

H263 Decoder

Case study on concurrency analysis

Technical University Eindhoven
Department of Electrical Engineering
Section Information and communication systems (ICS)

Rik Kneepkens (476121)
13 July 2003

Supervisors: ir. Sander Stuijk
 dr. ir. Twan Basten

Summary

For the design of large and fast applications that can demand a lot of computational power, new design path's need to be explored. In the early days that only one processor could do all the desired computation, the designers optimized their designs mainly on throughput. Now that the demand for more computational power has increased for instance for signal processing applications, like speech and video encoders, one tries to design their application for multiprocessor environments. In such an environment, the many different tasks of an application can be split up over the available computational power that every processor possesses.

The design of such a multiprocessor application needs more research., there in the past the main stream of applications where designed as single processor systems. This complex task of designing a multiprocessor application should be as automated as possible, to make the way of designing as easy as possible. In this field some work has already been done. In [1] Sander Stuijk presents five measures that should give an indication whether the designed application is an optimal implementation with respect to concurrency of the application. He also proposes a strategy to optimize these concurrency measures. A case study using a JPEG decoder was used to validate whether the strategy and the contained measures where in fact optimal for this application. In another case study, presented in [6] by Andre Carmo, a 3D recursive search algorithm is used. This study gives more insight in the way to come from a sequential implementation to an implementation that exploits the parallelism that is available when using multiple processors. The work of Sander Stuijk contains only one case study to support the concurrency measures and the proposed design strategy. Therefore another case study needs to be performed.

That case study is presented in this report. A H263 decoder is tried to optimize according to the measures that are presented in [1]. This optimization is tried to accomplish by following the strategy that is proposed in [1]. The design exploration was done in C++ using the CAST environment [4] to produce the desired concurrency measures. CAST is the tool developed by Sander Stuijk that computes the concurrency measures that are presented in [1]. To model the multiprocessor environment the application is modeled in YAPI [3]. To intuitively work with the outcomes of CAST, a GUI designed by Jan Ypma was used.

It is found that the proposed strategy could not be fulfilled exactly as proposed, because this strategy assumes ideal conditions. These conditions are not met in this case study. So the design trajectory results in three different designs that are the results of three little design explorations. This resulted in an altered version of the exploration strategy. The three designs that are the result of the exploration strategy, inhibits different behavior with respect to concurrency.

1.INTRODUCTION.....	2
1.1INTRODUCTION.....	2
1.2RELATED WORK.....	2
1.3GOAL OF THIS REPORT.....	3
1.4STRUCTURE OF THE REPORT.....	3
2.THEORY.....	4
2.1COMPUTATIONAL NETWORKS.....	4
2.2CONCURRENCY MEASURES.....	4
2.2.1.Computation load.....	4
2.2.2.Execution load.....	4
2.2.3.Restart.....	4
2.2.4.Synchronization.....	5
2.2.5.Structure.....	5
2.3STRATEGY.....	5
2.3.1.Task splitting.....	5
2.3.2.Data splitting.....	6
2.3.3.Communication granularity.....	6
2.3.4.Data merging.....	6
2.3.5.Task merging.....	7
2.4H263 DECODER.....	8
2.4.1.Frame construction.....	8
2.4.2.Functional description.....	9
3.CASE STUDY: DESIGN TRAJECTORY ON H263 DECODER.....	11
3.1EXPLORATION.....	11
3.1.1.Starting point.....	11
3.1.2.Data splitting.....	14
3.1.3.Task splitting.....	15
3.1.4.Communication granularity.....	17
3.1.5.Task merging.....	18
3.1.6.Data splitting.....	20
3.1.7.Data merging.....	21
3.1.8.Task merging.....	22
3.1.9.Result.....	23
3.2PROBLEMS.....	26
4.CONCLUSION.....	28
4.1RELEVANCE OF THE CONCURRENCY MEASURES.....	28
4.2EXPLORATION STRATEGY.....	29
4.3THE GUI.....	29
4.4FUTURE WORK.....	30
BIBLIOGRAPHY.....	31
APPENDIX A.....	32
FLOW CHART OF ALTERED EXPLORATION STRATEGY.....	32
APPENDIX B.....	33
GLOBAL RESULTS OF DESIGN 16 TO 31.....	33
APPENDIX C.....	35
NETWORK GRAPHS GRAPHICS.....	35

