Chapter 1

LOW-POWER DESIGN OF TURBO DECODER WITH EXPLORATION OF ENERGY-THROUGHPUT TRADE-OFF

Arnout Vandecappelle
Arnout.Vandecappelle@imec.be

Bruno Bougard
Bruno.Bougard@imec.be

K.C. Shashidhar
Shashidhar.Kodamballi@imec.be

Francky Catthoor
Francky.Catthoor@imec.be

IMEC vzw
Kapeldreef 75
3001 Leuven
Belgium

Abstract Turbo coding has become an attractive scheme for design of current communication systems, providing near optimal bit error rates for data transmission at low signal to noise ratios. However, it is as yet unsuitable for use in high data rate mobile systems owing to the high energy consumption of the decoder scheme. Due to the data dominated nature of the decoder, a memory organization providing sufficient bandwidth is the main bottleneck for energy. We have systematically optimized the memory organization's energy consumption using our Data Transfer and Storage Exploration methodology. This chapter discusses the exploration of the energy versus throughput trade-off for the turbo decoder module, which was obtained using our storage bandwidth optimization tool.

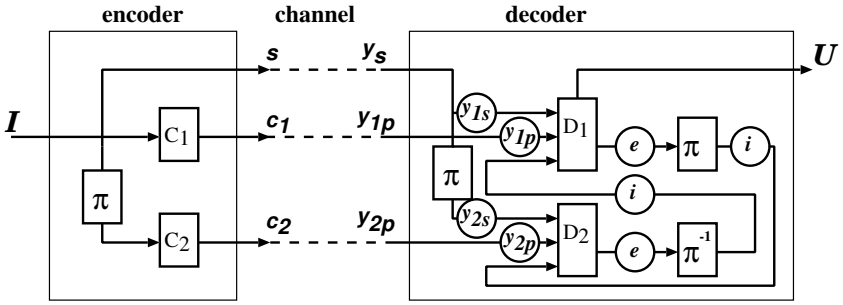


Figure 1.1. Turbo coding-decoding scheme

1. Introduction

Faithful replication of a transmitted signal at the receiver depends mainly on the performance limitation of the coding scheme employed. Most known traditional coding schemes have fallen short of the theoretical bound laid down by Shannon [16] on what best can be achieved. It is only fairly recently that a new scheme, called *turbo coding* [2], has enabled reliable data transmission at very low signal-to-noise ratios. Since it outperforms all previously known forward error correction codes, the technique has naturally become an attractive choice for design of current communication systems.

The main drawback of the turbo coding scheme, notwithstanding its near optimal performance, is the large latency and energy consumption of the decoder. This is due to the large number of memory accesses of the highly data dominated, iterative decoding scheme. Hence, memory organization is a very critical requirement for an efficient implementation of the scheme. Data Transfer and Storage Exploration (DTSE) [5] is a systematic methodology for optimization of memory accesses in such a data dominated application.

Many flavors of the turbo coding scheme can be seen in recent literature. We consider here the parallel concatenated convolution coding scheme [2], as shown in Figure 1.1. The data bit stream I is encoded by two convolution encoders C_1 and C_2. The input of C_2 is first interleaved by the interleaver Π. The encoded signals are transmitted over the channel, where noise gets added on to the signals. At the receiver, the signals are fed to the Soft-Input Soft-Output (SISO) decoders D_1 and D_2. A SISO produces *extrinsic* information (e); this is calculated from the uncoded, *systematic* input (y_s), the *coded* input (y_p), as well as the previously calculated *a priori* information (i). i is simply the (de)interleaved extrinsic information from the other SISO. The decoder is essentially an iterative scheme, wherein the extrinsic output from one decoder is fed to the other decoder as a priori information, and the process repeated until the required degree of convergence is achieved.

We consider here as SISO algorithm the Maximum A Posteriori (MAP) algorithm [1], which is the theoretically optimal algorithm to soft-decode the component codes. We use the *log-max* variant of the MAP [14], which is defined by the following equations:

$$\bar{\alpha}_k(m') = max_m(\bar{\alpha}_{k-1}(m) + \bar{\gamma}(S_k^m, S_{k+1}^{m'})) \quad (1.1)$$

$$\bar{\beta}_k(m) = max_m(\bar{\beta}_{k+1}(m') + \bar{\gamma}(S_k^m, S_{k+1}^{m'})) \quad (1.2)$$

$$\begin{aligned} LLR(d_k) &= max_{m,m'}(\bar{\gamma}(S_k^m, S_{k+1}^{m'}, d_k = 1) + \bar{\alpha}_k(m) + \bar{\beta}_{k+1}(m')) \\ &- max_{m,m'}(\bar{\gamma}(S_k^m, S_{k+1}^{m'}, d_k = 0) + \bar{\alpha}_k(m) + \bar{\beta}_{k+1}(m')) \end{aligned} \quad (1.3)$$

where $\bar{\alpha}$ is the forward state metric vector, $\bar{\beta}$ is the backward state metric vector, $\bar{\gamma}$ the branch metric vector; S_k^m represents the state m at time k, and d_k the transmitted bit. Branch metrics $\bar{\gamma}$ are computed as a function of the *systematic* (y_s), *coded* (y_p) and *a priori* (i) information. $LLR(d_k)$ is the log-likelihood ratio of the transmitted bit.

Issues relating to VLSI implementation of the turbo coding scheme have been addressed in [13, 17] and low complexity implementations to reduce energy consumption have been provided in [7, 8, 11], and many others. However, memory organization and optimization issues are not sufficiently addressed in literature. Memory access optimization, by applying early transformation steps of the DTSE methodology, has been investigated and important gains thereof have been shown in [12, 15]. However, the latter result is a single design point. The global energy and performance trade-off possibilities with respect to memory bandwidth, which are crucial for a designer to make a sound choice of the memory architecture, are as yet not available. In this work, our storage bandwidth optimization tool is demonstrated to provide such trade-off points in an implementation of the turbo decoder module. This is compared with a previous, manual design of the memory organization [9, 12].

