A Reconfigurable Multi-Processor Platform for Convolutional and Turbo Decoding

Timo Vogt, Christian Neeb, and Norbert Wehn
University of Kaiserslautern, Kaiserslautern, Germany
{vogt, neeb, wehn}@eit.uni-kl.de *

Abstract

Future wireless communications networks require flexible modem architectures with high performance. Efficient utilization of application specific flexibility is key to fulfill these requirements. For high throughput a single processor can not provide the necessary computational power. Hence multi-processor architectures become necessary.

This paper presents a multi-processor platform based on a new dynamically reconfigurable application specific instruction set processor (dr-ASIP) for the application domain of channel decoding. Inherently parallel decoding tasks can be mapped onto individual processing nodes. The implied challenging inter-processor communication is efficiently handled by a Network-on-Chip (NoC) such that the throughput of each node is not degraded. The dr-ASIP features Viterbi and Log-MAP decoding for support of convolutional and turbo codes of more than 10 currently specified mobile and wireless standards. Furthermore, its flexibility allows for adaptation to future systems.

1. Introduction

Next generation mobile communication networks, beyond 3G (B3G), feature new services, especially multimedia applications, high data rates, and multi-access interoperability. The International Telecommunication Union expects that new radio access technologies will be integrated with already existing wireless and mobile networks like UMTS, WLAN, and DVB into a heterogeneous network. Seamless services with soft handover must be guarantied. Modem architectures must adapt to these diverse requirements and support different technologies and standards at the same time. *Thus flexibility becomes a dominant aspect for future modems.*

The focus of this paper is put on *channel coding in mobile and wireless communications systems*. Here convolutional codes (CC) and concatenated convolutional codes, also known as turbo codes (TC), are established techniques. Table 1 shows coding schemes used in various existing standards. Turbo codes have an outstanding forward error correction capability. They consist of concatenated component codes that work on the same block of information bits, separated by an interleaver. The component codes are recursive convolutional codes which are decoded individually. Key to the performance of turbo codes is the iterative exchange of interleaved information between the component decoders. For an introduction to turbo codes see [1]. Convolutional codes are decoded by the Viterbi algorithm (VA) or the Maximum A posteriori Probability (MAP) algorithm. The VA generates hard decision output, whereas the MAP is used if soft decision output is required, as in turbo codes. The computational complexity of both algorithms is high, with the MAP being about 3 times as complex as the VA.

The flexibility challenge can only be met by programmable or reconfigurable architectures. ASIC implementations are not suitable for adaptation of changes as required in B3G systems. FPGAs feature bit level flexibility, but the programming model is complex. A simple programming model and instruction level flexibility is provided by processors. Hence, efficient implementations of channel decoders on programmable architectures are of great importance to efficiently support the various existing or even emerging standards. Throughput on single processor architectures, however, is limited. Thus for high throughput applications the parallelism has to be increased on various levels, e.g. on instruction and multi-processor level.

The XiRisc [7] provides a RISC architecture enhanced by a reconfigurable, FPGA alike array that is tightly coupled with the RISC pipeline. However, the restrictions to the RISC pipeline structure, the load-store architecture, and the narrow bandwidth between the register file and the reconfigurable array limit the performance.

Efficient utilization of application specific parallelism and flexibility is key to powerful, efficient, and flexible architectures. High performance combined with the advantages of processors, namely instruction level flexibility and simple programming models, can be achieved by application specific instruction set processors (ASIP). In [8] an ASIP based on the Tensilica XTENSA platform targeting the channel coding domain was presented. It is based on a fixed RISC pipeline extended by application specific instructions. This platform is limited to a load-store RISC architecture with four pipeline stages.

Total freedom in pipeline and memory architecture design gives room for further improvement. Moreover, it allows to add application specific *run time reconfigurability* to the ASIP approach: the flexibility requirements of the application domain can be balanced between instruction level flexibility and reconfigurability, as explained in Section 5. An ASIP using this approach was proposed in [9], but it only targets the field of turbo codes with 8 states or less. Convolu-

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant WE 2442/4-1 within the Schwerpunktprogramm "Rekonfigurierbare Rechensysteme".

tional codes with high constraint lengths, which have similar computational complexity as turbo codes, and 16-state turbo codes are not supported.

In multi-processor implementations several independent data blocks can be decoded on different processors independently, which multiplies the costs (memories, area, and power consumption) along with the throughput. Latency can not be improved with this approach. Exploiting the inherent parallelism of the decoding algorithms enables a far more efficient partitioning of the decoding task: as will be shown later, the block to decode can be divided into several sub-blocks. Decoding each sub-block on an individual processor significantly reduces latency as a critical parameter in many communication applications, and memory overhead.

Due to the iterative exchange of interleaved data each processor working on the same data block has to communicate with each other, yielding only limited locality. A communication network has to support the communication demands of the different applications without degrading the throughput of the overall system.

In this paper we present a multi-processor platform for channel decoding based on a new dynamically reconfigurable application specific instruction set processor (dr-ASIP). The platform is scalable and provides the flexibility to allocate different decoding tasks to different processors. Thus it is possible to decode multiple convolutional and turbo codes in parallel on different hardware resources. The resource allocation can be adapted to application constraints like data rate or latency.

The dr-ASIP features Viterbi and Log-MAP processing for all possible binary convolutional codes with constraint lengths between 3 and 9, and code rates between 1 and 1/4 and supports convolutional and turbo decoding of standards as GSM, EDGE, UMTS, CDMA2k, IEEE802.11, IEEE802.16, HIPERLAN, DAB, DVB-H, DVB-S, and DVB-T.

