loop (*ZOLB*) instruction is devised to execute lines (@26) and (@27) 32 times, i.e., half of window size (as set by the SET_SIZE instruction). The instruction at line (@26) implements the left side of the butterfly decoding scheme. During the first iteration of the shuffled decoding the extrinsic memories are uninitialized, hence the *WITHOUT_EXT* field of the instruction specifies not to use extrinsic information in the branch metric calculations. The *ADD M* field of the instruction forces the adder nodes to do 32 state metric calculations ($\alpha + \gamma$ or $\beta + \gamma$) in the *EX* stage of the pipeline. The *COLUMN* field configures the max nodes to calculate the maximum column-wise. An idle cycle is introduced via *NOP* instruction at line (@27) due to data dependency. Once the left side of butterfly decoding schedule is completed for a window, the right side of the butterfly schedule is processed by executing the instructions at lines (@30) and (@33) 32 times. The instruction *EXT ADD i LINE* computes the extrinsic information. The field *LINE* configures the max nodes in the *MAX1* pipeline stage to calculate the maximum row-wise. Additionally, this instruction activates *MAX2* stage to finalize the computation of the extrinsic information and *ST* stage to fetch the corresponding interleaved/de-interleaved addresses for NoC packetization.Two other short portions of assembly

Listing 8.1 Example of TurbASIP assembly code for the first turbo decoding iteration

```
1   .text
2   ;set configuration wimax
3           SET_CONF 0
4           SET_CONF 1
5           SET_CONF 2
6           SET_CONF 3
7   ;set half window size
8           SET_SIZE 32
9   ; set the scale factor 6=0.875
10          SET_SF 6
11  ;set number of windows and initial
12  ;window id counter to zero
13          SET_WINDOW_N 2
14          SET_WINDOW_ID 0
15  ;set boundary initialization of
16  ;RMC registers as uniform
17          SET_RMC UNIFORM, UNIFORM
18          REPEAT UNTIL _loop0 1 times
19          NOP
20  ;repeat instructions between _RW1 to _CW1
21  ;and between _CW1 to_LW1 SET_SIZE times
22          ZOLB _RW1,_CW1,_LW1
23          W_LD_BETA 0
24  ;configure max units to take max column wise.
25  ;store results in RMC
26  _RW1: DATA LEFT WITHOUT_EXT ADD M COLUMN
27  _CW1: NOP
28  ;configure max units to take max column wise.
29  ;store results in RMC
30          DATA RIGHT WITHOUT_EXT ADD M COLUMN
31  ;configure max units to take max rowwise.
32  ;to calculate extrinsic
33  _LW1: EXT WITHOUT_EXT ADD i LINE
34  ;increment the window number
35          EXC_WINDOW
36          NOP
37          NOP
38  _loop0: NOP
```

code, presented in [28], are required to process the subsequent iterations of the turbo decoding which differ by using extrinsic LLRs during branch metric calculation and computing the hard decision in the last iteration.

8.3.3.3 Results and Architecture Efficiency Definition

The devised architecture for TurbASIP was described in LISA and validated using Synopsys Processor Designer tool and a bit-true software reference model of the algorithm. The generated VHDL code has been validated and synthesized targeting 65 nm general purpose CMOS technology. The high flexibility of TurbASIP allows to support any trellis description at run-time and quantization-related configurations at design-time. Considering a quantization of 4 bits for input LLRs and 8 bits for extrinsic information, one TurbASIP occupies a total area of $\sim$0.19 mm^2 (logic and memories) for a maximum clock frequency of 500 MHz. A multi-ASIP configuration with 4 ASIPs for each component decoder (8 in total) occupies a total area of $\sim$1.53 mm^2 (including a NoC based on Butterfly topology).

The following expression allows to compute the achieved turbo decoding throughput:

$$Throughput_{TurbASIP} = \frac{2 \times Bits_{sym} \times f_{clk} \times N_A/2}{N_{instr} \times N_{iter}} \tag{8.6}$$

where N_A = number of ASIPs, N_{instr} = number of instructions to process two symbols, $Bits_{sym}$ = number of bits per symbol, f_{clk} = clock frequency, N_{iter} = number of iterations.

As TurbASIP needs on average 4 instructions per iteration (2 for left butterfly and 2 for right butterfly) in order to process two double binary symbols, thus we have $N_{instr} = 4$ and $Bits_{sym} = 2$ in DBTC mode. Considering a 4x4 TurbASIP turbo decoder, and using the above expression where $N_A = 8$, f_{clk} =500 MHz, and N_{iter} =6, the achieved throughput is around 333 Mbps.

In order to evaluate the effectiveness of the obtained results and to be able to compare with state-of-the-art implementations, we define the *Architecture efficiency* (*AE*) metric as follows:

$$AE = \frac{Throughput \times N_{iter}}{Area_{Norm} \times f_{clk}} \tag{8.7}$$

Its unit of measure is bits/cycle/iteration/mm^2 and it represents the number of decoded bits per clock cycle per iteration per mm^2 that the proposed iterative channel decoder implementation is able to deliver. A high architecture efficiency indicates an optimized design which exploits efficiently its hardware resources during its execution time. An interesting point in the above expression of the *AE* concerns the normalization of the throughput achieved with respect to the considered clock frequency (f_{clk}) which increases the fairness when comparisons are done between different decoding architectures running at different clock frequencies. Published results in this context consider either the maximum achievable clock frequency by the proposed architecture or a lower operational clock frequency which is sufficient to achieve the target throughput. Thus, normalizing the presented throughput by the considered clock frequency enables to better exhibit the efficiency of the proposed architectural choices. Towards the same objective, the above expression of the *AE* normalizes the throughput by the considered number of decoding iterations (N_{iter}) as the published results can use slightly different values which impact the overall throughput. In most of these works, the same low complexity decoding algorithms, with identical convergence speed, are used. Similarly, the *AE* expression uses a normalized area measure ($Area_{Norm}$) as the published decoders are often based on different technology nodes (e.g., 180 nm, 130 nm, 65 nm, etc.). In addition, when the published design area is given post-place and route a downscaling factor of 2 is applied to obtain a reasonable estimate of the post-synthesis area. This factor is not very accurate as it depends to many parameters (technology node, CAD tools, operating conditions, etc.), but it gives a reasonable idea as it corresponds to the usually (or even worst case) observed ratio.

Considering the synthesis results presented above, a 4x4 TurbASIP turbo decoder presents an architecture efficiency of 2.6 bits/cycle/iteration/mm^2. This value is somehow low compared to recent related state-of-the-art implementations as it will be illustrated in the next sub-section and this is due to the target design objectives of high degrees of flexibility and scalability. In fact, the example of this highly flexible ASIP allows to illustrate the effectiveness of the ASIP design approach allowing to explore and implement rapidly any desired algorithm/architecture choices and to generate high quality synthesizable HDL code. It allows also the design of scalable multi-ASIP turbo decoders meeting the high-throughput requirement. The next sub-section further illustrates how it is possible to increase significantly the architecture efficiency of ASIP-based turbo decoders.

8.3.4 TDecASIP: Parameterized Area-Efficient ASIP

In this sub-section we present another example of ASIP for multi-standard turbo decoding and we analyze and discuss the architecture efficiency with respect to related state-of-the-art implementations. The design objective behind this ASIP design example, namely TDecASIP [29], is twofold: (1) investigate the maximum attainable architecture efficiency for ASIP-based turbo decoding, and related to this first objective (2) investigate the possibility to design application-specific parametrized cores using the available ASIP design flow. Such possibility can potentially lead to a higher architecture efficiency by simplifying the instruction decoding logic and removing the program memory. Furthermore, it is still keeping the benefit of the short design cycle enabled by the well-established ASIP design tools.

8.3.4.1 Overview of TDecASIP Architecture

A first key element to increase the architecture efficiency is to limit the flexibility degree to the exact target application at design-time. Rather than supporting any trellis code as in TurbASIP, in this design example of TDecASIP the target flexibility is set to cover only the turbo codes and related parameters specified in 3GPP-LTE, WiMAX, and DVB-RCS standards. In addition, this choice enables to compare the results with existing state-of-the-art implementations.

The second key element concerns the algorithm/architectures choices in terms of parallelism techniques and degrees. Most appropriate choices have been made targeting a throughput in the range of hundreds Mbps, as specified in the considered standards. In order to fully exploit the metric level parallelism, two recursion units are devised using backward–forward schedule for window processing. The first recursion unit (processing in the backward direction of the trellis) processes a window j while the second recursion unit (processing in the forward direction of the trellis) executes on the window $j-1$ in parallel as illustrated in Fig. 8.3. This enables

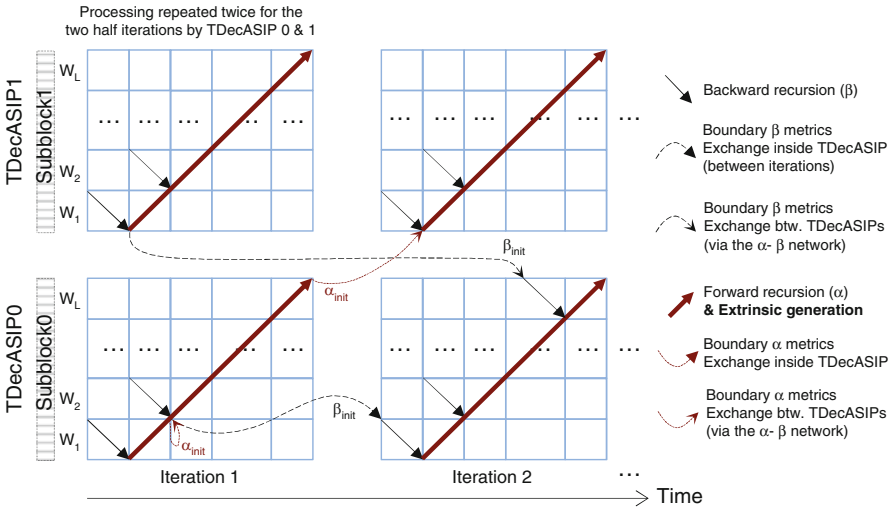

Fig. 8.3 Windowing and backward–forward schedule in TDecASIP

to achieve the throughput equivalent to butterfly schedule (as in TurbASIP design), yet the using of the backward–forward schedule further enables efficient use of hardware interleave address generators for extrinsic memory addressing.