In contrast to traditional design tools, which produce a *single optimal* solution, this trade-off allows the designer to explore the design space. For the turbo decoder, latency and throughput can be improved by sacrificing some energy consumption, or conversely more energy can be saved by lowering the performance. The designer can only efficiently decide on these trade-offs if there is a way to visualize them. Therefore, we propose Pareto plots, which are generated automatically, to estimate the energy cost of achieving a particular throughput.

This chapter is organized as follows. Section 1.2 briefly introduces the DTSE methodology and positions our work in it. Section 1.3 summarizes the platform-independent steps previously applied. Section 1.4 discusses the transformations oriented towards improving the storage bandwidth, and thus the throughput. Section 1.5 discusses the details of the memory organization. Section 1.6 draws conclusions.

2. Data Transfer and Storage Exploration Methodology

In data dominated applications, typically found in the multi-media and telecommunications domain, data storage and transfers are *the* most important factors in terms of energy consumption, area and system performance. DTSE is a systematic, step-wise, system-level methodology to optimize data dominated applications for memory accesses, and hence, energy consumption [5, 6]. The main goal of the methodology is to start from the specification of the application (for example in the C language) and transform the code to achieve an optimal execution order for data transfers, together with an optimal memory architecture for data storage.

The DTSE methodology is explained at length along with case studies in [5, 6], but to position our work, we briefly summarize it here. It essentially consists of the following *orthogonal* steps that are sequentially applied:

Global Data Flow Transformations The set of data flow transformations applied in this step have the most crucial effect on the system exploration decisions. Two main categories exist. The first one directly optimizes the important DTSE cost factors by removing redundant accesses. The second category serves as enabling transformations for the subsequent steps by removing the data flow bottlenecks.

Global Loop Transformations The loop and control flow transformations in this step aim at improving the data access locality for multi-dimensional arrays and at removing the system-level buffers introduced due to mismatches in production and consumption ordering.

Data Reuse Decisions The goal of this step is to better exploit a hierarchical memory organization by making use of available temporal locality in the data accesses. The result is that frequently accessed data is available in smaller and hence, less power consuming memories.

Storage Cycle Budget Distribution Application of this step results in distribution of the available cycle budget over the iterative parts of the specification in a globally balanced way, such that the required memory bandwidth is reduced.

Memory Allocation and Assignment The goal of this step is to select memory modules from a memory library and to assign the data to the best suited memory modules under the given cycle budget and other timing constraints.

In-place Optimization In this step, optimal placement of data in the memories is determined such that the required memory size is minimal and cache conflict misses are reduced.

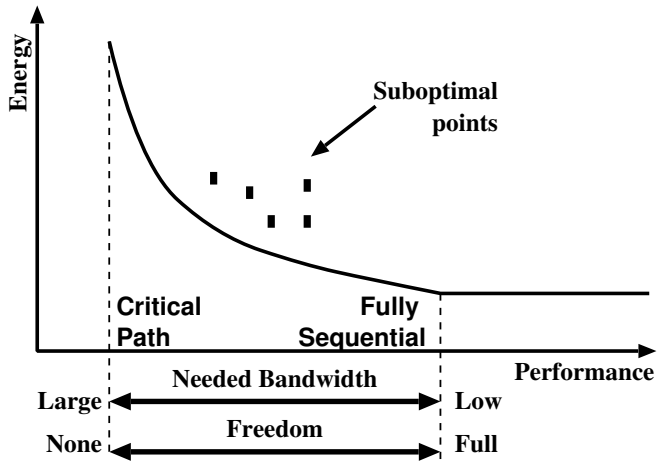
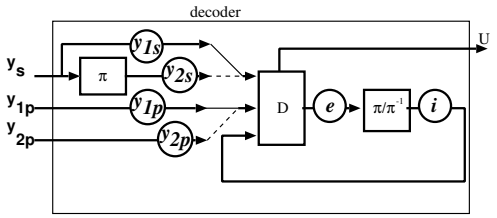


Figure 1.2. Energy-performance trade-off

The first three steps of the methodology have been discussed in previous work and are summarized in Section 1.3. This chapter completes the script with the two memory organization steps: storage cycle budget distribution is discussed in Section 1.4, and memory allocation and assignment in Section 1.5. These steps make it possible to identify a range of energy-performance trade-off possibilities. Step 6, as mentioned, optimizes the area consumption and the cache misses, but these are not bottlenecks in our implementation.

Figure 1.2 shows how the exploration points are represented. The vertical axis shows the memory organization's energy consumption (or another relevant cost measure), while the horizontal axis shows the performance (e.g. the cycle budget, throughput, or latency). The cycle budget ranges from fully sequential, where all memory accesses are scheduled one after the other, to the critical path schedule, where memory accesses are sequential only if there is a dependency between them. A smaller cycle budget means more parallel accesses, which will lead to a more complex memory organization with a higher energy consumption. Each possible memory architecture entails a certain energy consumption and a certain performance. The interesting trade-off points (called Pareto-optimal points) are those indicated by the curve, which are not dominated by any other point in both energy and performance. With such a Pareto curve, the designer can ignore the points which are clearly sub-optimal and concentrate on the interesting trade-off points. The purpose of the memory organization tools is to find precisely these trade-off points, without exploring all non-Pareto-optimal possibilities.




```
/* first SISO */
for (i)
 apriori[i] =
  ...(apriori[i])...

/* second SISO */
for (i)
 addr = interleaver[i]
 apriori[addr] =
  ...(apriori[addr])...
```

Figure 1.3. Transformed data flow of turbo decoding scheme

3. Global Data Flow and Loop Transformations

The first 3 steps of the DTSE methodology are essentially platform independent source code transformation steps. These have previously been applied on the turbo decoder [15, 12]. This section gives an overview of the results of these transformations, to facilitate the understanding of the later sections.