Section 2 introduces convolutional codes and the decoding algorithms, followed an introduction to turbo codes. Section 4 summarizes the flexibility requirements for the decoder architecture from the application point of view. The ASIP architecture is presented in Section 5, followed by the extension to a multi-processor platform in Section 6. Section 7 concludes the paper.

2. Convolutional Codes

In convolutional codes forward error correction is enabled by introducing parity bits at the encoder. Figure 1 shows a generic binary convolutional encoder. Binary convolutional codes are characterized by a single binary input sequence (*information sequence*). In the remainder of this paper only binary convolutional codes are considered. They are fully specified by the constraint length $K_c = m+1$ with m the size of the shift register, a feedback polynomial G_{FB}, and generator polynomials G_i for the parity bits. Note that one output sequence can be equal to the input sequence: the systematic information. The number of output values per information bit defines the rate R of the code.

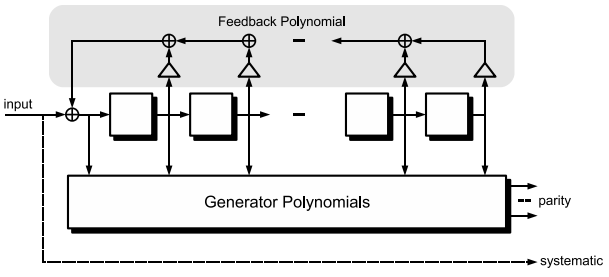

Figure 1. Generic binary convolutional encoder structure

In mobile and wireless communications systems a large variety of binary convolutional codes are used. The constraint length typically varies between $K_c = 5\ldots9$, and the number of generator polynomials between 1 and 4. The polynomials themselves basically assume arbitrary values.

Convolutional codes are also used as component codes in turbo codes. Here the constraint length typically is $K_c = 3\ldots5$.

Two decoding algorithms exist: the VA for Maximum Likelihood (ML) sequence decoding and hard output generation, and the Log-MAP algorithm for Maximum A posteriori Probability (MAP) symbol decoding and soft output generation. The algorithms are briefly discussed in the following subsections.

2.1. Viterbi Algorithm

The Viterbi Algorithm was introduced in 1967 as a method to decode convolutional codes [13]. Underlying is a discrete-time Markov chain with a finite number of N states S_m ($N = 2^{K_c-1}$). A transition takes place from the state at step k to a new state at step $k+1$ with a certain probability. The transition dynamics can be described by a *trellis*.

Given a received sequence of symbols, the VA finds the most likely **sequence** of state transitions. This is achieved by assigning a *transition* or *branch metric* to each state transition (a branch in the trellis) and selecting in each decoding cycle and for all states the path with the best sum of the transition metrics, called the *local survivor*. The decision bits dec_s indicating the local survivor for each state and each trellis cycle are stored in a survivor memory. In a proceeding traceback step the local survivors are read from this memory back in time to extract the most likely sequence of *final survivors*.

The *branch metric* $\gamma_{m,m'}^{k,k+1}(d^k)$ is the probability that a transition between S_m^k and $S_{m'}^{k+1}$ has taken place. It is derived from the received signals, the code structure and the assumption of $d^k = 0$ or $d^k = 1$, for details see[3].

The path metric λ_m is obtained by the so called ACS (Add Compare Select) recursion of the VA and can be described in the following way. For each state $S_{m'}^{k+1}$ and all its preceding states S_m^k, choose that path as optimum according to the following decision:

$$\texttt{for all } S_{m'}\texttt{:}\quad \lambda_{m'}^{k+1} = \max_{\texttt{all } m}(\lambda_m^k + \gamma_{m,m'}^{k,k+1}(d^k)) \qquad (1)$$

For binary convolutional codes each state in the trellis has two incoming and two outgoing branches. Hence, the ACS recursion can be divided in $N/2$ ACS butterflies. Each butterfly comprises 4 add and 2 compare and select operations.

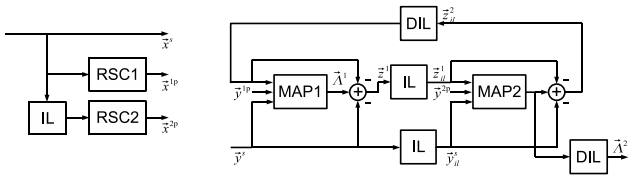

Figure 2. Turbo encoder and decoder

2.2. Log-MAP Algorithm

The MAP algorithm works on the same trellis as the VA algorithm. Given the received samples (*channel values*) for the whole block $y^{0..L-1}$ and potential a priori information, the MAP algorithm computes the probability for **each bit** to have been sent as $d^k = 0$ or $d^k = 1$. The logarithmic likelihood ratio (LLR) of these probabilities is the soft output of the decoder, denoted as:

$$\Lambda^k = \log \frac{\Pr\{d^k = 1 | y^{0..L-1}\}}{\Pr\{d^k = 0 | y^{0..L-1}\}}. \quad (2)$$

Equation 2 can be expressed using three probabilities, which refer to the encoder states S_m^k, where $k \in \{0 \dots L\}$ and $m, m' \in \{1 \dots 2^{K_c-1}\}$: the *branch metric* $\gamma_{m,m'}^{k,k+1}(d^k)$, and the *state metrics* α_m^k and $\beta_{m'}^{k+1}$.