Regarding the exploitation of the second level of parallelism (SISO decoder level), only sub-blocking parallelism with a degree of two is devised as it allows to meet the target throughputs. Shuffled decoding efficiency is demonstrated only for very high parallelism degrees [26]. Thus, in TDecASIP, half iterations are performed in serial order, i.e., all processing cores perform first half iteration by reading the systematic and extrinsic information sequentially from memories, followed by the second half iteration where the systematic and extrinsic memories are read in interleaved order. The generated extrinsic data are written at the same location as it was read from. In both of these half iteration cycles the parity memory is always read sequentially. This type of scheduling presents the following advantages:

- Only one copy of systematic information bits is needed to be stored. This reduces the number of memory banks required and the configuration network complexity.
- Only sequential counter and interleaved address generator are needed for addressing the memories while the shuffled decoding needs in addition a deinterleaved address sequence. Given the adopted low sub-block parallelism degree, this serial decoding reduces the memory access complexity as only low number of multiplexers would be sufficient (read/write exchange network). WiMAX interleavers support sub-blocking of 2 and 4 while LTE interleavers support sub-blocking of at least 2 and 4 [30] (with a maximum of 64).
- Small number of memory banks also results in less address decoding logic and hence reduced total memory area, resulting in area-efficient decoding core.

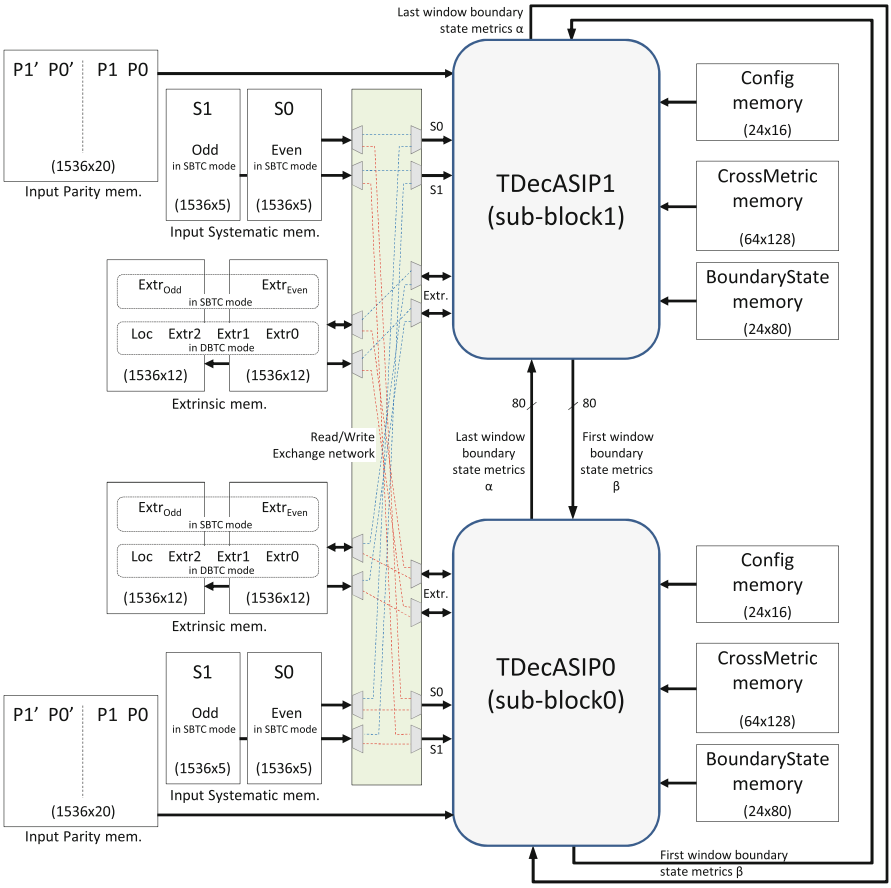

Fig. 8.4 Overview and memory organization of the 2-TDecASIP turbo decoder architecture [29]

Based on the above design motivations and choices, Fig. 8.4 illustrates the overall architecture of the two core turbo decoder. Each core (TDecASIP) processes a sub-block of the input frame and has direct access to configuration, CrossMetric, BoundaryState, and input Parity memories. The input Systematic and Extrinsic memory banks are connected to the cores through a simple read/write exchange network.

Figure 8.5 presents a detailed architecture of TDecASIP core. As for TurbASIP, the pipeline is structured in eight stages, of which the first three stages are dedicated for the data fetch from the memories and for the control of the pipeline. A modified ASIP design flow is proposed to enable parameterized core design. Rather than defining specialized instructions, the corresponding FSM is directly described in LISA. The current state of the FSM is treated as an instruction. This approach can be effective when the application exhibits a reduced number of flexible parameters

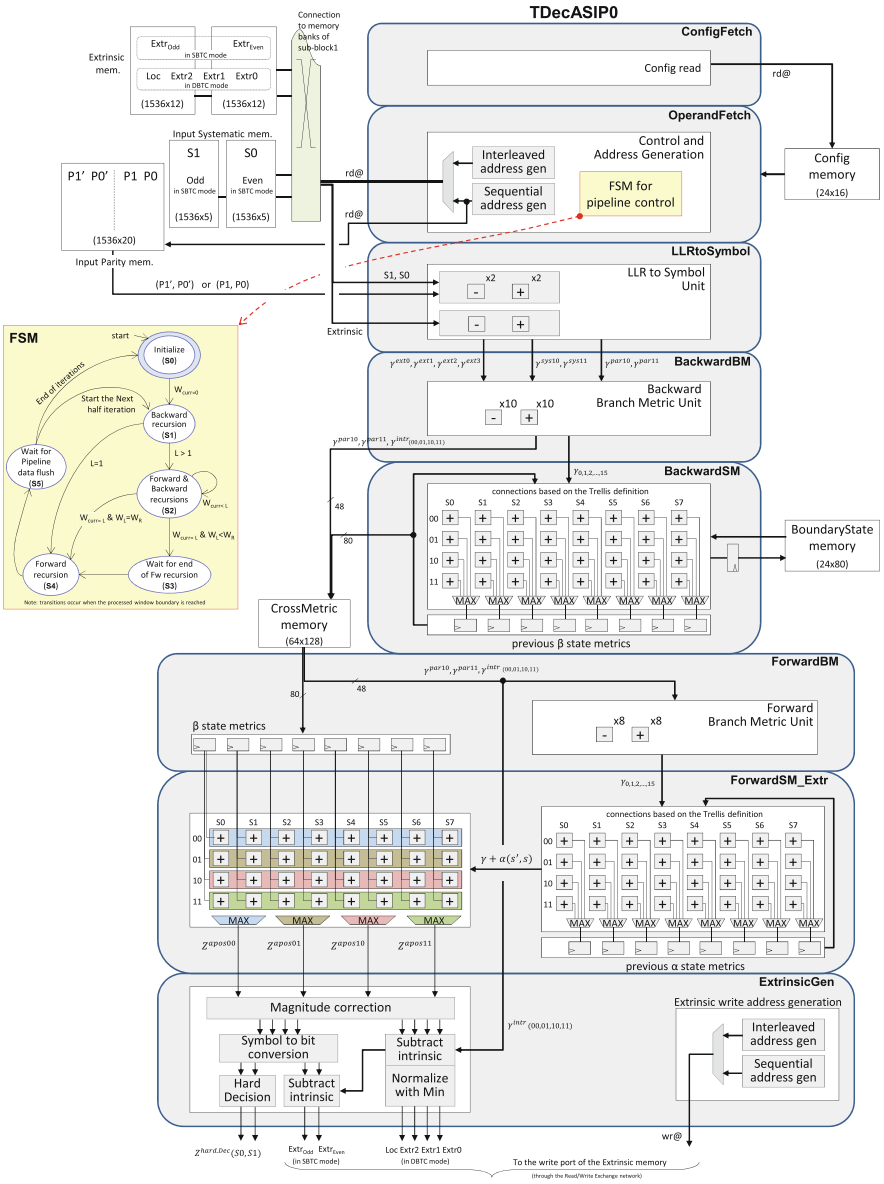

Fig. 8.5 Detailed pipeline architecture and FSM of the TDecASIP parametrized core [29]

and the corresponding processing presents a reduced number of control states. The target application in this study (flexible turbo decoding) is a good example with six states (as shown in Fig. 8.5) and few flexible parameters that do not change during the decoding process (code type, number of iterations, window size, number of windows, and extrinsic address generation initialization values). This FSM is implemented in the *OperandFetch* pipeline stage to generate appropriate control signals to activate or deactivate the appropriate stages of the pipeline (Fig. 8.5). As soon as the start signal is asserted, the processor starts with the *Initialize* state, initializing the registers to the default values and reading the configuration parameters mentioned above. At the end of the initialization, the FSM reaches *S1* state generating appropriate signals for the backward recursion execution. If the processor is executing the first half iteration, the generated addresses for systematic and extrinsic memories are sequential otherwise interleaved addresses are generated. The addresses for parity memories are always sequential. All FSM transitions in Fig. 8.5 occur when the window boundary is reached. In *S2* state both forward and backward recursions are executed in parallel (on two different windows). The complete presentation of the FSM control states and the pipeline execution can be found in [29].

The memory organization of the proposed architecture is illustrated in Fig. 8.4. With negligible performance loss, the channel LLRs can be quantized to 5 bits and the normalized extrinsic information to 7 bits. As radix-4 is adopted in SBTC, systematic LLRs are stored in two memory banks, and similarly for extrinsic LLRs. This memory organization and the corresponding efficient address generation are allowed by the QPP (quadratic permutation polynomial) interleaver adopted in LTE standard which maps even addresses to even addresses and odd to odd. The total depth of these memories allow to store up to 6,144 LLRs, which corresponds to the maximum specified LTE frame length. As the parity LLRs are always read in sequence, the consecutive parity LLRs information bits are combined and stored in one memory bank as shown in Fig. 8.4.