3.1 Removal of interleaver memory

The turbo coding paradigm as proposed by Berrou [2] is depicted in figure 1.1. This paradigm considers two SISO decoders and two interleavers. If the same code is used for both component encoders, which is the case for most of the currently standardized turbo coding schemes (e.g. UMTS, DVB-RCS), the turbo decoding can be done with only one SISO decoder. Moreover, the required storage and transfers for the interleaving and de-interleaving operations can be reduced by not storing the extrinsic information e and subsequently interleaving them, but directly writing the a priori information i in the correct place.

In addition, the same memory can be used for the a priori input of the SISO decoder and the extrinsic output, by storing the a priori/extrinsic information in a linear way. During the first half-iteration, corresponding to D_1 in the paradigm, the SISO decoder reads the a priori information linearly in the storage unit and writes the extrinsic information back in the same place, overwriting the value which is not needed anymore. During the second half-iteration (D_2), the SISO decoder reads the a priori information in the storage units at addresses generated by a look-up table storing the interleaving pattern, which emulates the interleaving of the paradigm. The extrinsic information is written back at the addresses of their corresponding a priori information, which emulates the de-interleaving process of the paradigm. The optimized data flow is depicted in Figure 1.3, together with a code example implementing this interleaved addressing.

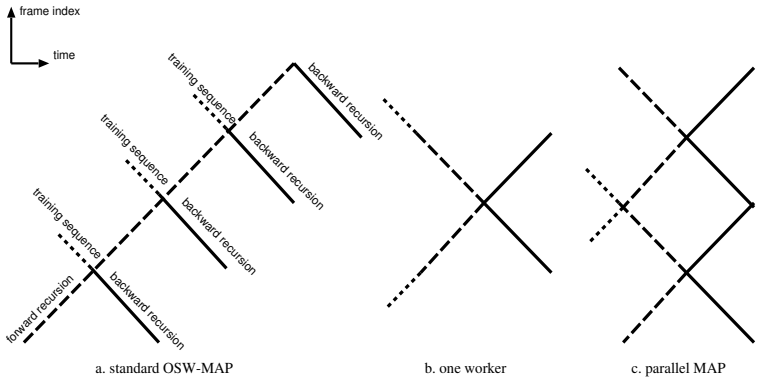


Figure 1.4. Parallelization of the MAP algorithm

3.2 Enabling parallelism

With the overhead of the interleaving removed by the above dataflow transformation, the double recursion ($\bar{\alpha}$ and $\bar{\beta}$ calculation) of the SISO module becomes the bottleneck of the design regarding latency as well as energy consumption. Usually, this latency bottleneck is broken using the so-called *overlapping sliding windows* MAP algorithms. The frame to be decoded is subdivided into windows of size N. The forward recursion ($\bar{\alpha}$ calculation) starts at the beginning of the window as in the normal algorithm. However, the backward recursion ($\bar{\beta}$ calculation) already starts at the end of the window instead of at the end of the frame. A training sequence then initializes the state metrics (β metrics) at the beginning of each window. A lower bound on the window size N is imposed by the minimum training sequence length. This length has to be at least 4 times the constraint length (K) of the code. Since we use a code with $K = 4$, N has to be at least 16.

Figure 1.4.a depicts the timing of the overlapping sliding windows scheme. The latency of the decoder is the time between the first bit of input and the first bit of output of the entire decoder. It is reduced by a factor roughly equal to the number of windows. However, the throughput stays unchanged. To increase the throughput, we proposed [15] to unroll the windows loop, i.e. calculating several windows in parallel. To reduce the required number of training sequences, adjacent windows are combined by pairs in structures called *workers* (Figure 1.4.b). The forward recursion of one window is used as training sequence for the (delayed) forward recursion of the adjacent window, and vice versa for the backward recursion. The workers can be combined in parallel as depicted in Figure 1.4.c. With such a structure, the decoding latency is no longer proportional to the frame size, but rather to the worker size ($2 \times N$).

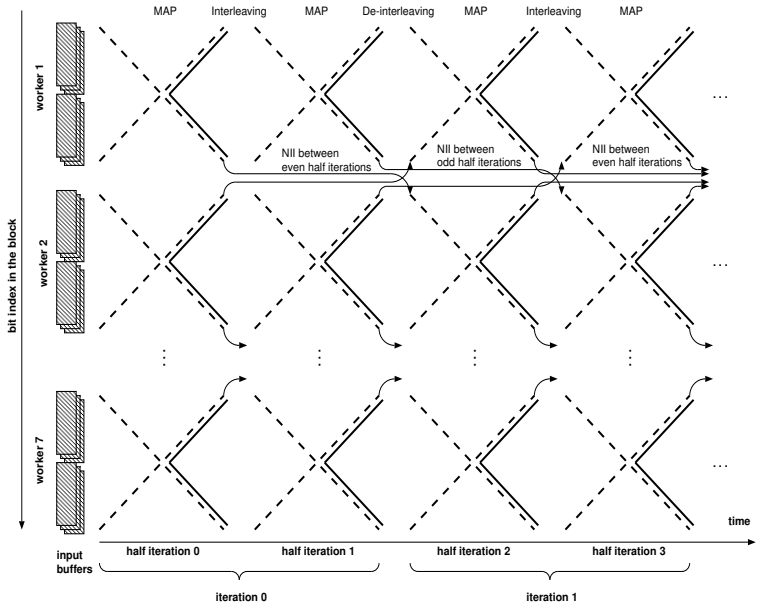


Figure 1.5. Turbo decoding data flow and timing

The remaining training sequences can be avoided by exploiting the iterative aspect of the turbo decoding paradigm. The idea is to initialize the state metrics of each worker at the first recursion step with values deduced from the last recursion step of adjacent workers at a previous half-iteration. This method is called Next Iteration Initialization (NII) and can only be used between half-iterations of same parity due to the interleaving between two half-iterations. Figure 1.5 illustrates graphically the turbo decoding timing and data flow with NII. One cross corresponds to one worker. The x-axis represents the time, and the y-axis represents the frame index.