The original probability based formulation involves a lot of multiplications and has thus been ported to the logarithmic domain to become the *Log-MAP Algorithm*[11]: multiplications turn into additions and additions into maximum selections with additional correction terms. The resulting *max** operation is defined as:

$$\max^*(\delta_1, \delta_2) = \max(\delta_1, \delta_2) + ln(1 + e^{-|\delta_2 - \delta_1|}).$$

This transformation does not decrease the communications performance. The new state metrics $\overline{\alpha}_{m'}^k$ and $\overline{\beta}_m^k$ are computed in a *forward* and *backward recursion*, respectively.

The order of α- and β-recursion execution is arbitrary. A lifetime analysis of the decoding algorithm shows that only one set of state metrics has to be stored during the first recursion, as the LLRs are computed in parallel to the second recursion.

Similar to the VA, windowing techniques allow for memory reduction at the expense of additional computations (*acquisition*). Several windows can thus be decoded sequentially on the same hardware for memory reduction as only the state metrics of a window have to be stored. Moreover, windowing allows to map sub-blocks to individual nodes for parallel processing, allowing to trade off hardware for latency. For further information on windowing refer to [4, 15].

3. Turbo Codes

In turbo codes, the original information ($\vec{x}^s$), denoted as *systematic information*, is transmitted together with the parity information ($\vec{x}^{1p}, \vec{x}^{2p}$). One RSC encoder works on the block of information in its original, the other one in an interleaved sequence, see Figure 2. On the receiver side a corresponding component decoder for each of them exists. The maximum a posteriori (MAP) decoder has been recognized as the component decoder of choice [14].

Standard	Codes	Rates	States	Blocksize	Throughput
GSM	CC	1/2..1/4	16, 64	33..876	..12kbps
EDGE	CC	1/2..1/3	64	39..870	5..62kbps
UMTS	CC	1/2..1/3	256	1..504	..32kbps
	TC	1/3	8	40..5114	..2Mbps
CDMA2k	CC	1/2..1/6	256	1..744	..38kbps
	TC	1/2..1/5	8	378..20736	..2Mbps
IEEE802.11	CC	3/4..1/2	64, 256	1..4095	6..54Mbps
IEEE802.16	CC	7/8..1/2	64	1..2040	..24Mbps
	TC	3/4..1/2	8	1..648	..24Mbps
Inmarsat	TC	1/2	16	..2608	..64kbps

Table 1. Selection of standards and channel codes

The soft output of each component decoder ($\vec{\Lambda}$) is modified to reflect only its own confidence (the extrinsic information $\vec{z}$) in the received information bit of being sent either as "0" or "1". These confidences are exchanged between the decoders to bias their next estimations iteratively (see Figure 2). During this exchange, the produced information is interleaved following the same scheme as in the encoder.

Windowing allows for parallel processing by breaking a data block into sub-blocks. At the borders of the sub-blocks accurate estimates are required for recursion initialization. These estimates can be computed locally by acquisition which requires data redundancy within the sub-blocks, or they are obtained by message passing between neighboring sub-blocks. These messages stem either from the actual iteration or from the previous one.

Interleaving is essential for the performance of turbo codes. Interleaver and deinterleaver tables contain one-to-one mappings of source addresses to destination addresses. One extrinsic value has to be read for each produced. Interleaving can be performed on the fly through indirect addressing for up to one value per clock cycle only. When more are produced, multiple extrinsic values have to be read and written concurrently. This is, *several* of them *have to be fetched* from *and stored* to memories *in the same clock cycle*. Without advanced communication schemes the resulting conflicts form the major bottleneck in parallel turbo decoding.

4. Application Requirements

Thorough investigation of 2G, 3G and upcoming 3.9G/4G mobile and wireless communication scenarios led to following requirements for a convolutional and turbo decoder platform (a selection of standards is summarized in Table 1):

- combined decoder for VA and Log-MAP decoding
- support of convolutional and turbo decoding
- support of $K_c = 3 \dots 9$, $N = 4 \dots 256$
- up to four channel values per information bit
- arbitrary but single feedback polynomial
- arbitrary generator polynomials
- high throughput (up to 100 Mbps for turbo applications)
- fast reconfigurability of channel code structure

The platform must be scalable to different scenarios like terminal or base station implementations.

5. ASIP design

5.1. General Considerations

Before designing the application specific processor, general architectural choices had to be made. The Log-MAP algorithm is computationally more expensive than the VA and will therefore be discussed first.

Various windowing schemes must be supported. Therefore the recursions are programmable meaning that the window size and acquisition length depend only on the number of instruction executions[1]. This gives flexibility for instance to adjust the acquisition length to the code structure and to the communication channel conditions. The different recursions are processed sequentially on the same hardware, and forward or backward recursion can be performed first. The soft output (either $\vec{\Lambda}$ or $\vec{z}$) is computed in parallel with the second recursion. One recursion step of turbo code applications is processed in a single cycle for high throughput support. Therefore it must be possible to read channel values, to process branch and state metrics in parallel, and to store multiple state metrics or soft output values, all at the same time. This requires data parallelism within the pipeline, and a customized memory architecture with high memory bandwidth. A maximum of 16 state metrics are computed in parallel. If N is larger than 16, the state metrics of a single trellis step have to be computed sequentially.

The channel code structure is at least constant for a whole data block. The overhead (area, energy, latency) of specifying it with each instruction is too high. On the other hand it is important to be able to switch within a few clock cycles from one channel code to another, for instance to support soft handover. Therefore the channel code configuration is not specified by the instructions but is kept dynamically reconfigurable within the dr-ASIP.