8.3.4.2 Results and Discussions

TDecASIP was modeled using Synopsys Processor Designer tool and the corresponding VHDL description was generated and synthesized targeting 65 nm general purpose CMOS technology (worst case 0.9v and 125C). The total logic area, including the interleaver, is 0.065 mm^2 while the memory area for one processor is 0.15 mm^2. The total area (post-synthesis) for the two core turbo decoder is 0.438 mm^2 with a clock frequency of 510 MHz. The error rate performance of the hardware implementation has negligible degradation (less than 0.1 dB) when compared to the floating point C-simulations using BPSK modulation over an additive white gaussian noise (AWGN) channel (Fig. 8.6). The throughput can be expressed as follows:

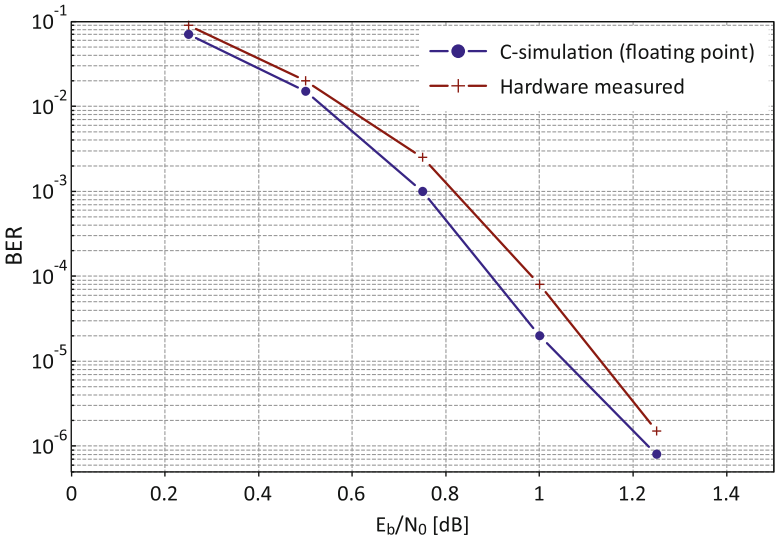

Fig. 8.6 Error rate performance comparison between the hardware implementation and the floating point simulation for WiMAX frame size 1,920 bits

$$Throughput_{TDecASIP} = \frac{N \times f_{clk}}{((\frac{\lceil N_{sym}/W \rceil}{Num_{procs}} + 1) \times W + N_{pip}) \times (2 \times N_{iter})} \tag{8.8}$$

For the presented architecture: $Num_{procs} = 2$ processors, number of pipeline stages $N_{pip} = 8$, window size $W = 64$ symbols, maximum clock frequency is $f_{clk} = 510$ MHz, considering the largest LTE frame size $N_{sym} = 3{,}072$ symbols or $N = 6{,}144$ bits and $N_{iter} = 6.5$ iterations, the throughput obtained is 150 Mbps.

Considering the architecture efficiency definition given in the previous subsection, the proposed 2 processor turbo decoder achieves an architecture efficiency of 4.37 bits/cycle/iteration/mm^2. Furthermore, the proposed architecture is scalable and can be extended to 4 processing cores, since both LTE and WiMAX interleavers support sub-blocking level of 4 with conflict-free memory accesses. In this case, the memory area of one processing core decoder becomes 0.097 mm^2 which results in a total area occupancy of 0.65 mm^2. The architecture efficiency in this case is 5.88 bits/cycle/iteration/mm^2. This further illustrates the area efficiency of the sub-block parallelism, where the throughput is doubled while the occupied area is increased only by 1.47 times (rather than doubled). This is due to the fact that Systematic, Parity, Extrinsic, and BoundaryState memory requirements remain unchanged. The achieved results of this design are summarized and compared along with few recent related works in Table 8.2. The cited three implementations [24, 31, 32] use a conventional parametrized design approach with almost similar internal computation, interleaving, and storage optimization techniques. However, each of them has selected a different sub-blocking parallelism level (8, 16, and 32). The increased

Table 8.2 Results and comparison with few recent related works

	TDecASIP		[24]	[31]	[32]
Standard supported	LTE, WiMAX		LTE, WiMAX	LTE	LTE
LTE modes supported #	188		188	188	188
WiMAX modes supported #	17		17	–	–
Technology (nm)	65		130	90	65
Core area (mm^2)	0.438	0.65	10.7[a]	2.1	7.7[a]
$Area_{Norm}$ @65 nm (mm^2)	0.438	0.65	1.335	1.1	3.85
Throughput (Mbps)	150 @6.5iter	300 @6.5iter	187 @8iter	284 @5iter	2150 @6iter
Parallel MAPs #	**2**	**4**	**8**	**16**	**32**
f_{clk} (MHz)	510		250	200	450
AE (bits/cycle/iter/mm^2)	4.37	5.88	4.48	6.49	7.45

[a] Post place and route

architecture efficiency with the sub-blocking parallelism degree is coherent with the above discussed results of the proposed 2- and 4-TDecASIP architectures. The 4-TDecASIP architecture achieves even a slightly better architecture efficiency than the one presented in [24] which supports both turbo modes (DBTC and SBTC) and uses 8 parallel MAP decoders. The LTE-dedicated implementations presented in [31] and [32] exploit the available higher sub-blocking parallelism degrees in this standard (parallel interleaving with conflict-free memory accesses). Results comparison illustrates how the TDecASIP architecture achieves a high architecture efficiency while using such an ASIP-based parameterized core approach by selecting the appropriate parallelism and optimization techniques.

8.4 Flexibility Increase to Support Multiple Channel Code Classes

Flexibility requirement of channel decoding architectures becomes more and more crucial when considering the emerging multi-mode and multi-standard applications, as well as the increasing interest for SDR and Cognitive Radio concepts. Even for a single standard like WiMAX, several error correction codes (convolutional, turbo, LDPC, and block turbo) are specified as mandatory or optional. Hence, in the last few years, several multi-code architectures have been explored and proposed to support the decoding of two or more different classes of error correction codes. The aim is to propose novel optimization and resource sharing techniques of the memory, logic, and/or communication interconnects in order to achieve better

efficiency in terms of area when compared to the direct assembly of dedicated individual decoders. This section presents a short review of the related state-of-the-art contributions to introduce the trend towards the use of ASIP-based design approach in this context. Due to the limited space, presenting few detailed examples and an exhaustive analysis is not affordable, yet a set of recent references is provided.

In this context, few initial initiatives have investigated the support of both convolutional and turbo codes. Authors of [33] and [34] have proposed a unified architecture designed for UMTS base-stations. A dual mode Viterbi/turbo decoder, sharing path metric calculation and extrinsic information memories, is proposed. A trellis processor used to update path metrics in both supported decoding algorithms. A 2 Mbps throughput at 88 MHz clock frequency is demonstrated when performing 10 turbo decoding iterations. In [35], another combined architecture is proposed. In this architecture the datapath and the memories are shared. A max-log-MAP algorithm is used for decoding both convolutional and turbo codes. However, this is only possible when the throughput requirement for convolutional codes (e.g., 12.2 kbps) is much lower than that of turbo codes (e.g., 384 kbps). In another effort to combine the two types of decoders, soft Viterbi decoding is used for turbo decoding and hard output Viterbi decoding is used for convolutional codes [36].

Similarly, unified decoder architectures for LDPC and turbo codes have been presented in [37–40]. Multi-code decoding is achieved in [37] by employing flexible add-compare-select (FACS) units. By representing LDPC codes as parallel concatenated Single Parity Check (SPC) codes, the authors have efficiently reused the turbo decoding hardware resources for LDPC decoding functions. The architecture supports decoding of SBTC codes of LTE and LDPC codes of WiFi and WiMAX. When implemented in 90 nm CMOS technology, the work reports a maximum throughput of 450 Mbps for SBTC decoding and 600 Mbps for LDPC decoding while occupying a total area of 3.2 mm^2. Similar architecture is presented in [40] to share logic and memory resources with additional decoding support of turbo codes specified in 3GPP, DVB-SH, and WiMAX standards. The entire design is implemented in 45 nm CMOS technology occupying an area of 0.9 mm^2 and clocked at 150 MHz to achieve low power and yet meeting the target throughput. However, studies presented in [41] conclude that such datapath sharing for LDPC and turbo decoding has little benefits and only for special configurations which have similar memory requirements between the decoding modes (LDPC/turbo). It further mentions that even in such cases, sharing memory is much more attractive than sharing computational hardware. In fact, the best match for a combined LDPC/turbo datapath can be achieved when both have the same granularity, e.g., at the check-node and log-butterfly operator level [41].

Besides the above-mentioned multi-code decoder architectures which can be considered as parameterized cores, several recent initiatives have explored ASIP-based design in this application context. As an example, the FlexiTreP ASIP presented in [42] supports trellis-based channel codes (i.e., convolutional, SBTC, and DBTC) for few target standards. Decoding of LDPC codes was later added to

this architecture and presented in [43] as FlexiChap where memory sharing across turbo and LDPC modes was explored. Targeting the support of WiFi and WiMAX, the total area of the channel decoder has increased from 0.42 mm^2 for FlexiTreP to 0.62 mm^2 for FlexiChap in 65 nm CMOS technology. Moreover, in [39], the authors propose an ASIP architecture addressing in a unified way the turbo and LDPC coding requirements of LTE, WiFi, WiMAX, and DVB-S2/T2 with datapath and memory reuse across the different FEC families. Results illustrate how the obtained area was lower than the cumulated area of dedicated individual turbo and LDPC decoder solutions. Furthermore, the authors of this chapter have presented in [28] several contributions belonging to these last efforts targeting ASIP-based multi-code channel decoding architectures. Promising results are presented while investigating the maximum achievable architecture efficiency when adopting the rapid design methodology and well-established tools related to ASIP design concept.