1. Introduction

1.1 Introduction

Nowadays, applications tend to get larger and more complicated. For some areas the applications even needs to do all the computation in real time, e.g. video processing, sound processing, etc. In order to answer this demand for more computational power, one could just increase the computational speed of the one processor which executes the task or application. But then you'll always be constrained by the highest processing speed of the current generation of processors. Another option to answer the demand for more computational power could be the specialization of the processor for a given task or application. This is already done in a lot of areas, for instance in the signal processing area, where different DSP's are already available in enormous varieties. A third solution for more computational power is the use of more processors. Let every processor do a part of the job, assigned by a (run-time or compile time) scheduler, which in total implements the desired application. In this way, the processor speed doesn't need to be as high as with the first solution, but the computational power can be spread over all the available processors with a maximum speed lower than a single processor implementation. This advantage of lower maximum processor speed comes with the price of having communication between the different processors themselves and between the scheduler and the processors.

In order to realize such multi processor environment, a different way of designing applications needs to be explored. In the days that the main stream applications were all designed for execution on one processor, the design methods hardly exploited parallelism of an application. For the design of multiprocessor applications it is highly relevant to find the parallelism of computations in order to spread the computations over the available computational power. This new way of realizing multi processor environments is a complex task. This task consists of mapping the specification of the desired program to the eventual hardware architecture. Figure 1.1.1 shows the steps in this mapping traject.

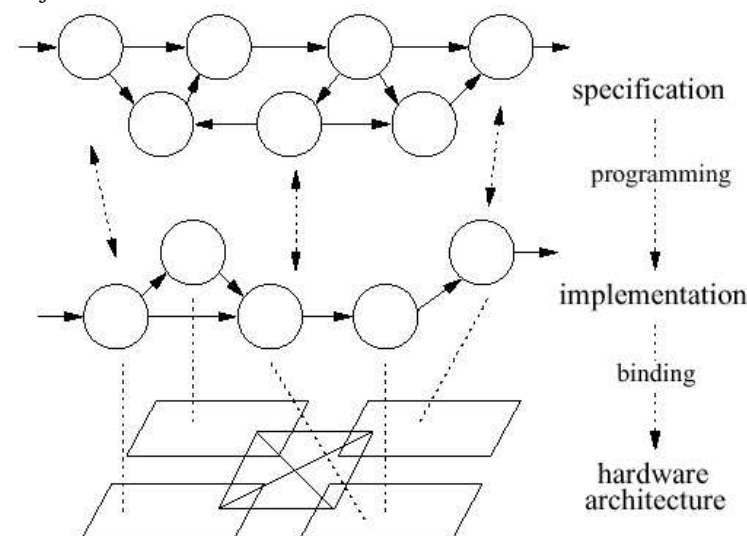


Figure 1.1.1 Mapping traject

1.2 Related work

In order to realize the complex task of mapping the design specification to the hardware architecture, work has already been done. In [1], Sander Stuijk introduces concurrency measures that should give a good indication whether a design is optimal with respect to concurrency. These measures operate on the higher level of extraction, between the specification level and the implementation level from Figure 1.1.1. This work also describes a design exploration that should lead from a first implementation to an optimal implementation with respect to their present concurrency measures. Also a JPEG case study is

presented that tries to optimize the presented concurrency measures using the proposed design exploration. To compute the measures, the system-level software tool CAST is used. This software uses a network specification in YAPI [3]. YAPI models applications as networks using the programming language C++. CAST computes the concurrency measures as presented in [1]. A graphical user interface (GUI) has also been developed by Jan Ypma [4]. This GUI presents the concurrency measures in a intuitive way.

Besides the JPEG case study, also a study was done implementing a 3D recursive search algorithm by Andre Carmo [6]. This work also describes some steps that can be taken in order to get from code designed for serial implementation to code that exploits parallelism.

1.3 Goal of this report

The case studies performed in [1] and [6] are too few cases to state that the presented concurrency measures and the proposed design strategy do what they claim. That is, give a good indication whether the concurrency measures are the measures that capture the concurrency aspects of a design. And that the design strategy truly optimizes these concurrency measures.

This report presents a case study on a H263 decoder. This case study tests the proposed design strategy and to get more insight in the use of the concurrency measures. A H263 decoder has been chosen because of its general form as streaming application. Many parts of this H263 decoder are basic building blocks in present and future streaming applications. These kind of applications are considered to benefit substantial from the exploitation of parallelism.