In case of the VA the path metrics computation and the trace back are performed sequentially. The path metric computation utilizes the same hardware as the forward state metric recursion of the Log-MAP algorithm.

For the implementation of ASIPs different frameworks are provided by EDA vendors. The major differences between these frameworks are that some offer instruction set extensions to a fixed programmable core (like Tensilica), whereas others give total freedom over the ASIP architecture (like CoWare). The latter allows to implement just the flexibility required with minimum hardware overhead. In this work the LISATek framework from CoWare Inc. was used.

5.2. Architecture Overview

Figure 3 shows the overall architecture of the dr-ASIP. It consists of the dr-ASIP core, memories, and an interleaver and deinterleaver unit (IL/DIL) connected to a network interface. Data is exchanged through the packet based network interface (see Section 6).

Several memories (one channel value (CV) memory and two IO memories) can be accessed by the dr-ASIP core as well as by the network interface. The IO memories are suited for storage of soft or hard output values, local survivors, and extrinsic information. Typically these memories are implemented by synchronous SRAMs. Their size can be tailored to the application and is only limited by the address space assigned to the different memories.

[1] An upper bound for the window size is given by the size of the state metric memory.

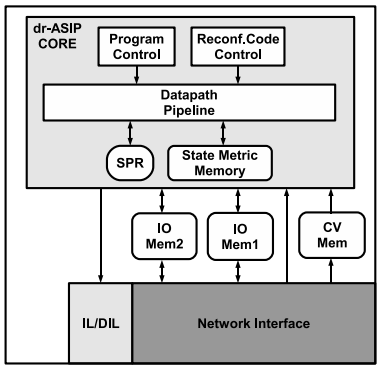

Figure 3. Overall dr-ASIP architecture

In multi-processor turbo decoder applications, the new extrinsic information has to be distributed to different target processors. This task is performed by the IL/DIL and the network interface. The IL/DIL maps the source address of the old extrinsic information to a target address before the data is sent to the network interface. The interface itself performs the message distribution. During decoding one IO-MEM, storing the old LLRs of the actual iteration, is read by the dr-ASIP core, while the other is filled with the received new extrinsic information for the next iteration.

The control part of the dr-ASIP core consists of two parts: program control and a dynamically reconfigurable channel code control. The program control supports two nested zero overhead loops (ZOL), branches, and limited interrupt services. Pipeline control is also implemented here. The channel code control specifies the channel code structure specific parameters. It is look up table (LUT) based and consists of two sets of LUTs: a working and a shadow configuration. Within a single clock cycle the shadow configuration can be transfered to the working configuration to support run time reconfigurability. The channel code control configures for instance the number of channel values that are read in parallel from the CV memory, and the generator polynomials of the convolutional code. The configuration memory consists of 629 bits.

The data manipulation part comprises a single data path pipeline, special purpose registers (SPR), and a state metric memory (SMM). The SMM is single ported and can store 16 state metric values in parallel. It holds the state metrics generated during the first recursion until they are consumed by the LLR computation during the second recursion. For convolutional codes with constraint lengths $K_c > 5$ the SMM is also used to store intermediate state metrics both during forward and backward recursion. The SPRs implement address generation for the MEM RD and MEM WB stages of the processor's pipeline (see Figure 4), especially the survivor memory read and write pointer generation during VA operation.

5.3. ASIP Pipeline

A more detailed representation of the processor's pipeline is depicted in Figure 4. It consists in total of 11 stages. The

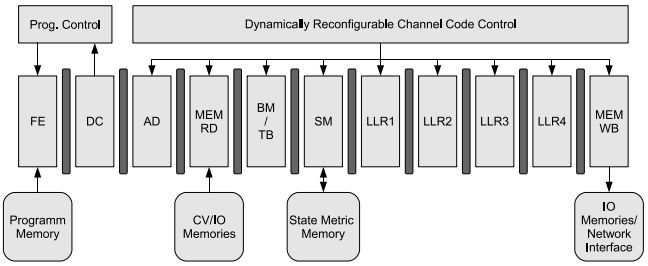

Figure 4. Processor pipeline with 11 stages: fetch (FE), decode (DC), address generation (AD), memory read (MEM RD), branch metric computation or trace back (BM/TB), state metric computation (SM), 4 LLR computation stages (LLR1...LLR4), memory write (MEM WB).

first two (FE, DC) fetch and decode the instructions and handle the program and pipeline control.

The functionality of the following stages is controlled by the decoded instruction as well as by the reconfigurable channel code control. The third stage (AD) generates memory read addresses for the MEM RD stage, the write address for the MEM WB stage, and the address for the SMM access. In MEM RD the channel values, a priori information or local survivors are read from the CV and IO memories. The remaining six stages perform the data manipulation for the VA and Log-MAP algorithm. They are optimized for a code with $K_c = 5$ and $R = 1/4$.

The BM/TB stage computes up to 16 branch metrics in parallel, or performs the trace back operation of the VA. The branch metrics are utilized in the SM stage to compute up to 16 state or path metrics in parallel. The metrics are directly forwarded from the output register of the SM pipeline stage to its input. This allows for fast computation of the metric recursions. If $K_c < 5$ parts of the pipeline are powered down. If $K_c > 5$ a load store architecture is implemented: intermediate state metrics are loaded from the SMM to a pipeline register, processed, and then stored back to the SMM. A single trellis cycle with $N = 256$ states can thus be computed in 16 consecutive steps. The soft-output computation of the Log-MAP algorithm is pipelined due to critical path reduction and is located in stages LLR1 to LLR4. These stages are idle during VA-operation. Due to the possibility of dynamic reconfiguration of the processor pipeline, any code structure with $3 \leq K_c \leq 9$ and rates between 1 and 1/4 are supported.