Finally, several scalable multiprocessor architectures based on the use of dedicated NoC for combined LDPC/turbo decoding were investigated. High decoder parallelism degrees are necessary to achieve the increasing throughput requirement imposed by the emerging applications. However, this incurs memory access conflicts due to the interleaving rules specified in the turbo and LDPC codes. Efficient algorithms which can compute collision-free memory mapping of interleaving laws with no constraint imposed on the code itself and the target parallelism degree are proposed in [44] and [45]. The latter further proposes novel technique that allows for low-latency dynamic reconfiguration. Another, or complementary, alternative proposes the use of adequate NoC topologies with optimized message transfer techniques. Several flexible on-chip interconnection networks have been proposed with the aim of fully exploiting the parallelism of the LDPC/turbo decoder architecture by reducing the message latency, alleviating the memory conflicts and efficiently routing any permutation law. Among the recent related contributions we can cite the work presented in [46] which proposes a NoC architecture based on binary de Bruijn topology and the work presented in [47] which proposes a NoC-based multiprocessor architecture, based on generalized Kautz topology.

8.5 Summary

This chapter has illustrated the use of ASIP models and tools as one of the main recent design approaches towards the target of unify flexibility and efficiency in the design of multi-standard channel decoders. Many contributions are emerging rapidly in this domain, seeking to increase the flexibility support and to improve the resulting architecture efficiency. The presented design examples of ASIP for turbo decoding illustrate how this approach enables to implement any set of architecture choices targeting different design objectives. The flexibility degree can be tuned to be very high or very limited; even parameterized architecture models can be developed using ASIP design tools. The main benefits correspond to the

structured and rapid design approach which accelerates the design and validation flow and enables high design-time flexibility with respect to traditional digital design practices.

The chapter illustrated how flexibility, architecture efficiency, and rapid design-time can be combined when using an ASIP design methodology and tools to implement novel cores for multi-standard turbo decoding. The highly optimized parameterized core presented supports both SBTC of 3GPP-LTE and DBTC of WiMAX and DVB-RCS standards. It achieves, in both modes, a high architecture efficiency of 4.37 bits/cycle/iteration/mm^2 and meets the 150 Mbps maximum targeted throughput of the LTE standard. The proposed architecture is scalable and the architecture efficiency increases with the sub-block parallelism degree. Detailed analysis and comparisons with relevant state-of-the-art solutions have been discussed.

Same approach of exploring the maximum achievable architecture efficiency using ASIP design concept in LDPC decoding can be found in [48]. This design trend has been further shown in the proposal of flexible channel decoder supporting multiple code types; in particular the support of LDPC and turbo codes as specified in recent wireless communication standards. The chapter has presented in this context a short review of the related state-of-the-art contributions. Finally, although the chapter has mainly considered the evaluation of the architecture efficiency, similar conclusions should be driven evaluating the energy consumption and efficiency. Furthermore, several recent initiatives have proposed to tackle the aspect of reconfiguration optimization and efficient management for multi-standard ASIP-based channel decoders [49]. Finally, the promising results demonstrated in recent state-of-the-art in channel decoding have paved the way to extend this design approach to other key components of advanced communication systems. Several recent contributions start appearing in this context, e.g., for MIMO detection, and even beyond digital communication applications domain.

References

1. Ienne P, Leupers R (2006) Customizable embedded processors–design technologies and applications. Morgan Kaufmann, San Mateo
2. Cadence Xtensa Customizable Processors (formerly product of Tensilica) (2014). http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable
3. Stretch Software-Configurable Processors (2014). http://www.stretchinc.com/technology/
4. Mei B, Lambrechts A, Mignolet J-Y, Verkest D, Lauwereins R (2005) Architecture exploration for a reconfigurable architecture template. IEEE Trans Des Test Comput 22(2):90–101
5. Synopsys Processor Designer (formerly product of CoWare) (2014). http://www.synopsys.com/Systems/BlockDesign/ProcessorDev
6. Synopsys IP Designer (formerly product of Target Compiler Technologies) (2014). http://www.synopsys.com/IP/ProcessorIP/asip/ip-mp-designer
7. Robertson P, Hoeher P, Villebrun E (1997) Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding. Eur Trans Telecommun 8(2):119–125

8. Bickerstaff M, Davis L, Thomas C, Garrett D, Nicol C (2003) A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless. In: Proceedings of the IEEE international solid-state circuits conference (ISSCC), vol 1, pp 150–484
9. Berrou C (1991) Procédé de décodage itératif, module de décodage et décodeur correspondants/Iterative decoding method, corresponding decoding module and decoder, Patent EP0735696 B1, France Telecom & TDF.
10. Hsu J-M, Wang C-L (1998) A parallel decoding scheme for turbo codes. In: Proceedings of the IEEE international symposium on circuits and systems (ISCAS)
11. Wang Z, Suzuki H, Parhi K (1999) VLSI implementation issues of TURBO decoder design for wireless applications. In: Proceedings of the IEEE workshop on signal processing systems (SiPS'99), pp 503–512
12. Kwon TW, Kim DW, Kim WT, Joo EK, Choi JR, Choi P, Kong JJ, Choi SH, Chung WH, Lee KW (1999) A modified two-step sova-based turbo decoder for low power and high performance. In: Proceedings of the IEEE region 10 conference (TENCON'99), vol 1, pp 297–300
13. Chaikalis C, Noras J (2002) Implementation of an improved reconfigurable sova/log-map turbo decoder in 3gpp. In: Proceedings of the third international conference on 3G mobile communication technologies, May, pp 146–150
14. Boutillon E, Douillard C, Montorsi G (2007) Iterative decoding of concatenated convolutional codes: implementation issues. Proc IEEE 95(6):1201–1227
15. Gnaedig D (2005) Optimisation des architectures de décodage des turbo-codes. Ph.D. dissertation, Telecom Bretagne – UBS
16. Viglione F, Masera G, Piccinini G, Ruo Roch R, Zamboni M (2000) A 50 mbit/s iterative turbo-decoder. In: Proceedings of the ACM/IEEE design, automation and test in Europe conference and exhibition (DATE'00), pp 176–180
17. TMS320C64x DSP turbo-decoder coprocessor (2004). http://www.ti.com/lit/ug/spru534b/spru534b.pdf
18. Loo K, Alukaidey T, Jimaa S (2003) High performance parallelised 3GPP turbo decoder. In: Proceedings of the 5th European personal mobile communications conference
19. Zhong Z, Peng T, Zhong Z, Wang W, Liu Z (2010) Hardware implementation of Turbo coder in LTE system based on PICOCHIP PC203. In: Proceedings of the 12th IEEE international conference on communication technology (ICCT)
20. Gilbert F, Thul M, Wehn N (2003) Communication centric architectures for turbo-decoding on embedded multiprocessors. In: Proceedings of the design, automation and test in Europe conference & exhibition (DATE)
21. Wong C-C, Lee Y-Y, Chang H-C (2009) A 188-size 2.1mm2 reconfigurable turbo decoder chip with parallel architecture for 3GPP LTE system. In: Proceedings of the symposium on VLSI circuits, pp 288–289
22. Ilnseher T, Kienle F, Weis C, Wehn N (2012) A 2.15GBit/s turbo code decoder for LTE advanced base station applications. In: Proceedings of the international symposium on turbo codes and iterative information processing (ISTC)
23. Lin C-H, Chen C-Y, Chang E-J, Wu A-Y (2011) A 0.16nJ/bit/iteration 3.38mm2 turbo decoder chip for WiMAX/LTE standards. In: Proceedings of the international symposium on integrated circuits (ISIC)
24. Kim J-H, Park I-C (2009) A unified parallel radix-4 turbo decoder for mobile WiMAX and 3GPP-LTE. In: Proceedings of the IEEE custom integrated circuits conference (CICC)
25. Muller O, Baghdadi A, Jezequel M (2006) ASIP-based multiprocessor SoC design for simple and double binary turbo decoding. In: Proceedings of the design, automation test in Europe conference & exhibition (DATE)
26. Muller O, Baghdadi A, Jezequel M (2010) Parallelism efficiency in convolutional turbo decoding. EURASIP J Adv Signal Process 2010:927920. doi:10.1155/2010/927920
27. Moussa H, Muller O, Baghdadi A, Jezequel M (2007) Butterfly and Benes-based on-chip communication networks for multiprocessor turbo decoding. In: Proceedings of the design, automation test in Europe conference & exhibition (DATE)

28. Murugappa P (2012) Towards optimized flexible multi-ASIP architectures for LDPC/turbo decoding. Ph.D. dissertation, Telecom Bretagne – UBS
29. Murugappa P, Baghdadi A, Jezequel M (2013) Parameterized area-efficient multi-standard turbo decoder. In: Proceedings of the design, automation test in Europe conference & exhibition (DATE)
30. Sun Y, Zhu Y, Goel M, Cavallaro J (2008) Configurable and scalable high throughput turbo decoder architecture for multiple 4G wireless standards. In: Proceedings of the international conference on application-specific systems, architectures and processors (ASAP)
31. Ahmed A, Awais M, Rehman A, Maurizio M, Masera G (2011) A high throughput Turbo decoder VLSI architecture for 3GPP LTE standard. In: Proceedings of the IEEE 14th international multitopic conference (INMIC), pp 340–346
32. Ilnseher T, Kienle F, Weis C, Wehn N (2012) A 2.15GBit/s turbo code decoder for LTE advanced base station applications. In: Proceedings of the 7th international symposium on turbo codes (ISTC)
33. Bickerstaff M, Garrett D, Prokop T, Thomas C, Widdup B, Zhou G, Davis L, Woodward G, Nicol C, Yan R-H (2002) A unified turbo/viterbi channel decoder for 3gpp mobile wireless in 0.18-mm cmos. IEEE J Solid-State Circuits 37(11):1555–1564
34. Thomas C, Bickerstaff MA, Davis LM, Prokop T, Widdup B, Zhou G, Garrett D, Nichol C (2003, April) Integrated circuits for channel coding in 3g cellular mobile wireless systems. IEEE Commun Mag 150–159
35. Kreiselmaier G, Vogt T, Wehn N (2004) Combined turbo and convolutional decoder architecture for UMTS wireless applications. In: Proceedings of the design, automation test in Europe conference & exhibition (DATE)
36. Cavallaro JR, Vaya M (2003) VITURBO: a reconfigurable architecture for Viterbi and turbo decoding. In: Proceedings of the international conference on acoustics, speech, and signal processing (ICASSP)
37. Sun Y, Cavallaro JR (2008) Unified decoder architecture for LDPC/Turbo codes. In: Proceedings of the IEEE workshop on signal processing systems (SiPS)
38. Sun Y, Cavallaro J (2011) A flexible LDPC/turbo decoder architecture. J Signal Process Syst 64(1):1–16
39. Naessens F, Bougard B, Bressinck S, Hollevoet L, Raghavan P, Van der Perre L, Catthoor F (2008) A unified instruction set programmable architecture for multi-standard advanced forward error correction. In: Proceedings of the IEEE workshop on signal processing systems (SiPS)
40. Giuseppe Gentile MR, Fanucci L (2010) A multi-standard flexible turbo/LDPC decoder via ASIC design. In: Proceedings of the 6th international symposium on turbo codes and iterative information processing
41. Dielissen J, Engin N, Sawitzki S, van Berkel K (2008) Multistandard fec decoders for wireless devices. IEEE Trans Circuits Syst II Express Briefs 55(3):284–288
42. Vogt T, Wehn N (2008) A reconfigurable application specific instruction set processor for convolutional and turbo decoding in a sdr environment. In: Proceedings of the design, automation test in Europe conference & exhibition (DATE)
43. Alles M, Vogt T, Wehn N (2008) FlexiChaP: a reconfigurable ASIP for convolutional, turbo, and LDPC code decoding. In: Proceedings of the 5th international symposium on turbo codes and related topics
44. Tarable A, Benedetto S, Montorsi G (2004) Mapping interleaver laws to parallel turbo and ldpc decoders architectures. IEEE Trans Inf Theory 50(9):2002–2009
45. Sani A, Coussy P, Chavet C (2013) A first step toward on-chip memory mapping for parallel turbo and LDPC decoders: a polynomial time mapping algorithm. IEEE Trans Signal Process 61(16):4127–4140
46. Moussa H, Baghdadi A, Jezequel M (2008) Binary de Bruijn on-chip network for a flexible multiprocessor LDPC decoder. In: Proceedings of the 45th design automation conference (DAC)