4. Storage Cycle Budget Distribution

Since in a data-dominated application like the Turbo decoder, the memory organization is a bottleneck for both energy and performance, the memory architecture is optimized before doing the detailed scheduling and synthesis. To this end, the storage cycle budget distribution steps makes an exploration of the memory bandwidth, i.e. the number of memories or memory ports accessible in parallel. The goal is to find a minimum-cost memory organization for different points with different performance, leading to a Pareto plot as shown in Figure 1.2.

Several memory organization issues are explored in the storage cycle budget distribution step, which are systematically approached in four sub-steps. First,

value stored		layer	depth×width
y_{1s}	systematic information for D_1	2	$2NM\times4$
y_{2s}	systematic information for D_2	2	$2NM\times4$
y_{1p}	coded information for D_1	2	$2NM\times4$
y_{2p}	coded information for D_2	2	$2NM\times4$
i	a priori/extrinsics	2	$2NM\times6$
$\bar{\alpha}$	state metrics	2	$7\times2NM\times7$
$\bar{\beta}$	state metrics	2	$7\times2NM\times7$
γ	branch metrics	1	$6\times M\times6$
α	state metrics (copy)	1	$7\times M\times7$
β	state metrics (copy)	1	$7\times M\times7$
Π	interleaver	ROM	$2NM\times9$

Table 1.1. Data structures, sizes and memory hierarchy layer assignment. N is the window size, M is the number of workers, $2NM$ is the size of one frame which is iteratively decoded.

the background data storage is distributed over a memory hierarchy, so that often-accessed data can be stored in small memories close to the data processing units. This is called memory hierarchy layer assignment and is the topic of Section 1.4.1. Second, data structures are organized to be efficiently accessed in parallel when needed, as discussed in Section 1.4.2. Third, loop transformations (Section 1.4.3) create possibilities for parallelization across the scope of loops. Finally, Section 1.4.4 introduces the storage bandwidth optimization proper, which makes a detailed exploration of parallelism of memory accesses within loop bodies.

4.1 Memory hierarchy layer assignment

This is the first sub-step in storage cycle budget distribution, which decides for each data structure, the layer in the memory hierarchy it will be assigned to. The subsequent memory allocation and assignment step determines the optimal memory architecture for each layer. In effect, this sub-step fully determines the static caching decisions.

To benefit from the available temporal locality in the data accesses, a memory hierarchy has been defined [12]. Table 1.1 summarizes the memory hierarchy layer assignment decisions we made for the turbo decoder. The farthest memory layer (layer 2) consists of background memories (SRAM) storing the received systematic (y_s) and coded information (y_p), the a priori/extrinsic information (e) and the state metrics ($\bar{\alpha}$, $\bar{\beta}$). The closest memory layer (layer 1) contains the branch metrics (γ). The state metrics are also buffered in this layer. Energy consumption is reduced because the bigger memories of layer 2 are only accessed once per recursion step to load or store layer 2 values while

value stored		layer	depth×width
y_{1s}	systematic information for D_1	2	$2N\times4$
y_{2s}	systematic information for D_2	2	$2N\times4$
y_{1p}	coded information for D_1	2	$2N\times4$
y_{2p}	coded information for D_2	2	$2N\times4$
i	a priori/extrinsics	2	$2N\times6$
$\bar{\alpha}\bar{\beta}$	state metrics	2	$2N\times98$
γ	branch metrics	1	2×18
$\alpha\beta$	state metrics (copy)	1	1×98
Π	interleaver	ROM	$2N\times9$

Table 1.2. Data structures, sizes and memory hierarchy layer assignment after data restructuring. $2N$ is the size of one worker. Each of these data structures exists M times, i.e. once for each worker.

the first layer, which is smaller and less energy consuming, is accessed more intensively by the data paths.

4.2 Data restructuring

Data restructuring refers to merging and/or splitting of data structures in order to make more efficient parallel accesses to the data structures. Merging of data structures is done when elements in two different data structures are always read and written together. Splitting of a data structure is done if two separate parts of the data structure are often accessed in parallel.

Theoretically, each worker has to be able to compute and store 2^{K-1} state metric values. With $K = 4$, this implies that 8 α and 8 β metrics have to be stored (for each worker). To reduce the cost (energy and area) of the corresponding functional units, special attention has been paid to the quantization of the state metrics ($\bar{\alpha}$ and $\bar{\beta}$). Since all metrics are invariant if a constant is added, they can be *normalized* by subtracting the state-0 metrics. By doing that, state-0 metrics are always equal to zero after normalization and therefore do not have to be stored. Moreover, such a normalization reduces the dynamic range of the metrics and, by consequence, the number of bits required to represent them. Quantization of the state metrics to 7 bits only marginally ($< 0.1dB$) reduces the decoding performance compared to a floating point decoder [9]. Similarly, branch metrics (γ) as well as a priori/extrinsic information (e and i) are quantized to 6 bits. Input log-likelihood ratios (systematic and coded inputs used to compute the branch metrics) are quantized to 4 bits. These quantizations result in the word widths indicated in Table 1.1.