5.4. Instruction Set

Each instruction is 24 bit wide and includes opcode and address information for accessing the CV, IO or state metric memories. The in total 121 assembler instructions can be grouped in control, address modification, and algorithmic instructions.

A special control instruction loads the actual channel code configuration. It is important to flush the pipeline before loading a new configuration. As mentioned before, two nested ZOLs are implemented. Each allows to repeat a maximum of 512 instructions 512 times. The minimum number of instructions within a loop is two. It is also possible to define a loop that runs forever, it can only be exit by application of an interrupt. An unconditional branch instruction is also implemented and uses direct addressing for the branch destination.

The address modification instructions allow for setting or modifying the three SPRs that specify the addresses for the MEM RD and MEM WB stages of the pipeline. Thus absolute and relative addressing modes are supported. These address registers can also be modified, with restrictions, by the algorithmic instructions.

The algorithmic instructions can be grouped into VA instructions, and "single cycle" and "multi cycle" Log-MAP instructions. All of these instructions have in common that data is read from the CV or IO memories and that data is written back to the IO memories or directly to the network interface. The VA instructions either compute the path metrics or perform the trace back operation. The "single cycle" Log-MAP instructions can compute all state metrics of one trellis step in parallel. If the constraint length of the code is larger than 5, the "multi cycle" instructions allow for sequential processing of a single trellis step. Due to the ZOLs and the relative addressing mode it is possible to write very compact instruction code, resulting in a small program memory.

5.5. Implementation Results

The dr-ASIP core was implemented with the LISATek tool set, the generated VHDL model was synthesized with the Synopsys Design Compiler with a 65 nm low power standard cell library at 400 MHz under worst case conditions (1.05V, 125C). The total gate count of the dr-ASIP core without memories is 56250 gate equivalents (GE). The SMM for a maximum window size of 64 adds another 7867 GE. The data path pipeline alone accounts for 84.5% of the dr-ASIP core area. Compared to the processors of [8] and [9] with 104 kGE and 93 kGE for the core's logic, respectively, the dr-ASIP saves more than 40% of the area.

The throughput for the Log-MAP algorithm at 400 MHz clock frequency varies from 0.9 Mbits/s for $K_c = 9$ to 200 Mbits/s for $K_c \leq 5$, and from 12 Mbits/s to 133 Mbits/s for the VA, respectively. Table 2 summarizes Log-MAP decoder implementations for UMTS turbo code applications on different target platforms. The clock frequencies listed are maximum values. They differ, among other things, because the target technology is not the same. However, the dr-ASIP outperforms the other processor implementations even if they all run with the same clock frequency.

Standards supported by the dr-ASIP, among others, are GSM, EDGE, UMTS, CDMA2k, IEEE802.11, IEEE802.16, HIPERLAN, DAB, DVB-H, DVB-S, and DVB-T.

Platform	Architecture	Clock freq.	cycles/ (bit*MAP)	Throughput/ 5 iter
Processor STM ST120	GP-DSP VLIW[8]	200 MHz	≈ 100	≈ 0.2Mbps
XTENSA	Conf. RISC[8]	133 MHz	≈ 33	≈ 0.4Mbps
FPGA	VitexII-3000[10]	80 MHz	≈ 1	≈ 8Mbps
Reconf. Proc.	XiRisc[7]	100 MHz	≈ 100	≈ 0.1Mbps
ASIP	[9]	335 MHz	≈ 7.5	≈ 4.4Mbps
ASIP	dr-ASIP	400 MHz	≈ 2	≈ 20Mbps

Table 2. Comparison of different Log-MAP implementations for UMTS turbo code applications

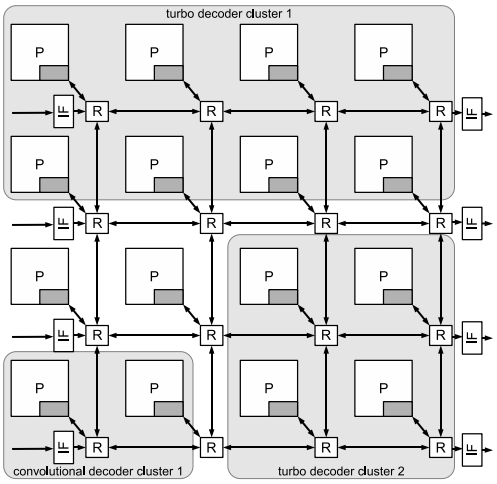

Figure 5. Example mapping of multiple decoding tasks on application specific multi-processor platform (AP-MPSoC)

6. Application specific multi-processor platform

As already mentioned, future communication standards will demand for very high data rates with constrained latency. Hence, only increasing the instruction level parallelism of a single processor core is not sufficient, and we have to move to massively parallel multi-processor architectures, so-called application specific multi-processor systems-on-chip (AP-MPSoC). In [6] a flexible multi-processor architecture for parallel turbo decoding was presented for the first time. Customized RISC cores are augmented with application specific hardware accelerators to increase computational power. However, the flexibility to support multiple coding standards is rather limited and only a subset of turbo decoding applications is supported at moderate performance.