47. Condo C, Martina M, Masera G (2012) A network-on-chip-based turbo/LDPC decoder architecture. In: Proceedings of the design, automation test in Europe conference & exhibition (DATE)
48. Murugappa P, Lapotre V, Baghdadi A, Jezequel M (2013) Rapid design and prototyping of a reconfigurable decoder architecture for QC-LDPC codes. In: Proceedings of the IEEE international symposium on rapid system prototyping (RSP)
49. Lapotre V, Murugappa P, Gogniat G, Baghdadi A, Diguet J-P, Bazin J-N, Hubner (2013) Optimizations for an efficient reconfiguration of an ASIP-based turbo decoder. In: Proceedings of the IEEE international symposium on circuits and systems (ISCAS)

Chapter 9
Hardware Design of Parallel Interleaver Architectures: A Survey

Cyrille Chavet, Awais Hussain Sani, and Philippe Coussy

9.1 Motivation

Early developers of digital communication systems assumed that information could be transmitted through noisy channel with high reliability by increasing the signal to noise ratio. This could only be achieved at that time by increasing transmitted signal power enough to ensure that signal can reliably be transmitted. The revolutionary work of Shannon [1] changed this view by proving that it is possible to send digital data to receiver through noisy channel with high reliability by first encoding digital message with error correction code at transmitter and then subsequently decode it at receiver to generate original message.

The function of the encoder is to map X digits message into C digits codeword where $C > X$. The code rate $r = X/C$ defines the redundancy introduced by corresponding error correction code. Encoded message passes through channel which corrupts the message by adding some noise into it. At receiver, error correction decoder uses this added redundancy to determine the original message despite the noise introduced by channel. Typical communication chain is shown in Fig. 9.1.

Different error correction codes are introduced in literature. They can be classified into two broad categories: *block codes* and *convolutional codes*. In block codes, original information sequence is divided into different message blocks and each message is independently encoded to generate codeword bits whereas in convolutional codes, encoder takes information sequence as a continuous stream

C. Chavet (✉) • P. Coussy
Lab-STICC laboratory, Université de Bretagne Sud, 4 Rue Jean Zay, 56100 Lorient, France
e-mail: cyrille.chavet@univ-ubs.fr; philippe.coussy@univ-ubs.fr

A.H. Sani
SATT Ouest Valorisation, 14C rue du Pâtis Tatelin Métropolis 2, CS 80804,
35708 Rennes Cedex 7, France
e-mail: awais-hussain.sani@ouest-valorisation.fr

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_9

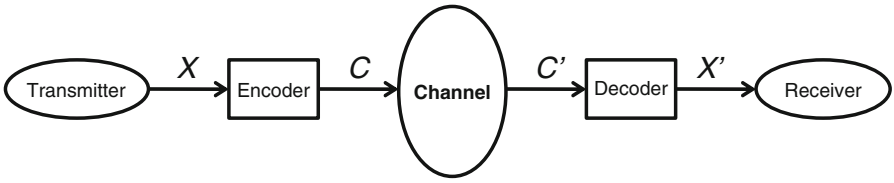

Fig. 9.1 Communication system

and generates a continuous stream of codeword bits. Therefore in block codes, encoder must wait for the entire message block before it starts encoding whereas convolutional encoder can start encoding and transmitting codeword before it obtains the entire message.

Many types of block codes are used in different applications but among the classical ones, Reed–Solomon [2] is the most popular due its widespread use in CD, DVD, and hard disk drives. Other examples of classical block codes are Golay codes [3] and Hamming codes [4]. Low density parity check codes (LDPC) is a class of linear block codes with error correction capabilities very close to the channel capacity. Due to their excellent error correction performance, it has already been included in several wireless communication standards such as DVB-S2 and DVB-T2 [5], WiFi-*IEEE 802.11n* [6], or WiMAX-*IEEE 802.16e* [7].

Convolutional codes, such as Turbo codes [8], perform like a finite state machine which converts continuous stream of X message bits into continuous stream of C coded bits (where $X > C$). Due to their simple structure and efficiently implementable iterative decoding algorithm, convolutional codes are increasingly used in different telecommunication standards. Thanks to their excellent error correction capabilities, Turbo codes are part of current telecommunication standards such as [9, 10] and digital broadcasting [11]. These Turbo codes are constructed through the parallel concatenation of two convolutional codes to achieve good error correction performance. Their outstanding performances are also possible due to the presence of pseudo-random interleaver that scrambles data to break up neighborhood relations.

Meanwhile, the large acceptance of smart-phones, laptop, digital television, and mobile broadband devices leads to the era of high data rate wireless applications. The rapid and huge increase in data traffic strains network capacity and researchers are developing new techniques to cope with this high throughput requirement. As a result of this effort, advanced technologies such as OFDM, MIMO, and advanced error correction techniques are included in different standards to reliably transfer data at high rates on wireless networks.

However, the excellent performance of error correction codes comes at the expense of computational complexity. Hence, parallel architectures must be employed to speed up the decoding process and support required application throughputs. Moreover, several parameters such as scheduling, parallelism level, memory organization, and network architecture need to be explored to trade

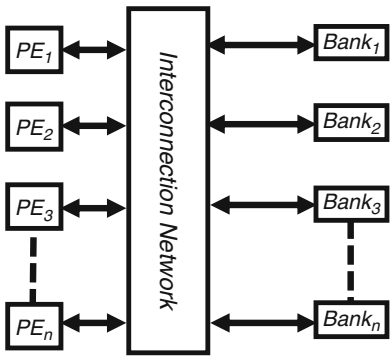

Fig. 9.2 Decoder parallel architecture

off circuit area and performances. This requires the development of dedicated approaches to efficiently implement decoder architecture. In such implementation (cf. Fig. 9.2), several processing elements (PEs) are used to obtain the required throughput. Memory is used to store different messages generated during the decoding process. These messages are written into and read out of the memory according to particular permutation defined by a permutation law. This architecture, however, suffers from memory access collision problem when more than two processing elements want to access the same memory bank. Collision problem becomes a significant issue with the increase of code word length and is discussed in the next section. Moreover, with the growing demand of high data rate applications and shrinking time to market constraint, this problem proves to be one of the most problematic factors in designing efficient decoder architectures.

To manage this problem, conflicts can be resolved either during definition of interleaving law or at run time or at design time. Designing conflict free interleaving law often simplifies the construction of parallel decoder architectures. However, it traditionally only supports particular parallelism used in decoding algorithms (e.g., LTE only supports SISO decoder level parallelism for a subset of block lengths). Managing conflict problems at runtime (e.g. serializing/postponing conflicting accesses through buffers) results in additional hardware cost and delay, and may be less interesting for high data rate and low power applications. In order to resolveconflict problem for any type of parallelism and interleaving law, design time conflict resolution is another solution. Here, conflict-free memory mappings are found off-chip either manually or automatically. In manual approach, the designer finds the conflict-free memory mapping after analyzing the interleaving law and then designs a controller using FSM controllers. However, in automatic approach different dedicated algorithms are developed and run on computer to generate ROM-based controllers. In order to support multiple block lengths and standards on a single chip, automated approaches require to pre-calculate memory mapping for each block length and to store them on-chip which results in large memory footprint. More recently, a new kind of approach has been proposed based on a hybrid strategy that aims to benefit both from runtime and design time approaches through on-chip memory mapping. The idea is to generate on-chip conflict free memory mappings, during the execution of the application.

9.2 Problem Formulation

First, we present a problem formulation based on an example based on access order of turbo decoder. However, it has been already been shown in [12–15]... that the problem is the same for LDPC codes.

In parallel decoder architectures, several processing elements *PEs* are concurrently used to decode the received information. In order to increase memory bandwidth, several memory banks *Bs* are connected with these *PEs* through a dedicated interconnection network (see Fig. 9.2). This network exchanges data between *PEs* and *Bs* according to predefined access orders. These orders are parameterized by block lengths and *PEs* parallelism.

Typical parallel decoder architecture is shown in Fig. 9.2. In our example, natural and interleaved orders are defined as follow:

Natural order = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
Interleaved order = {0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 11}

For parallel processing, this codeword is divided into four windows in both natural and interleaved order and arranged in data access matrices of Fig. 9.3. In this figure, each row (or window) is processed by one processing element whereas data in each column (or time instance) need to be accessed concurrently.