To enable parallel accesses, all state metrics (7 x 7-bit α values and 7 x 7-bit β values) are stored in a very long word (98-bit). Similarly, three (out of six) branch metrics are stored together in a 18-bit word. This merging allows the

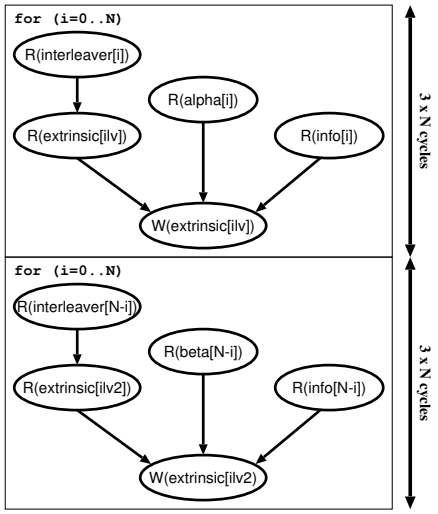


Figure 1.6. Dependencies between memory accesses of two loops

data to be used in parallel without need of a seven-port respectively three-port memory. In addition, splitting is applied on the metrics, extrinsic and input data: the data for each window is stored in its own memory. Table 1.2 shows the resulting sizes for the data structures. Previous investigation [12] has shown that the introduction of this very distributed memory architecture together with the quantization and normalization rules and the memory hierarchy described above lead to an energy reduction by a factor 14 compared to a straightforward implementation.

4.3 Loop transformations for parallelization

The storage bandwidth optimization tool parallelizes the memory accesses within a loop body. When the available parallelism is limited, it can be improved by looking across the scope of loops for parallelization opportunities. This is achieved by transformations of the loops: loop merging, loop pipelining and partial loop unrolling.

4.3.1 Loop merging. The turbo decoder originally consists of a sequence of four loops: D_1, Π, D_2 and Π^{-1}. Due to the dataflow transformations (see Section 1.3.1), the interleaver loops (Π and Π^{-1}) are merged with D_2. Each SISO module (D_1 and D_2) still comprises a sequence of three loops: alpha calculation, beta calculation and extrinsic calculation. To improve parallelism, the alpha and beta calculation are merged together, as shown in Figures 1.6 and 1.7. By merging the two loops, accesses from the two bodies can

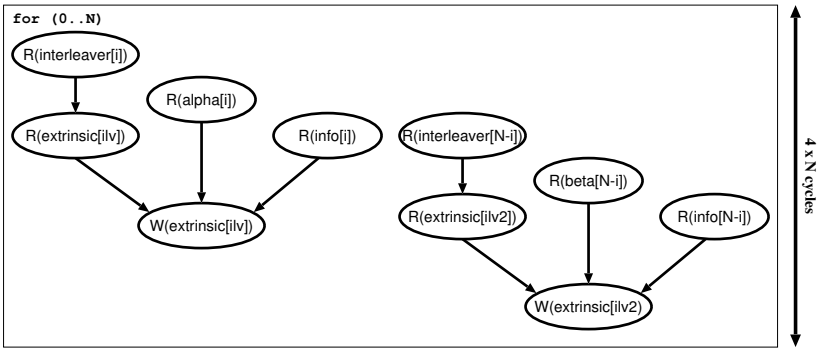


Figure 1.7. Dependencies after merging the two loops of Figure 1.6

be placed in parallel (since there are no dependencies between them). Since our memories have maximum one read and one write port, the merged loop body can be scheduled in four cycles, for a total of $4 \times N$ cycles. The original loop bodies required 3 cycles each, for a total of $6 \times N$ cycles.

4.3.2 Loop pipelining. Due to dependencies between memory accesses the available parallelism in the merged loops is still limited. For instance, in Figure 1.7, one cycle of the loop body is reserved exclusively for the write to extrinsic. To create more opportunities for parallelism, we pipeline the loop. Figure 1.8 shows the result of this. The dependencies between two accesses within one execution of the loop body have changed to dependencies between two different iterations. For instance, alpha is read in one iteration and the result of the extrinsic calculation is written to extrinsic in the next iteration. This has been indicated by dashed arrows in Figure 1.8. The result of loop pipelining is that the total execution time is then reduced to $2 \times N + 2$ cycles. This transformation is applied to the inner loops of D_1 and D_2.

4.3.3 Partial loop unrolling. Finally, partial unrolling (combined with the data structure splitting mentioned in Section 1.4.2) allows different iterations of a loop to be executed in parallel. In the turbo decoder, the sliding windows scheme was selected for exactly this purpose. Since there are no (or limited) dependencies between two windows, it is possible to process them in parallel on different processors (with separate memories). This is reflected in the source code by partially unrolling the loop over the windows.