The graphical user interface, designed by Jan Ypma, is also tested during the trainee ship that led to this report. The testing of the GUI was done to give feedback on the use of the GUI, and to have some debug possibilities.

1.4 Structure of the report

Chapter 2, Theory, shall give the necessary background in order to understand the steps taken later in the design process. In section 2.1 a brief description of a computational network, which implements the specified application, is given. The concurrency measures are outlined in short in the next section, section 2.2. Next in chapter 2, section 2.3, the proposed strategy of the performed design exploration is explained. The decoder under study is discussed in 2.4.

The case study, performed on the H263 decoder is presented in 3. In the first section, section 3.1, the followed trajectory is described that leads to the final designs. The exploration didn't go smoothly. The main problems encountered during the exploration are mentioned in the last section of chapter 3, section 3.2.

Chapter 4 draws up the conclusions that can be drawn from the case study with respect to the goals that are stated in the upcoming chapter. Also some future work is proposed in that last section of chapter 4.

2. Theory

This chapter discusses the theory on computational networks very briefly. A more extensive overview is given in [1]. After the short definition of computational networks, the five measures for a computational network are given. These measures are presented in [1]. In the next section of this chapter a design strategy is presented. This strategy should optimize the therefore presented measures. During the case study presented in the next chapter, this design strategy is roughly applied when optimizing a H263 decoder. At the end of this chapter the necessary theory on the H263 decoder is presented to understand steps that are taken during the design exploration that is described in chapter 3.

2.1 Computational networks

Informally, a computational network consists of some compute nodes that communicate with each other using connections. Every compute node does a part of the job that the complete computational network needs to realize. The nodes receive their data by their connections only. The computational network itself receives the data also by one or more connections to its environment. For the output the same reasoning can be applied. Every compute node will transform the incoming data and give the results back using one or more connections. The computational network also communicates the results by means of a connection.

In this report the terms process and node are used as equivalents. This arose from the fact that every node of a computational network is a process in the YAPI environment [3].

2.2 Concurrency measures

The execution of a computational network can be described as a sequence of events. These events are ordered in time and per node. Based on these events, certain measures can be derived which describe the concurrency of the network under consideration.

In the following five subsections the concurrency measures presented in [1] are briefly explained. In the next section of this chapter a strategy is presented to optimize these measures during the design of a concurrent system. Optimizing these concurrency measures generally means that the concurrency measures should be as close to 1 as possible.

2.2.1. Computation load

The first concurrency measure is the computation load. This measure describes the ratio of the amount of time that a compute node (or computational network) spends on computation with respect to computation and communication. When this measure is 1 (optimal) then the node of network is only doing computation. That is, no time is wasted on communicating with other nodes, networks or with the environment. When the computation load is 0 for a given node or network, then this node or network spent its time solely on communication with other nodes or networks. This shouldn't be the case, because no work is done by the node or network under consideration. Therefore, this measure should be as close as possible to 1 for all nodes in the network and for the network itself.

2.2.2. Execution load

The second concurrency measure concerns the balance of workload over the individual nodes of a network. This measure is named execution load. The execution load is the ration between the time that the node is executing and the total runtime of the system. If the execution load of all nodes is close to one, it means that all nodes have the same amount of work to do. If nodes have an execution load close to 0, it means that they have little work to do compared to those nodes having a higher execution load. Therefore the execution load is desired to be close to one, for all nodes.

2.2.3. Restart

As a third measure for concurrency, the throughput of a node or network is covered in the restart. This measure is dependent on the inverse of the runtime of the node. This runtime is the time that the computation occupies the node, and the time the node has to wait to do the computation [1]. The

inverse of the runtime means that a low restart of one node or network, relative to the restart values of other nodes or networks, indicates that that node has a low throughput. And vice versa. For an optimal design with respect to concurrency, the throughput of the nodes, should be the same. No node or network should wait for another node or network to acquire or transmit data. Therefore all nodes or networks should have more or less the same restart value. The height of these restart values is dependent on the runtime of the system, and is therefore not able to reach 1 in useful designs. It is important however, that the restart is as high as possible.

2.2.4. Synchronization

The fourth measure for concurrency mentioned here is the synchronization. In this measure the speedup of the network is covered in comparison to a sequential system. That means that if the synchronization is smaller than 0, no speedup has been made. The network is even slower than the sequential system. When the synchronization measure is close to one, executing the network takes almost no time in comparison of the time it takes to execute the sequential system. Therefore this measure will be as close as possible to 1 in an optimal design.