To increase memory bandwidth, three memory banks are used so that processing elements can concurrently get data elements in parallel. Data elements must be stored in banks in such a manner that at each time instance in natural order, all the processing elements always access different memory banks as shown in Fig. 9.4a. However by using this memory mapping, all processing elements always access the same memory bank at each time instant in interleaved order as shown in Fig. 9.4b. This results in memory conflict problem [16] and increases latency and thus reduces system throughput and increases system cost.

To solve this memory conflict problem, several approaches can be proposed in order to *manage concurrent parallel access to all the data elements in both read and write accesses with or without any conflict and/or with or without dedicated additional hardware mechanism*.

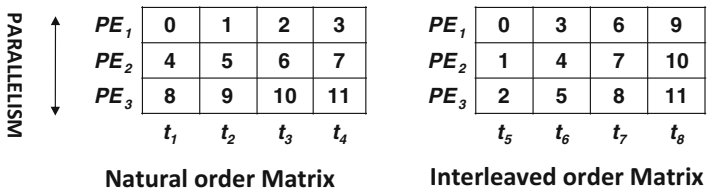

Fig. 9.3 Data access matrices

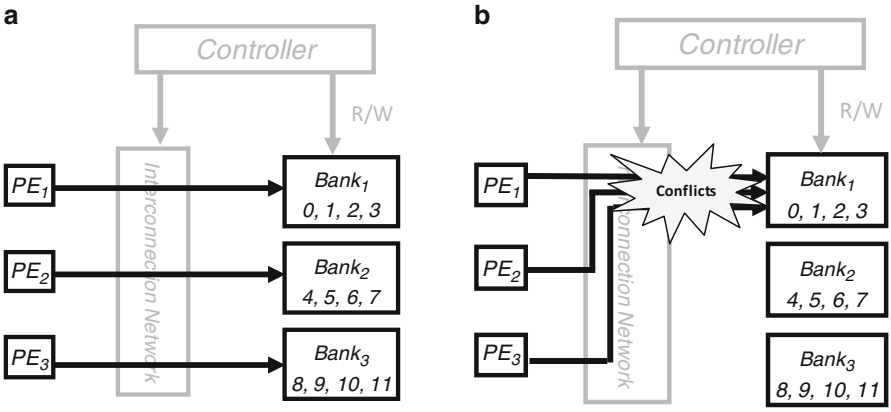

Fig. 9.4 Memory conflict problem in parallel turbo decoder. (**a**) Conflict-free natural order access. (**b**) Conflict-full interleaved order access

9.3 An Overview of Memory Access Conflict Solving Approaches

In recent standards, different conflict free interleaving laws have been defined. For example, 3GPP-LTE uses quadratic permutation polynomial (QPP) interleaver [17] whereas WiMAX [7] uses ARP [18] interleaver to permute the data. These interleavers often simplify the parallel decoder architecture. However, they are conflict-free only for particular types (e.g., [19] or [20]) or degrees of parallelism used in turbo decoding or for a subset of block lengths. Hence, several solutions are proposed in literature to solve such conflicts at runtime or at design time. At runtime, architectures use routing and/or buffering techniques in the interconnection network to serialize conflicting accesses. Design time approaches are able to generate in-place memory mapping (i.e., a given data is stored in one and only one memory bank, and one memory address) in order to reduce the cost of the controller. Some other approaches try to strongly optimize the controllers by moving a data from one memory place to another between each access to this data.

9.3.1 Conflict Solving During the Definition of the Interleaving Law

A first class of approach consists in defining conflict free interleaving law. An example of such solution is proposed in [21] based on Turbo-codes. In this approach spatial and temporal permutations of data are introduced to construct conflict-free interleaver with random interleaver like properties. Consider a block length of n data arranged row by row into a matrix M. Interleaver function is the sum of both

temporal and spatial permutations of the lines and columns of M. The benefit of this approach is that one can use barrel shifter interconnection network to realize turbo decoder for this interleaving law in practical applications. However, the approach is dedicated, i.e., not standard compliant.

More recently, [17] has proposed the QPP interleaver architecture. The authors show that QPP interleaver is maximum contention-free, i.e., for every window size W, which is a factor of the interleaver length N, the interleaver is contention free. QPP interleaver is defined by the following equation:

$$\Pi(\mathrm{x}) = \left(f_1 x^2 + f_2 x\right) \mod N$$

where x and $\Pi(\mathrm{x})$ represent the original and interleaved address respectively, and integers f_1, f_2 are different for different block lengths and can be found in the standard.

This kind of approach has been recently used in standard like LTE. However, QPP contention-free property is true for SISO decoder level parallelism only. For higher data rate applications when trellis and recursive units parallelism are also included in each SISO, QPP interleaver is no more contention-free and requires additional router and buffer mechanisms to manage problems. Moreover, having no conflict for QPP interleaver allows to design conflict-free architecture in 3GPP LTE decoder only: it indeed results in designing a channel interleaver that has to manage memory conflicts in order to present data to QPP in the required organization.

From LDPC point of view, these codes are completely specified by their parity matrices. These matrices represent how data (named *variable nodes* in LDPC) must be processed by the processing elements (named *check nodes* in LDPC) in order to achieve good error correction performances. Hence, proper construction of such matrices is necessary to obtain excellent error correction capabilities of LDPC. Different constraints can be added during the construction of parity matrices either to achieve significant coding gains or to simplify the decoder architecture. The matrices can also be constructed is such a way that data transfer between check nodes (CNs) and variable nodes (VNs) is made without any conflict for partially parallel architecture [22, 23]. The codes obtained during such construction procedure are called *structured codes*.

Structured codes remove memory conflict problem because transfer of messages between (CNs) and (VNs) is carried out through simple rules (like indices permutation). Also, structured codes simplified the decoder architecture since interconnection network can be implemented through simple network (like barrel shifter) by exploiting the regularity introduced during code construction. Due to simplicity in construction, structured codes are part of current telecommunication standards; e.g. [6] or [7].

Although it is proved in [22] that performances of structured codes are very close to random codes, adding constraints to construct structured codes may degrade the code's decoding performance. Therefore, special attention should be taken while selecting constraints to develop structured codes to keep remarkable error correction capabilities of LDPC. Also, structured codes only support one class of LDPC codes

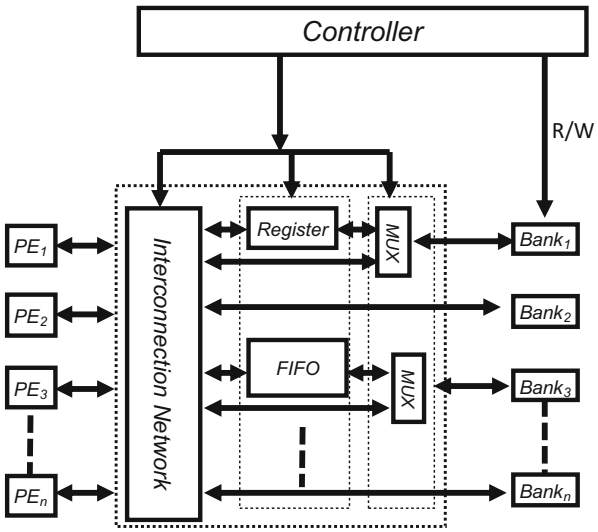

Fig. 9.5 Conflict buffering mechanism added to the network architecture

and to handle diverse existing and future classes of LDPC codes (such as no-binary LDPC codes), a general approach to handle memory mapping problem is required. Finally, it must be noticed that if the data access order can be defined to be conflict-free in the decoder part of the architecture; this supposes that this decoder can be fed with data in a proper order, which can differ from the order of data coming from the channel. Hence, in this case (like for QPP interleavers) the problem is only moved from inside the decoder up to its interface, i.e., its environment.

9.3.2 Conflict Solving Through Dedicated Runtime Approaches

A second class of solution to deal with memory access conflict problem is to simply store data elements in different memory banks without considering conflicting accesses and then use additional buffers in the interconnection network to manage conflicts at runtime [24] see Fig. 9.5.

In [25] the authors propose to add in the interconnection network a dedicated Double-Buffer based Contention-free (DBCF) to design a HSPA+ decoder. This architecture is configured, thanks to the statistical property of the memory conflict based on simulation results analysis. As soon as DBCFs detect conflicts, the conflicting accesses are routed into a dedicated circular buffer (see Fig. 9.6).

Another example of conflict-solving oriented architecture can be found in [26]. In this approach, the interconnection network is based on a Network-on-Chip that can be configured on-the-fly to emulate any "classical" interconnection network such

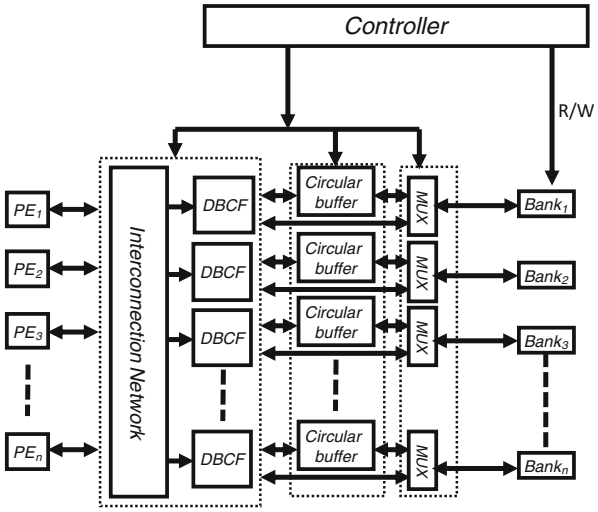

Fig. 9.6 Architecture based on DBCF

like Butterfly, Benes, De Bruijn, cross-bar… Then, if a memory conflict access is detected by the routers, then only one access will be performed to the memory banks, and the other conflicting ones will be re-routed into the network. Then, these conflicting "packets" could re-try to access the memory banks later on (see Fig. 9.7).

Runtime approaches generally increase the cost and latency of the system due to presence of interconnection network and buffer management mechanism to manage conflicts. The total latency of the system is also increased since each conflicting data access must travel buffers before being stored in the memory banks. Hence, such approaches are also often referred as *time relaxation* since additional cycles can be used to solve memory access conflicts.