Even after unrolling, parallelizing the different workers is not possible due to the interleaved addressing of the second SISO. Indeed, the pseudo-random characteristic of the interleaving pattern used for the address mapping introduces collisions in memory accesses. Typically, several workers would write in the same memory at the same time. Restrictions to the permutation pat-

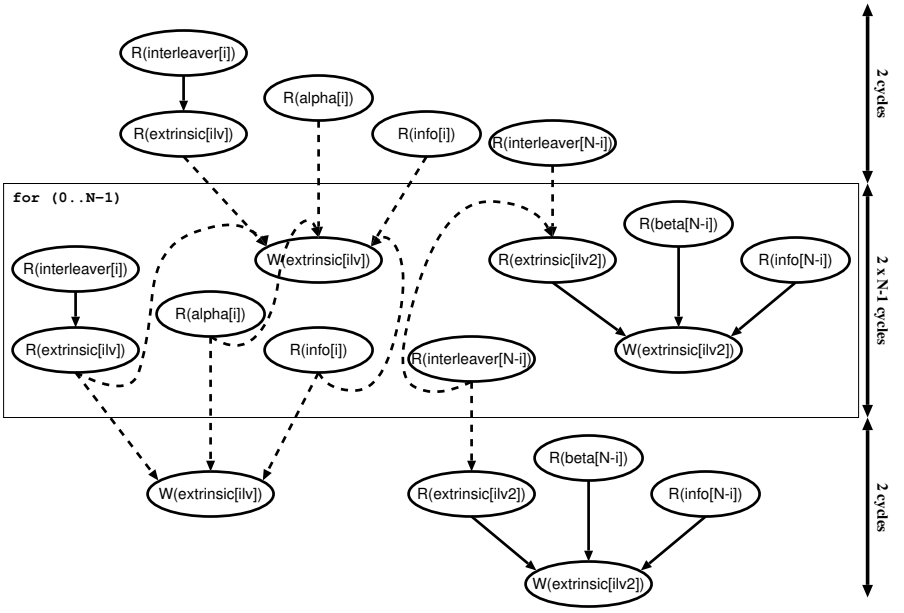


Figure 1.8. Dependencies after pipelining the merged loop of Figure 1.7

tern can be made in order to arrive at a collision free behavior. However, such regularity creates the risk of loosing much of the code's quality. Therefore, a method has been developed to generate collision-free interleavers in a systematic way, showing at the same time excellent coding gain [10]. The energy consumption overhead due to the interleaving is mainly due to the address generation. In our implementation, the addresses are generated by look-up tables stored in low-power ROMs.

4.3.4 Loop transformation results. Table 1.3 summarizes the result of the loop transformations on the performance of the turbo decoder. Each transformation significantly improves both the throughput and the latency. For the loop pipelining transformation, it may be surprising that also latency is improved. However, this behavior is to be expected since the *inner* loops were pipelined; if, on the other hand, the decoder iterations loop, which lies outermost, is pipelined, there is no corresponding reduction of the latency.

4.4 Storage bandwidth optimization

The storage bandwidth optimization tool fully automatically minimizes the required storage bandwidth for a given cycle budget by partially ordering the application's memory accesses. The outputs are the conflict graphs, for each

Loop transformation	Throughput [Mbit/s]	Latency [μs]
None	6.4	70.4
Merging	9.7	46.0
Pipelined	15.6	28.8
Unrolled	115.8	3.87

Table 1.3. Effect of parallelizing loop transformations on maximally achievable throughput and latency

cycle budget, that contain information about which data structures are accessed simultaneously and therefore have to be assigned to different single port memories or a multi-port memory.

5. Memory organization

The storage bandwidth optimization tool is used in conjunction with a tool which determines the memory architecture and the assignment of data structures to memories. This tool provides accurate energy and area estimates for each input memory architecture and for each cycle budget, from fully sequential to the critical path schedule. These estimates, in our case only for energy, plotted against the throughput (which is derived from available the cycle budget) gives the Pareto curve, which the designer can use to decide on the memory architecture.

In the results that follow, we have used the energy consumption values of a memory module generator. The memory modules have one read port and one write port which can be accessed in parallel. For each memory memory used, we have evaluated the access energy by simulating with SPICE the netlist extracted from the corresponding RAM layout in a typical $.18\mu$ technology operating in typical conditions (voltage $1.8V$, junctions temperature 27^oC) and assuming a constant wire load of $.20pF$.

5.1 Memory organization exploration

We first explore the possibilities for an architecture limited to two memories per worker, to avoid overly complex layout. At low bit rates, the optimal memory organization is to store the $\alpha-\beta$ metrics in one memory (per worker), which is 98 bits wide and contains 32 words and consumes 0.36 nJ per decoded bit. All other data is stored in the other memory, which is 6 bits wide and contains 160 words. Its energy consumption is 0.89 nJ per decoded bit, for a total of 1.25 nJ per decoded bit.

This memory organization, however, limits the bit rate to 27.5 Mbit/s. To increase it, one of the coded inputs must be stored separately from the other arrays, so it can be accessed in parallel to other array accesses. It can still be

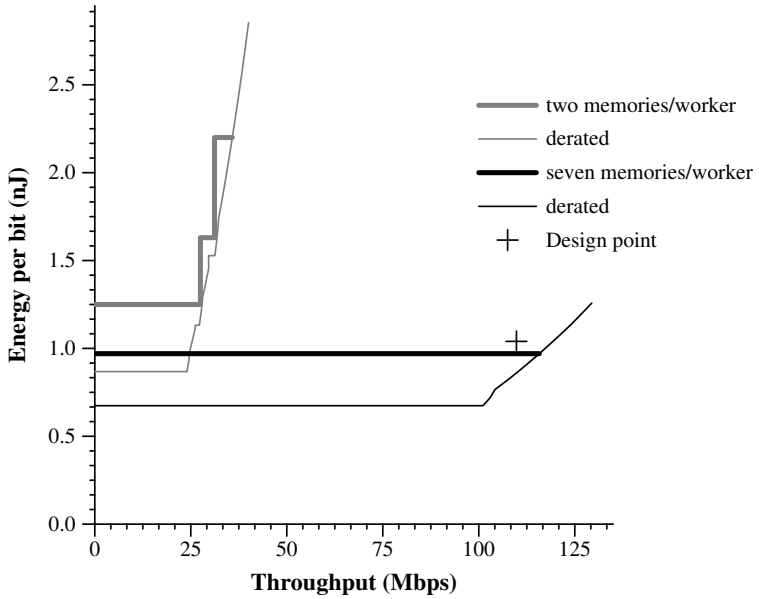


Figure 1.9. Pareto curves for 7 workers, for two and for seven dual-port memories per worker