2.2.5. Structure

The number of different data streams is taken into account in the last concurrency measure, structure. When this measure is close to one, all nodes in the computational network take part in only few data streams. The parallelism is high in that case. If the structure measure is close to 0, it indicates that most of the data streams are running through all nodes. When this occurs there is little data parallelism. Therefore the structure measure should be as close as possible to 1.

2.3 Strategy

A computational network should perform the transformation of data as fast as possible and preferably faster than a sequential implementation. The measures described in the previous section should therefore be maximized. In [1], a design exploration strategy is proposed, that should enable the designer to come to an optimal design with respect to concurrency.

This strategy will be discussed briefly in this section.

The proposed strategy is as follows:

1. Task splitting
2. Data splitting
3. Communication granularity
4. Data merging
5. Task merging

These last two steps are taken as one step in later versions of the exploration method, as can be found in [4].

In brief, these six action points are discussed below.

The design exploration strategy starts with a fully functional computational network that specifies the desired computation. In the case study, this meant implementation of the H263 decoder in a computational network in YAPI [3].

2.3.1. Task splitting

The first step of the design exploration strategy is task splitting. The purpose of this step is to split the computation in series of tasks. In this way, the task-parallelism gets increased.

In order to do this, the computational network will be split in different computational nodes in a serial manner. See Figure 2.2.1 .



Figure 2.2.1 Task splitting

The criteria for splitting one node into possibly several computation nodes depends on the restart measure. The computation nodes with the lowest restart values, are the ones that need to be split in serial computation nodes.

2.3.2. Data splitting

The second step of the exploration strategy concerns the data parallelism. In this step computational nodes will be split up into parallel computation nodes. Data can be transformed in parallel in this way. In Figure 2.2.2 there is an illustration of this principle.

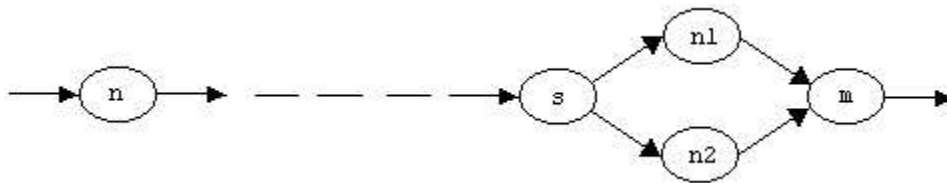


Figure 2.2.2 Data splitting

As can be seen in the figure, the computations inside the compute nodes $n1$ and $n2$ can be performed in parallel. This introduces two extra nodes s and m to split the data stream into two separate streams (s) and to merge the two separate streams into one data stream (m). It might be possible, in a later stage of the design exploration, merge these nodes again with other nodes. The efficiency of the transformation depends therefore on the gain in parallelism introduced by parallelism of $n1$ and $n2$ and the communication overhead introduced by the split- and the merge nodes s and m respectively. The nodes that must be considered for data splitting are all nodes. The focus should be on the nodes with the most data paths going through it. These nodes require the most synchronization. Therefore these nodes can become communication bottlenecks.

2.3.3. Communication granularity

This third step in the exploration strategy should balance the computation load. The goal of the computational network is to compute as fast as possible. The extra execution time that arise from communication have to be as small as possible, because communication is not the goal of the network, but computation is. Communication granularity tries to minimize the communication time of the node and should make sure that the node can do its computation at the desired time, so as few as possible idle periods occur. To influence the communication time, the size of blocks of data that are communicated, can be varied.

Concluding it can be stated that this step balances the computation time with respect to the communication time.

2.3.4. Data merging

From this step, in the computational network, the number of computational nodes should decrease. In this fourth step the computational paths will be merged in such a way that all the paths have the same amount of work to do. In this way, the execution load is increased.

The paths that should be involved in this step are the paths including nodes with low execution loads. These parallel paths should be merged with each other. This step is in general the inverse step of data splitting. But the nodes merged in this step are mostly different nodes than the ones that were split up in the third step.

2.3.5. Task merging

In the last step of the exploration strategy, there will be tried to let every computational node compute as much time of the total execution time as possible. To establish that, serial computation nodes are merged again to achieve a higher execution load. The serial neighboring nodes that have a summed execution load of 1 are candidates of this step of task merging.