9.3.3 Conflict Solving Through Dedicated Memory Mapping Approaches

A third class of solution to deal with memory access conflict problem is to store data elements (mapping process) in different memory banks so that all the Processing Elements (PEs) can access the data without any conflict. Dedicated mapping algorithms are used to perform some pre-processing steps to determine the memory locations for each data element used.

These algorithms can be categorized as (1) *unconstrained*, i.e., the targeted interconnection network supports any permutations (e.g., Benes networks, cross-bar…), (2) *hard-constrained*, i.e., the designer wants to use a cheaper (from architecture point of view, e.g., a barrel-shifter) interconnection network, but at

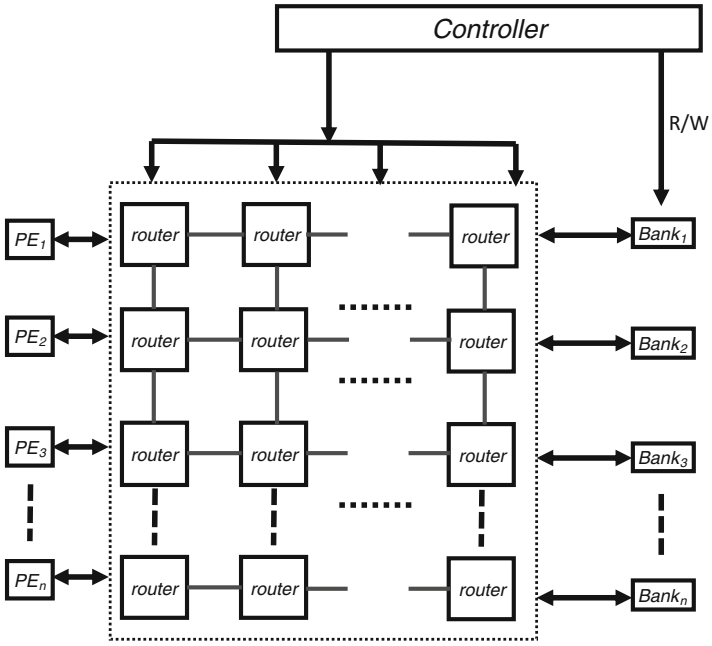

Fig. 9.7 NoC-based architecture

the expense of a strong limitation of the set of possible permutations, and (3) *soft-constrained*, i.e., the architecture can be modified during the memory mapping step in order to reduce its final complexity.

9.3.3.1 Unconstrained Memory Mapping Approaches

Several unconstrained mapping approaches are proposed in state of the art to find a memory mapping that will be natively conflict free; i.e., each processing elements can access all its data without any conflict at each time instance [13–15, 27]...

Simulated annealing approaches like [13] (in-place memory mapping only) or [14, 15] (in-place memory mapping and multi-read/multi-write memory mapping) are able to find a conflict-free memory mapping for both Turbo-codes and LDPC. In order to apply the proposed approach on LDPC, the authors of [13] proposed to modify memory access schedule by using a static single assignment (SSA) form which results in oversized memory architecture to store data since each data access is stored in a dedicated memory address. However, in [14, 15] the authors removed this limitation by using several memory locations for each data and by sharing this memory location among multiple data.

In [27] authors take leverages of Bipartite Edge Coloring techniques to solve the mapping problem for both turbo and LDPC codes in polynomial time. Hence,

they first transform data access matrices for Turbo Codes and mapping matrices for LDPC into bipartite graphs. Afterwards, bipartite edge coloring algorithm is applied on these graphs to solve mapping problem. Since edge coloring algorithm is always able to color the edges of bipartite graph with minimum colors, therefore it is always possible to solve memory mapping problems for both turbo and LDPC codes in polynomial time.

However, these approaches store data "randomly" in memory banks. Parallel interleaver architectures could thus be significantly optimized in terms of interconnection network and memory controller costs. Indeed no regularity can be easily extracted from the control words generated by the memory controller, and as a consequence, no optimization can be efficiently performed. Even if the designer would prefer to use an optimized network (e.g., Butterfly, Barrel-shifter...), if no dedicated memory mapping approach is proposed then the resulting interconnection network must be a full crossbar or a Benes network. Hopefully, some approaches are able to find conflict free memory mapping that is fully compatible with a user-defined interconnection network.

9.3.3.2 Hard-Constrained Memory Mapping Approaches

In [14, 15], an approach called Static Address Generation Easing—SAGE—is presented that considers a target interconnection network to find memory mapping. In this approach, two empty matrices called SAGE Mapping Matrices are used to store banks information during algorithm execution. To find architecture-oriented memory mapping, two constraints are defined to be respected during algorithm execution. First, each column of the mapping matrices (see Sect. 9.2) should contain different memory banks and second, if the interleaving law allows, each column should respect the rules of the steering network component. This approach guaranties if the target interconnection network is compatible with the interleaving law, then the final memory mapping will respect it.

In [28], an approach based on transportation problem modeling which finds conflict-free memory mapping for every type of turbo codes and which optimizes the resulting interleaving architecture has been proposed. The mapping problem for turbo codes is transformed as transportation problem by considering all the data nodes as producers and all the time nodes as consumers. The main interest of this approach is that the designer is able to obtain conflict-free memory mapping in polynomial time and this mapping respects the targeted interconnection network, if it is compatible with the interleaving law.

The main weakness of this kind of approaches is that they are limited to one targeted interconnection network defined at design time. Moreover, they cannot be applied to any data permutation approaches since they are limited to Turbo-codes based systems.

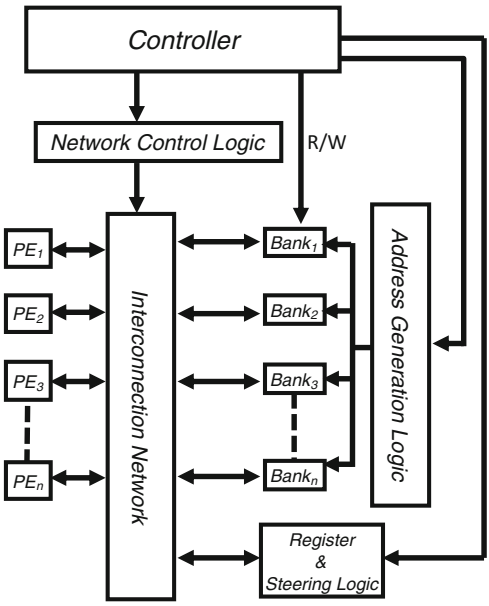

Fig. 9.8 Architecture generated by memory relaxation approach

9.3.3.3 Soft-Constrained Memory Mapping Approaches

The last class of approaches tries to take advantage of both runtime approach mechanisms and design time approach efficiency. In previous mapping approaches, the generated memory mappings induce sets of control words in order to control the memories and the network of the decoder architecture. If no regularity can be extracted from these control words no optimization can be performed. On the contrary, if such regularity can be extracted, the addressing sequence, and the associated controller cost can be greatly reduced.

Such regularity can be obtained by applying the approach described in [12]: in this solution, additional registers are used to store conflicting data but also some non-conflicting data if and only if this enables to simplify the memory controller architecture (see Fig. 9.8). This approach, referred as *Memory relaxation* (i.e., additional memory elements—registers or FIFOs—can be allocated to remove conflicts or to enable strong optimization of the controllers) is also able to generate a conflict-free memory mapping with respect to a target interconnection network (Barrel-Shifter, Butterfly...). However the final architecture suffers from hardware overhead due to additional registers and their dedicated additional steering logic. This overhead depends on the compatibility between interleaving law and permutation characteristics on the targeted interconnection network. For higher incompatibilities, the approach results in higher hardware overhead and latency.

Since interconnection network also impacts the cost of the architecture [28], then a smart memory mapping approach could also focuses on optimizing this network in order to adapt its structure to the interleaving law.

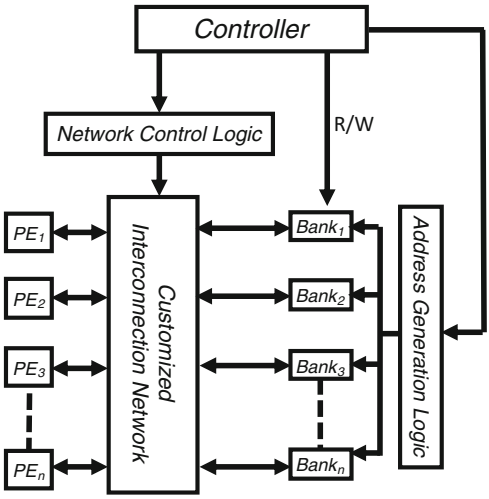

Fig. 9.9 Architecture generated by network relaxation approach

Ur Rehman et al. [29] presents a mapping approach that considers the customization of the interconnection network and reduces the cost of the controller architecture. This approach, referred as *Network relaxation*, modifies the original network by adding additional multiplexers/switches (see Fig. 9.9). The mapping step aims to fully explore the memory mapping solution space by checking all the permutations of the selected network. If no memory mapping solution exists for this network, then the set of permutations will be extended by adding a steering component which results into a customized network architecture with enriched set of permutations.

This approach proved to be the most efficient (compared to state of the art) in terms of hardware cost and latency. However, this approach, like the other ones from the literature, generates a static architecture that cannot be modified on-the-fly depending on system requirements (QoS, application switching, changing application parameters...).

9.3.4 Hybrid Approach: On-chip Memory Mapping Approach

In parallel decoder architectures, design time approaches require storing into ROM the control words to drive the network and to address data memories for particular block length or/and interleaving law. So, in order to design flexible decoder architectures that support multiple block lengths and multiple interleaving laws, several ROMs are needed (see Fig. 9.10) which results in an important hardware overhead.

In order to be flexible at a reasonable hardware cost (i.e., reduced memory footprint), a solution where the mapping algorithm is run on chip has been proposed.