stored in the same memory as the $\alpha - \beta$ metrics. This memory then contains 64 words (still of 98 bits) and consumes 0.94 nJ per decoded bit. The other memory becomes somewhat smaller, 128 words of 6 bits, and only consumes 0.69 nJ per decoded bit, bringing the total to 1.63 nJ per decoded bit. Thus, a throughput of 31.2 Mbit/s can be achieved. At even higher throughput, also the extrinsic information must be stored separately from the other input arrays, which results in a total memory energy consumption of 2.20 nJ per decoded bit. Figure 1.9 shows these results in a Pareto curve.

In addition to the variation due to the memory organization, the trade-off can be adapted by changing the supply voltage. Scaling it up from its nominal 1.8 V increases the throughput but also the energy consumption, and vice versa for scaling down. Figure 1.9 shows this derating trade-off with a thin line. Note that it is not possible to reliably decrease the voltage under 1.5 V, therefore the curve flattens when this voltage is reached. Likewise, the upper bound for the voltage is 2.05 V.

The trade-off points to be considered by the designer depends on the performance constraint that is being addressed. An increase in performance, as shown, also increases energy consumption. However, the available range of possibilities make a good trade-off feasible. When area is at a premium, though not in our context, a further energy-area trade-off becomes necessary.

The other extreme of a very distributed memory architecture is also shown in Figure 1.9. With seven memories per worker, the additional memory band-

width increases the maximal achievable throughput to 115.8 Mbit/s. This memory organization also consumes less energy than the two memories per worker, since each memory is much smaller. Note that the energy dissipated in interconnect is assumed constant here.

5.2 Memory organization decision

For the turbo decoder design we chose to maximize the throughput and to lower the energy consumption by selecting the more distributed memory organization. An architecture has been defined and implemented previously [9]. This section highlights the major characteristics of the proposed architecture.

All state metrics are updated in parallel by a combinational data path implementing the Equations 1.1 and 1.2. Output values are clamped if needed to fit with the 7-bit representation. The extrinsic information is computed from the state- and branch metrics according to Equation 1.3 by a 3-level pipelined data path, they are also clamped if needed. Such a structure allows to do one recursion step in one cycle. To set up and break down the processor pipeline, an additional 4 cycles per MAP decoding (D_1 or D_2) are needed. The critical path, which is located in the state metrics calculation unit in our implementation in .18μ CMOS technology, has been reduced to 5 ns. Consequently, the total delay to do one MAP decoding (half-iteration) is $(2 \times N + 4) \times 5ns$. With $N = 32$ and a clock frequency of 200 MHz, the data path throughput of one worker is 188.2 Mbit/s.

The proposed architecture achieves constant decoding latency. The throughput is determined by the number of workers M. We do not pipeline the iterations over the SISO modules, so that the turbo-decoder throughput is given by:

$$throughput[\text{Mbit/s}] = \frac{2 \times M \times N}{2 \times I} \times \frac{f[\text{MHz}]}{2 \times N + 4} \tag{1.4}$$

where f is the system clock frequency (up to 200 MHz with the considered .18μ technology), $2 \times N + 4$ represents the MAP delay, I the number of full-iterations and $2 \times M \times N$ is actually the frame size. The minimum decoding latency is given by:

$$latency[\text{ns}] = 2 \times I \times (2 \times N + 4) \times \frac{10^3}{f[\text{MHz}]} \tag{1.5}$$

The energy consumption depends on the memory architecture and on the number of accesses to each memory. The selected memory architecture is summarized in the first two columns of Table 1.4. The number of accesses to each memory is derived as follows. Each input log-likelihood ratio is written once for each frame, and three ratios (one systematic and two coded) exist for each of the $2 \times M \times N$ frame indices. In each MAP decoding, one systematic and

memory type	**#occurrences**	**access energy [pJ/access]**	**#reads**	**#writes**
$2N \times 4$	$6 \times M$	12.70	$16 \times I \times M \times N$	$6 \times M \times N$
$2N \times 6$	$2 \times M$	18.75	$8 \times I \times M \times N$	$4 \times I \times M \times N$
$2N \times 98$	M	60.03	$2 \times I \times M \times N$	$2 \times I \times M \times N$

Table 1.4. Memories architecture with simulated access energy and number of accesses per frame

one coded input is read once for the forward and once for the backward recursion, for a total of four reads per half-iteration, while $2 \times I$ half iterations have to be executed to decode one frame.. The extrinsic information for each frame index is written once in each half-iteration, and read twice. The state metrics are written once for each frame index in the first half of each half-iteration, and read once in the second half. Table 1.4 summarizes these access rates per frame. The third column shows the energy per access for each memory, as derived from SPICE simulations. The average of read and write is taken here. The memory energy (assuming $N = 32$) to decode one frame is then given by the following equation.

$$energy[\text{nJ}] = I \times M \times 21.39 + M \times 2.44 \tag{1.6}$$

We have implemented this architecture on silicon [9] with parameters $N = 32$ and $M = 7$. The corresponding frame size is equal to $2 \times N \times M = 448$ bits. I is programmable up to 6 or can be adapted to the required coding quality with an early stop criterion. Assuming the average case with early stop criterion ($I = 3$), by applying formula 1.6, the decoding energy per decoded bit for this implementation is estimated at 1.04 nJ/bit. Assuming a clock frequency of 200 MHz, which is allowed by our critical path delay, the worst-case ($I = 6$) performance with those parameters according to Equations 1.4 and 1.5 will be a throughput of 109.8 Mbit/s and a latency of 4.08 μs.