A simple solution to increase the degree of parallelism is to decode independent data blocks on different processors independently. However, such a solution does not decrease the decoding latency. Since latency is a very critical issue such a solution is infeasible in many cases. Furthermore, due to the limited size of the processors local IO memory, the support of large block sizes can only be achieved by splitting it into several sub-blocks. Thus, we decompose the algorithm itself into a set of parallel tasks running on a set of communicating dr-ASIP cores forming processing clusters. We exploit windowing to break up a complete block of length L into n smaller sub-blocks of length B, where each sub-block is mapped onto one of the n cores in the platform.

The arising need for data communication in parallel architectures is efficiently realized by a Network-on-Chip [2] [12]. It allows to share physical bandwidth among different classes of data like IO channel data or concurrent interleaving data during decoding. The dimensioning of an application tailored network architecture is a crucial step in the MPSoC design which otherwise might lead to poor performance of the entire platform.

6.1. Architecture Template

Figure 5 shows the arrangement of the dr-ASIP cores integrated by a packet switched Network-on-Chip (NoC). It allows the dynamic mapping of independent decoding tasks onto a single dr-ASIP processor or onto multiple processors

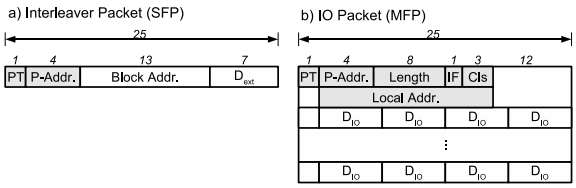

Figure 6. Two packet types: a) single flit packet (SFP), b) variable length packets (MFP)

grouped in a cluster. For applications demanding for very high throughput with tight latency constraints, all processors can be assigned to a single decoding task. An example mapping of two independent turbo decoders and one convolutional decoder is depicted in Figure 5. The platform offers a high flexibility to adapt to situations of changing decoding requirements and workloads.

Each of the dr-ASIP cores is directly attached to a network *router* (R) which implements the required communication services. The routers are interconnected by bidirectional point-to-point channels forming a 2D mesh topology. For performance reasons, we provide multiple *IO interfaces* (IF) to the environment which are directly connected to the boundary routers of the network. They allow the adaption of different communication protocols like OCP or Amba AXI to the optimized internal network protocol.

Typically, IO data form long data streams because entire blocks must be completely loaded and stored by a processor before decoding. This is in contrast to the exchange of interleaving data where very small chunks of data must be transferred at high data rates. We pay attention to this by the use of two types of network packets depicted in Figure 6. The first one serves for interleaving purpose and consists of a single flit only (SFP) containing all header information and a small data payload. A flit (flow control digit) represents the atomic data unit of transfer in our architecture and so determines the width of a physical channel. The header comprises the target processors address, a block address and the extrinsic information itself. For the remaining data, a variable size packet format (MFP) is used which consists of at least three flits. The first two are header flits specifying, in addition to the target processor and a local address information, the packet *length* in terms of flits and a packet classifier (*cls*) for processor internal purpose. The payload can include up to four input channel or soft output values per flit, program instructions, configuration data, state metrics, or interleaver information. All flits are 25 bit wide, and a control bit (*PT*) differentiates between the two packet types.

We use the concept of *virtual networks* [5] to control the allocation of network bandwidth separately for the two traffic classes. This is reflected in the data path of the routers where packets located in two separate data queues form independent virtual channels. These virtual channels are then multiplexed over a common physical channel by means of a crossbar switch.

6.2. Network Traffic Estimation

To quantify the resulting network traffic, we analyze the different processing and communication phases for the parallel turbo and convolutional decoding as depicted in Figure 7. The amount of data for program code and configuration of

the code parameters is assumed to be small compared to the IO and interleaving data if reconfiguration occurs rarely. We thus neglect it in the following discussion.

For both decoding applications, the processors must fetch the block data of length $B \cdot R$ from the interface and store them in their internal memory before the actual decoding can be started. For turbo decoding, B additional entries of the interleaver tables have to be loaded if the block length has changed.

In the following, a fixed number of k iterations is performed by each processor. During each Log-MAP iteration, every dr-ASIP processes its data split into q windows in a sequential loop. During the backward recursion phase (bwd), a burst of extrinsic values $\vec{\Lambda}$ is calculated by each of the n parallel cores. These values must be exchanged over the network for interleaving as already mentioned before. The competition of packet transfers for network resources leads to heavy network congestion in this phase. Thus only a fraction of packets can be delivered in one clock cycle which causes a degradation of the theoretical network throughput. Every time the network cannot accept the injection of a new packet, the processing core must be either stalled or the packet must be buffered temporarily to prevent data loss. We prefer the latter approach as it does not reduce the computational throughput of the core at the expense of extra buffer storage. The employed windowing technique allows the start of the next sub-iteration in parallel to interleaving which efficiently hides its latency $lat_{IL}(n)$. Only after the last sub-iteration $q-1$ this leads to an idle phase which has to be finished before the next iteration can be started due to data dependencies. Meanwhile, the exchange of the state metrics for sub-block initialization (smx) can be done in parallel but usually takes a much shorter time to finish. We have to make sure that the exchange of interleaving data can be achieved within a time which is smaller than that required to finish a single sub-iteration, such that $lat_{IL} \leq W + acq$. Otherwise the capacity of the employed network is too small blocking all cores in subsequent iterations for a considerable amount of time.