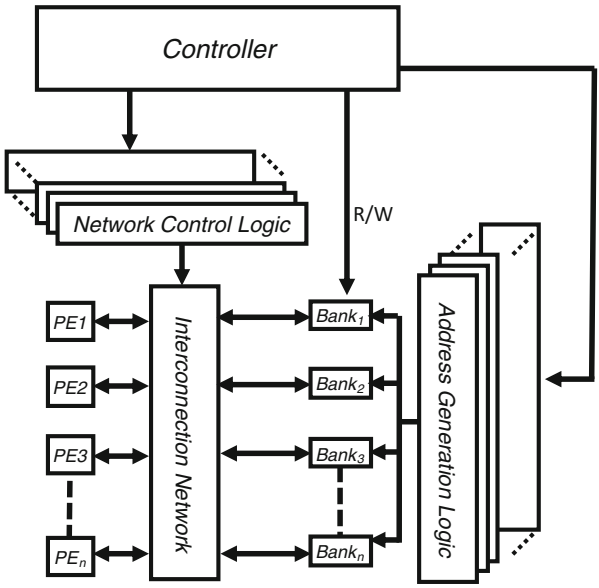

Fig. 9.10 Parallel decoder architecture supporting multiple block lengths

The approach starts by computing new mapping information on the fly as soon as a new block length needs to be decoded and updates these new generated control information in the memory.

Since this approach requires a fast on-chip mapping algorithm, then a novel polynomial time algorithm [29] derived from [30] is used for on-chip implementation. Low computational cost of this algorithm enables memory mapping approaches to implement this algorithm on chip using embedded processor, an ASIP or a dedicated hardware accelerator and to generate network and addressing control bits on-chip. This approach supports multiple standards and block lengths on single chip with reduced memory footprint.

The hardware architecture for embedded memory mapping is shown in Fig. 9.11. Control unit includes a dedicated processing element (General Purpose Processor GPP, Application Specific Instruction set Processor ASIP or Application Specific Integrated Circuit ASIC) to execute the mapping algorithm. The multiple networks and addressing ROMs are replaced by two RAMs, i.e., *Network RAM* and *Addressing RAM*. Control Unit executes the mapping algorithm and updates RAMs if required as soon as the decoder requires decoding a new block length or a new application (e.g., LTE, HSPA+, Wifi).

The execution flow is shown in Fig. 9.12. In the first step, the data access order is generated based on the particular interleaving law along with other required input parameters like block length, parallelism, and scheduling. This first step can be avoided if the designer wants to feed the system with pre-computed data consumption orders. This data access order is simply a scheduling of data accesses.

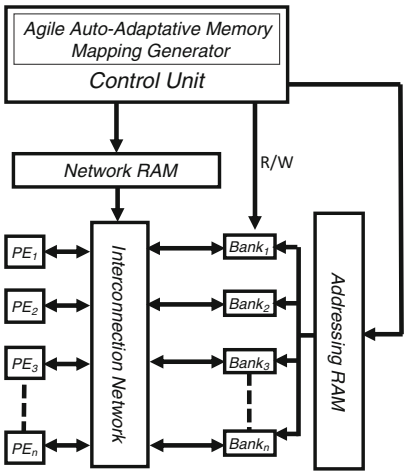

Fig. 9.11 Parallel decoder architecture to embed memory mapping algorithms on chip

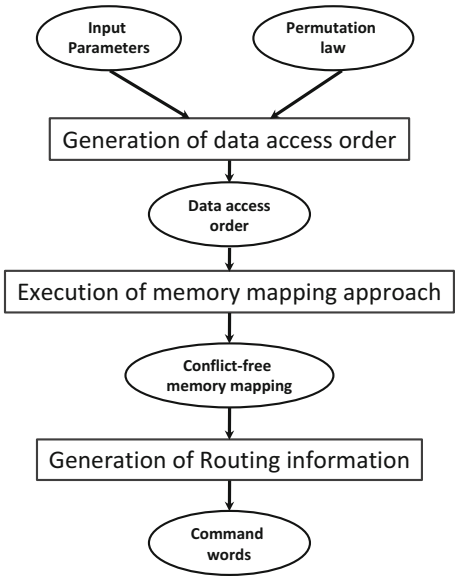

Fig. 9.12 Execution flow

However, the conflict-free memory mapping still need to be generated which is the goal of the second step of the design flow. Finally, the last step generates the routing and control information for the interconnection network and the memory banks.

Low computational cost of the mapping algorithm enables to implement this algorithm on chip using embedded processor, an ASIP or a dedicated hardware accelerator and to generate network and addressing control bits online. This approach will enable designers to support multiple standards and block lengths on single chip with reduced memory footprints.

The significant reduction in execution time and area obtained using this approach encourages embedding memory mapping and routing algorithm in future telecommunication devices.

Conclusion
In this chapter, we have presented a survey of existing solutions to deal with memory conflict accesses in parallel hardware decoder architectures of Turbo-codes and LDPC. The first type of approaches is based on definition "natively conflict-free" interleaving laws. A second family of approaches proposes to use dedicated hardware mechanisms to deal with conflicts at runtime, but with the expense of additional hardware components and latency. The third family of solutions proposes to find conflict-free memory mappings at design time, but at the expense of important memory footprint and hardware overhead when designing flexible decoders. Finally, a new class of approach that runs mapping algorithm on chip has been presented. In this context, a polynomial time memory mapping approach and a dedicated routing algorithm are embedded in the decoder architecture.

References

1. Shannon C (1948) A mathematical theory of communication. Bell Syst Tech J 27:379–423, 623–656
2. AHA. Reed-Solomon error correction codes (ECC). ANRS01-0404
3. Golay MJ (1961) Complementary series. IRE Trans Inf Theory IT-7:82–87
4. Hamming RW (1950) Error detecting and error correcting codes. Bell Syst Tech J XXVI(2):147–160
5. DVB (2008) Frame structure channel coding and modulation for the second generation digital terrestrial television broadcasting system (DVB-T2). DVB Document A122
6. WIFI (2008) Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: enhancements for higher throughput. IEEE P802.11n/D5.02, Part 11
7. WiMAX (2006) Air interface for fixed and mobile broadband wireless access systems – amendment 2: physical and medium access control layers for combined fixed and mobile operation in licensed bands, and corrigendum. IEEE P802.16e, Part 16
8. Berrou C, Glavieux A, Thitimajshima P (1993) Near Shannon limit error-correcting coding and decoding: turbo-codes. In: Proceedings of the IEEE international conference on communications (ICC 93), Geneva, pp 1064–1070
9. HSPA (2004) Technical specification group radio access network; multiplexing and channel coding (FDD). 3GPP, 25.212 V5.9.0
10. LTE (2008) Technical specification group radio access network; evolved universal terrestrial radio access; multiplexing and channel coding (release 8). 3GPP Std. TS 36.212
11. DVB-SH (2008) Digital video broadcasting (DVB); framing structure, channel coding and modulation for satellite services to handheld devices (SH) below 3 GHz. ETSI EN 302–583 V1.1.1
12. Briki A, Chavet C, Coussy P (2013) A conflict-free memory mapping approach to design parallel hardware interleaver architectures with optimized network and controller. In: IEEE workshop on signal processing systems, Tapei

13. Tarable A, Benedetto S, Montorsi G (2004) Mapping interleaving laws to parallel turbo and LDPC decoder architectures. IEEE Trans Inf Theory 50(9):2002–2009
14. Chavet C, Coussy P (2010) A memory mapping approach for parallel interleaver design with multiples read and write accesses. In: IEEE international symposium on circuits and systems, Paris, pp 3168–3171
15. Chavet C, Coussy P (2010) Static Address Generation Easing: a design methodology for parallel interleaver architecture. In: IEEE international conference on acoustics, speech and signal processing (ICASSP), pp 1594–1597
16. Giulietti A, van der Perre L, Strum M (2002) Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements. Electron Lett 38(5):232–234
17. Takeshita OY (2006) On maximum contention-free interleavers and permutation polynomials over integer rings. IEEE Trans Inf Theory 52(3):1249–1253
18. Berrou C, Saouter Y, Douillard C, Kerouédan S, Jézéquel M (2004) Designing good permutations for turbo codes: towards a single model. In: Proceedings of the IEEE international conference on communications (ICC 2004), pp 341–345
19. Woodard JP, Hanzo L (2000) Comparative study of turbo decoding techniques: an overview. IEEE Trans Veh Technol 49:2208–2233
20. Blankenship BC (2005) High-throughput turbo de-coding techniques for 4G. In: International conference on 3G wireless and beyond, pp 137–142
21. Gnaedig D, Boutillon E, Jezequel M, Gaudet VC, Gulak PG (2003) On multiple slice turbo codes. In: Proceedings of 3rd international symposium on turbo codes, pp 343–346
22. Mansour M, Shanbhag N (2003) High-throughput LDPC decoders. IEEE Trans VLSI Syst 11(6):976–996
23. Zhang T, Parhi KK (2001) Joint code and decoder design for implementation-oriented (3;k)-regular LDPC codes. Asilomar Conf Signals Syst Comput 2:1232–1236
24. Thul MJ, Gilbert F (2002) Optimized concurrent interleaving architecture for high-throughput turbo-decoding. In: Ninth international conference on electronics, circuits and systems, vol 3, pp 1099–1102
25. Wang G, Sun Y, Cavallaro JR, Guo Y (2011) High-throughput contention-free concurrent interleaver architecture for multi-standard turbo decoder. In: IEEE international conference on application-specific system, architectures and processors (ASAP) pp 113–121
26. Moussa H, Muller O (2007) Butterfly and Benes-based on-chip communication networks for multiprocessor turbo decoding. In: Design, automation and test in Europe, pp 654–659
27. Sani AH, Coussy P, Chavet C, Martin E (2011) An approach based on edge coloring of tripartite graph for designing parallel LDPC interleaver architecture. In: IEEE international symposium on circuits and systems
28. Sani AH, Coussy P, Chavet C, Martin E (2011) A methodology based on transportation problem modeling for designing parallel interleaver architectures. In: Proceedings of the 36th IEEE International Conference on Acoustics, Speech and Signal Processing, Prague, May 22-27, 2011
29. Ur Rehman S, Sani A, Chavet C, Coussy P (2014) Embedding polynomial time memory mapping and routing algorithms on-chip to design configurable decoder architecture. In: Ninth IEEE international conference on acoustics, speech and signal processing
30. Sani AH, Chavet C (2013) A first step toward on-chip memory mapping for parallel turbo and LDPC decoders: a polynomial time mapping algorithm. IEEE Trans Signal Process 61(16):4120–4140