For comparison with the memory architecture exploration, the throughput-energy trade-off of the actual design has been plotted in Figure 1.9. Two reasons exist why this point lies slightly off the curve. First, the throughput of the actual design includes the 4 cycles per half-iteration (corresponding to 5%) overhead needed for the data processing (setting up of MAP decoder pipeline), while the exploration only takes into account the data transfers. Second, the energy estimate for the exploration is based on a formula derived from fitting to different simulated memory architectures, while energy estimate of the actual design directly derives from SPICE simulations and thus is much more accu-

rate. Still, the actual design point is close enough to the exploration Pareto curve to validate its usefulness in deciding on design parameters.

6. Conclusions

Identifying energy-performance trade-offs is crucial for an optimal implementation of turbo decoder. Since the decoder is highly data dominated, the maximum gain is in finding the energy optimal memory architecture which meets the performance requirements. We have identified the available trade-offs of an implementation of MAP decoding algorithm, using a systematic methodology. Tool support made possible an almost impossible task of manually checking the entire range of trade-offs for different combinations of memory architectures. Code transformations have been crucial to allow the SBO tool to provide an enhanced range of cycle budget distributions and hence, exposing the otherwise hidden trade-offs.

The methodology explained makes the trade-off points in the turbo decoder visible to the designer, very early in the design cycle, allowing better design exploration. This coupled with further optimization of memory accesses is an important step towards the possibility of using turbo coding schemes in a whole range of wireless communication applications.

References

[1] L. R. Bahl, J. Cocke, F. Jelinek, J. Raviv. *Optimal Decoding of Linear Codes for Minimising Symbol Error Rate*. IEEE Transactions on Information Theory, Vol. **20**, pp. 284–287. 1974.

[2] C. Berrou, A. Glavieux. *Near Optimum Error Correcting, Coding and Decoding: Turbo Codes*. IEEE Transactions on Communications, Vol. **44**, No. 10, pp. 1261–1271. 1996.

[3] E. Brockmeyer, A. Vandecappelle, S. Wuytack, F. Catthoor. *Low Power Storage Cycle Budget Distribution Tool Support for Hierarchical Graphs*. 13th International Symposium on System Synthesis, pp. 20–22, Madrid, Spain. 2000.

[4] E. Brockmeyer, A. Vandecappelle, F. Catthoor. *Systematic Cycle Budget versus System Power Trade-off: A New Perspective on System Exploration of Real-time Data-dominated Applications*. International Symposium on Low Power Electronics and Design, pp. 137–142, Rapallo, Italy. 2000.

[5] F. Catthoor, S. Wuytack, E. de Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle. *Custom Memory Management Methodology — Exploration of Memory Organisation for Embedded Multimedia System Design*. ISBN 0-7923-8288-9, Kluwer Academic Publishers, Boston. 1998.

[6] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P-G. Kjeldsberg, T. van Achteren, T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, Boston. 2002.

[7] H. Dawid, H. Meyr. *Real-time Algorithms and VLSI Architectures for Soft Output MAP Convolutional Decoding*. 6th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, Vol. **1**, pp. 193–197. 1995.

[8] F. Gilbert, A. Worm, N. Wehn. *Low Power Implementation of a Turbo-Decoder on Programmable Architectures*. Asia South Pacific Design Automation Conference (ASP-DAC), pp. 400–403, Yokohama, Japan. 2001.

[9] A. Giulietti, B. Bougard, V. Derudder, S. Dupont, J. W. Weijers, L. Van der Perre. *A 80 Mb/s Low-power Scalable Turbo Codec Core*. IEEE Custom Integrated Circuit Conference, pp. 389–392, Orlando, May 2002

[10] A. Giulietti, L. Van der Perre, M. Strum. *Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements*, IEE Electronics Letters, Vol. **38**, No. 5. February 2002.

[11] S. Hong, W. E. Stark et al. *Design and Implementation of a Low Complexity VLSI Turbo-Code Decoder Architecture for Low Energy Mobile Wireless Communications*. Journal of VLSI Signal Processing, Vol. **24**, No. 1, pp. 43–57. 2000.

[12] F. Maessen, A. Giuletti, B. Bougard, L. Van der Perre, F. Catthoor, M. Engels. *Memory Power Reduction for the High-Speed Implementation of Turbo Codes*. IEEE Workshop on Signal Processing Systems (SIPS) Design and Implementation, pp. 16–24, Antwerp, Belgium. September 2001.

[13] G. Masera, G. Piccinini, M. R. Roch, M. Zamboni. *VLSI Architectures for Turbo Codes*. IEEE Transactions on VLSI Systems, Vol. **7**. No. 3, pp. 369–379. 1999.

[14] P. Robertson, P. Hoeher. *Optimal and sub-optimal Maximum a Posteriori Algorithms Suitable for turbo decoding*, IEEE International Conference on Communications, pp. 1009–1013, 1995

[15] C. Schurgers, F. Catthoor, M. Engels. *Memory Optimization of MAP Turbo Decoder Algorithms*. IEEE Transactions on VLSI Systems, Vol. **9**, No. 2, pp. 305–312. 2001.

[16] C. E. Shannon. *A Mathematical Theory of Commumication*. Bell System Technical Journal, Vol. **27**, pp. 379–423, 623–656. 1948.

[17] Z. Wang, H. Suzuki, K. K. Parhi. *VLSI Implementation Issues of Turbo Decoder Design for Wireless Applications*. IEEE Workshop on Signal Processing Systems: Design and Implementation, Taipei. 1999.