2.4H263 Decoder

The performed case study involved the optimization of a H263 decoder. The H263 decoder has been chosen for this case study because it inherits lots of functionality that the modern (video) decoders use. Such as variable length and DCT coding. Video decoders are considered to benefit greatly when more computational power can be used. That is more frames per second can be processed and better quality can be achieved when larger bit rates come available. The general use of the decoding functionality and the expected benefit of exploiting parallelism are the main reasons why this H263 decoder has been chosen.

In this section the basics about the H263 decoder are discussed. A functional description of the decoder is given as well as some explanation about a few characteristics of the decoder. These characteristics are helpful to have some background on the motivations that are given in the next chapter describing the exploration. Detailed information about this decoder can be found in the H263 Standard [5] and the software code made available by Telenor Research [8].

2.4.1. Frame construction

On prominent characteristic of the decoder is the way the output of the decoder build up. This output is graphical presented in Figure 2.2.3.

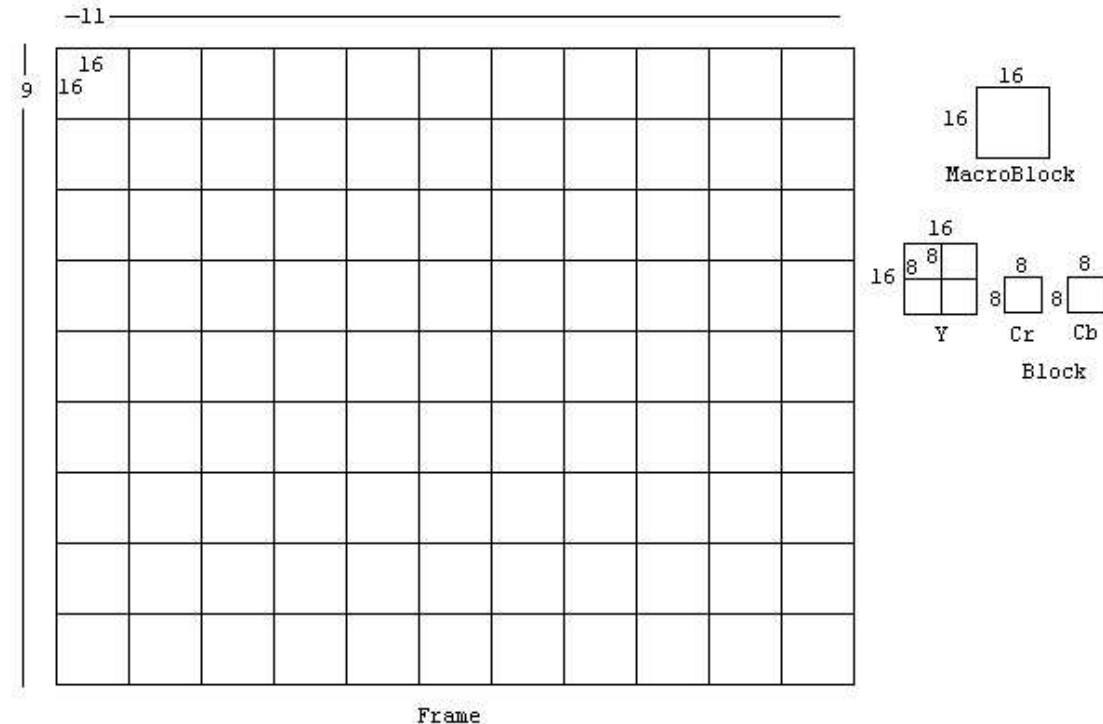


Figure 2.2.3 Frame construction

The output of the decoder is a sequence of frames, with undefined length. Every frame consists of 9 x 11 macroblocks of 16 x 16 pixels. Every macroblock contains information about the luminance (Y) and the chrominance (Cr and Cb) in a 4:1:1 ratio. That means that one macroblock contains 4 blocks of 8 x 8 pixels of luminance data, 1 8 x 8 block of pixels of Cr data and 1 8 x 8 block of pixels of Cb data. The spreading of the 64 (8 x 8) pixels Cr and Cb over the available 256 (16 x 16) pixels of one macroblock is not of importance for this case study. Next a simplified functional description of a H263 decoder is given.