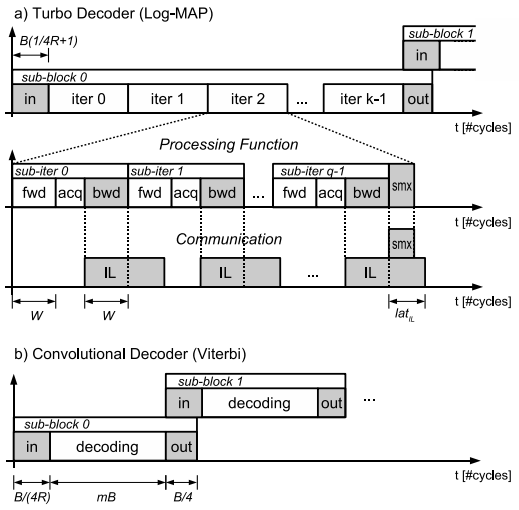

Figure 7. Processing and communication phases of a) turbo decoder and b) convolutional decoder

After the last iteration $k-1$ has finished, the decoded data block is read out of the cores IO memory and sent to the egress interface. This can be parallelized with the loading of the next data block which is stored in the second IO memory (see Section 5.2). Note that the decoded data does not include redundancy specified by the code rate R as the input data and thus leads to much lower output data rates.

To quantify network traffic for the worst case, we first estimate the data rates originating from both IO-traffic $\rho_{IN,OUT}$ as well as from interleaving ρ_{IL} with respect to a single processor core:

$$\rho_{IN}^{turbo} = \frac{B(\frac{1}{4R}+1)}{B(\frac{1}{4R}+1)+k[\frac{B}{W}(2W+acq)+max\{smx, lat_{IL}(n)\}]}$$
$$= \frac{1}{54+\frac{20}{B}max\{smx, lat_{IL}(n)\}} \leq 0.15 \quad (3)$$

$$\rho_{OUT}^{turbo} = \frac{B/4}{B(\frac{1}{4R}+1)+k[\frac{B}{W}(2W+acq)+max\{smx, lat_{IL}(n)\}]}$$
$$= \frac{1}{6.78+\frac{5}{2B}max\{smx, lat_{IL}(n)\}} \leq 0.02 \quad (4)$$

$$\rho_{IL}^{turbo} \geq \frac{W}{2W+acq} = 0.43 \quad (5)$$

$$\rho_{IN}^{conv} = \frac{1}{1+4Rm} \leq 0.25 \quad (6)$$

$$\rho_{OUT}^{conv} = \rho_{IN} R \leq 0.06, \quad \text{with} \quad (7)$$

$$R = 1/4, \quad k = 5, \quad W = 64, \quad acq = 20, \quad smx = 16, m = 3,$$
$$lat_{IL} = W + acq = 84$$

All data rates are normalized to the raw bandwidth of a single point-to-point channel making it independent of the actual channel width, such that $0 \leq \rho \leq 1$ holds. Doing so, data rates represent the average number of flits produced by a node per clock cycle. As can be seen easily from the above equations, interleaving generates a multiple of data compared to the IO data rates of both types of codes. As stated above, the network must at least be able to provide this normalized throughput ρ_{IL}^{turbo} in average, otherwise decoding throughput would suffer poor performance due to stalling.

The ensure that a 2D mesh network offers sufficient bandwidth for all configurations, we model it as a directed graph $I(R,C)$, where a vertex $r_i \in R$ represents a router and a directed weighted edge $c_{i,j}$ a (physical) communication channel between its incident routers r_i and r_j. An associated edge weight $\tau(c_{i,j}) = \tau_{i,j}$ represents the average channel traffic measured in flits per clock cycle $0 \leq \tau_{i,j} \leq 1$. Figure 8 illustrates the derivation of the network traffic. Every communication between any two nodes r_i, r_j is mapped to at least one routing path according to the employed routing algorithm. This increases the average traffic $\tau_{i,j}$ on all channels belonging to the routing path. Consequently, network traffic depends on the choice of the topology, the routing algorithm and the communication pattern of the processors. In the case of interleaving, a distributed block permutation is performed where all nodes must communicate with all other nodes with equal probability. This reflects the behavior of good interleavers where data is scrambled randomly inside the data block. From this point of view, a random uniform model adequately models the communication pattern of the decoder cores for interleaving. Hence, all packets have equal probability $1/n$ to be sent to a specific target processor.

To estimate network traffic caused by interleaving, we refer to the network's *bisection bandwidth*. It defines the aggregate bandwidth of a minimum cut which divides the network

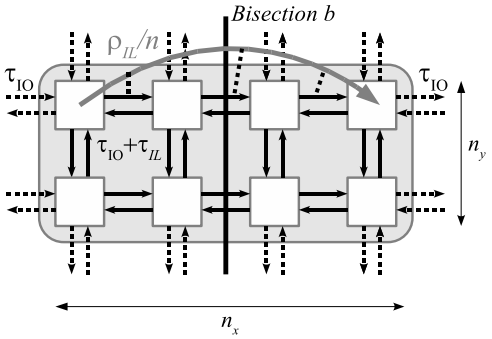

Figure 8. Derivation of network traffic for interleaving inside a turbo decoder cluster comprising $n_x \cdot n_y$ dr-ASIP cores

into two equal node sets and, accordingly, the *bisection width* b as the number of channels that have to be cut. Due to the regular construction and symmetry of the 2D mesh, the bisection width b can easily be derived as $b = 2 \cdot min\{n_x, n_y\}$, with $n = n_x n_y$. It is always orthogonal to the dimension containing most of the nodes. Due to the uniform communication pattern, $n\rho_{IL}/2$ interleaver packets have to cross this bisection during each cycle in average. The bisection channels carry the maximum of the interleaver traffic:

$$\tau_{IL,max}^{turbo} = \frac{n\rho_{IL}^{turbo}}{2 \cdot 2min\{n_x, n_y\}} = 0.11\, max\{n_x, n_y\} \quad (8)$$

$$\tau = \tau_{IL,max}^{turbo} + \tau_{IN} + \tau_{OUT} \leq 1 \quad (9)$$

The above equation restrict the possible shapes of the turbo processing clusters. Less than nine cores may be grouped along any dimension if no IO traffic arises during interleaving crossing the cluster. This is always true if all cores are configured to process only one turbo code in parallel where the IO phase and interleaving never occur simultaneously. For a quadratic arrangement of the $N = 16$ cores in a 2D mesh no configuration exists that violates the above requirement.

7. Conclusion

Application specific flexibility is mandatory to meet the flexibility and performance requirements of B3G communications systems. It can be achieved by application specific instruction set processors with specialized pipeline topology and dedicated communication and memory infrastructure. Dynamic reconfigurability is necessary to switch during run time between different coding schemes. In this paper we presented a dynamically reconfigurable ASIP for the application domain of channel decoding (dr-ASIP). It features Viterbi and Log-MAP processing for binary convolutional codes with constraint lengths between 3 and 9, code rates between 1 and 1/4, and arbitrary feedback and generator polynomials. Convolutional and turbo decoding for more than 10 currently specified mobile and wireless standards is supported.

For high-throughput decoding, we proposed a reconfigurable application specific multi-processor platform (AP-MPSoC) as a natural transition to parallel decoding. Here, multiple dr-ASIP cores can be configured to form parallel processing clusters enabling low latency decoding. As the inter-processor communication becomes the bottleneck for high degrees of parallelization, we presented a Network-on-Chip approach to efficiently interconnect the dr-ASIP cores. By a detailed traffic analysis we showed that the overall processing performance is not degraded by the network's limited communication bandwidth.

Future work will focus on increasing the flexibility to support other channel codes like duo-binary turbo and LDPC codes.

References

[1] S. A. Barbulescu and S. S. Pietrobon. Turbo Codes: A Tutorial on a New Class of Powerful Error Correcting Coding Schemes, Part 1: Code Structures and Interleaver Design. *Journal of Electrical and Electronics Engineering, Australia*, 19(3):129–142, Sept. 1999.

[2] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 35(1):70–78, Jan. 2002.

[3] M. Bossert. *Kanalcodierung*. B.G.Teubner Stuttgart, 1998.

[4] H. Dawid and H. Meyr. Real-Time Algorithms and VLSI Architectures for Soft Output MAP Convolutional Decoding. In *Proc. 1995 International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC '95)*, pages 193–197, Toronto, Canada, Sept. 1995.

[5] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, 1997, San Francisco, California, USA, 1997.

[6] F. Gilbert, M. Thul, and N. Wehn. Communication Centric Architectures for Turbo-Decoding on Embedded Multiprocessors. In *2003 Design, Automation and Test in Europe (DATE '03)*, pages 356–361, Munich, Germany, Mar. 2003.

[7] A. LaRosa, C. Passerone, F. Gregoretti, and L. Lavagno. Implementation of a UMTS Turbo-Decoder on a dynamically reconfigurable platform. In *Proc. 2004 Design, Automation and Test in Europe (DATE '04)*, Paris, France, Feb. 2004.

[8] H. Michel, A. Worm, Münch, and N. Wehn. Hardware/Software Trade-offs for Advanced 3G Channel Coding. In *Proc. 2002 Design, Automation and Test in Europe (DATE '02)*, Paris, France, Mar. 2002.

[9] O. Muller, A. Baghdadi, and M. Jezequel. ASIP-Based Multiprocessor SoC Design for Simple and Double Binary Turbo Decoding. In *Proc. 2006 Design, Automation and Test in Europe (DATE '06)*, Munich, Germany, Mar. 2006.

[10] P. Paulin, J.-M. Balzano, A. Silburt, K. V. Berkel, R. Bramley, and N. Wehn. Panel: "Chips of the Future: Soft, Crunchy or Hard?". In *Proc. 2004 Design Automation and Test in Europe (DATE '04)*, Paris, France, Feb. 2004.

[11] P. Robertson, P. Hoeher, and E. Villebrun. Optimal and Sub-Optimal Maximum a Posteriori Algorithms Suitable for Turbo Decoding. *European Transactions on Telecommunications (ETT)*, 8(2):119–125, March–April 1997.

[12] M. J. Thul, C. Neeb, and N. Wehn. Network-on-Chip-Centric Approach to Interleaving in High Throughput Channel Decoders. In *Proc. 2005 IEEE International Symposium on Circuits and Systems (ISCAS '05)*, Kobe, Japan, May 2005.

[13] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, Apr. 1967.

[14] J. Vogt, K. Koora, A. Finger, and G. Fettweis. Comparison of Different Turbo Decoder Realizations for IMT-2000. In *Proc. 1999 Global Telecommunications Conference (Globecom '99)*, volume 5, pages 2704–2708, Rio de Janeiro, Brazil, Dec. 1999.

[15] Y. Zhang and K. K. Parhi. Parallel Turbo Decoding. In *Proc. 2004 International Symposium on Circuits and Systems (ISCAS '04)*, pages II–509–II–512, Vancouver, Canada, May 2004.