2.4.2. Functional description

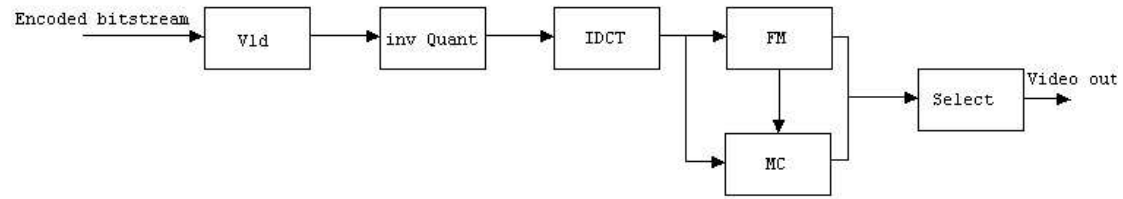


Figure 2.2.4 Simplified H263 decoder

Figure 2.2.4 shows a simplified flow diagram of a H263 decoder. As input the decoder receives a string of data representing the encoded video data. This data is variable length coded. So the first step in the decoder is to extract all the variable length coded data and decode it. The larger part of the data stream then contains the information on the blocks of one frame. These blocks are quantized in the encoder, so the decoder needs to undo this operation in the *inv Quant* block. After that stage, the blocks of the picture contains data that represent DCT coefficients of the data blocks of one frame. In order to get the desired data representing the actual block of one frame, there needs to be an inverse DCT operation. This is done in the *IDCT* block. Next the blocks, representing the decoded information of one block could either be inter coded or intra coded. When a block is intra coded, that block contains all information needed to represent that block of the frame completely. If a block is inter coded, that block contains only an update of the block with respect to the previous (and up coming) frame. So far, all the blocks could just go to a frame memory (*FM*) where the frames could be build up with the decoded blocks. But the H263 standard also describes motion compensation. The motion compensation only gets inter coded blocks, and does the following computation:

When doing forward motion compensation (so called I-frame), the motion compensation uses the previously decoded frame to fetch the moved block for the current frame, and puts an update layer represented by the inter block on top of that.

If also backward motion compensation is needed (so called B-frame) then there first will be a forward motion compensation by fetching the correct block of the previously decoded frame. After that the backward motion compensation fetches the appropriate block of the next frame, which also is under decoding when processing a B-frame. To complete a block in a B-frame, the inter coded block is superposed on the previously fetched block. Figure 2.2.5 shows this procedure in a orderly fashion.

At the end of the functional scheme of the H263 decoder, some selection mechanism should select the correct frame, to maintain the chronological order of the video sequence. This is necessary because an B frame and its next frame are ready at almost the same time.

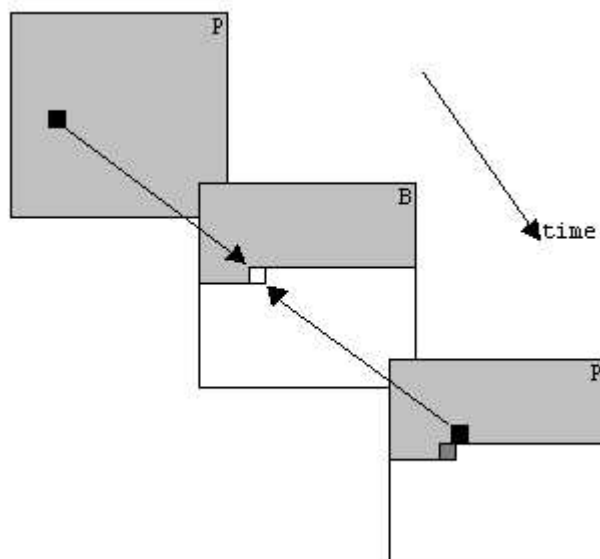


Figure 2.2.5 Backward motion compensation

This short introduction of the H263 decoder should be sufficient enough to understand the actions taken in the design trajectory presented in the next section.

3. Case study: Design trajectory on H263 Decoder

This chapter describes the steps taken which should end up with the best process network implementing the H263 decoder. With the best it's meant that the design is optimal according to the measures presented in the previous chapter. This does not mean, that an optimal design is the optimal design for every designer. This depends on the wishes that the designer has. If only speedup is preferred and inefficiency does not matter, that the synchronization is the main criteria for the design. If on the other hand, efficiency is important, than all measures should be accounted for when looking at the results.

In the next section the exploration is described, whereas in the end of that section the results are briefly stated. The networks created during the case study are pictured in Appendix C as graphs. The design numbering is pure based on chronological ordering. Not all the designs resulted in useful information or didn't even to turn out to be a correct functioning implementation. Therefore the designs mentioned in this report do not fully cover the range of all design numbers 1 to 31. The design numbers not mentioned do not contribute enough to the case study to be mentioned at all.

3.1 Exploration

3.1.1. Starting point

To start the exploration a implementation of the decoder in C++ [7] and in YAPI [3] is necessary. Also some test sequences to test the correct behavior of the decoder are necessary. These test sequences can use different options of the decoder, which can result in variation between results per design. In Table 3.3.1 the test sequences that are created are listed. The parameters they possess are also listed in this table.

Filename	#frames	PB frames (#)	Quant (q)	Remarks
carp20.inp	7	no	10	
carp10.inp	4	no	10	
carp10_15.inp	4	no	15	
carp10_5.inp	4	no	5	
carpPB60.inp	19	yes (1)	10	
carpBP120.inp	39	yes (3)	10	Frame 15, 35, 37 B-frames
carpBP60_15.inp	19	yes (1)	15	
carpBP120_15.inp	39	yes (3)	15	
carpIntra.inp	382	no	10	Contain intra blocks exc. 1st frame
carpMoreIntra.inp	76	no	10	Contain intra blocks exc. 1st frame
salesPB60.inp	21	yes (6)	10	Frame 7, 9, 11, 13, 16, 17 B-frames

Table 3.3.1 Test sequences

The sequences were taken from either the carphone movie file or from the sales movie file. The number of frames per test sequences varies, so large impacts of the intra blocks in the first frame can be localized.

There are five sequences that contain PB-frames. These sequence have been created because PB-frames use extra functionality. The backward motion compensation that is.

The quantization parameter (Quant in table) was easy to vary when the test sequences were created. It was not tried to find out if this should have any influence on the concurrency of the design. However this parameter is only a scale parameter. With that it is meant that for every realistic value of this parameter, only one site of a multiplication varies. Therefore this parameter is not used during the exploration.

These test sequences produced per design various results. It would be logical if these results were statistically combined so more general conclusions can be drawn from the experiments. Keeping in mind the functional differences of the decoder that these parameters require of course. Due to the lack of time and the fact that CAST and the GUI were not able to present these combined figures at the time

of the exploration, this statistically combined figures were not used. There is chosen to present the results using the salesPB60 and carpMoreIntra test sequences. These sequences showed to the rightful directions of the steps that had to be taken. This does not mean that these two sequences are representative for the whole range of test sequences and for the sequences the H263 decoder could receive. When more time was available more representative sequences could have been found.

The design exploration starts with a functional implementation of a process network, implementing the H263 decoder. In order to come to a process network it was advised to get rid of all the pointers and global variables in order to speed up the exploration in later stadium [5]. Getting rid of all pointers in function calls is important. If variables are inly passed by copies between functions (i.e. no pass by reference or through global variables), then we can easily replace the passing of data with the reading data from and writing data to a fifo. During the case study this was hard to achieve in the first part of the design trajectory. This was mainly caused by the absence of good documentation about the source code. To get rid of all the global variables didn't seem to be very helpful. Of course it is easier to handle one function as a complete autonomic process, so extractions of one function into one computation node can be made very easily. But the code used so much global variables that function calls should become impractical long. Therefore the global variables were inventoried to make sure that these didn't slow down the exploration process. This has been proven very valuable for the rest of the design process. Although at every point in time during the design process, parts of the program made more sense.

The way to cope with massive numbers of global variables was to create the global variables which were used in the extracted nodes as well as in the nodes from which the extraction took place in the extracted node. If the global variables of the extracted node depended on the node from which the node was extracted from, than a fifo channel was created to get the desired value of that global variable to the correct node. If the variable could stay updated in the processes by using its own data, then of course, no fifo channel was used to acquire the correct value of a global variable. Another step carried out at the start of the exploration was removal of all unnecessary options of the application. Such as viewing the video sequence via the X-environment and many other options. This removal ended up in a basic H263 decoder, that could switch on or of the use of PB-frames. No other advanced options are supported in the implemented version of the decoder.

To start with the exploration, first a functional split up of the decoder needed to be made. After this stage the design should be a functional split up more or less as section 2.4.2 describes. Of course other split ups can be made. But the functional split up of the decoder is a easy way of getting familiar with the code and the structure of the application. The first step was to create one network process doing the computation completely and two processes to communicate with the environment. The computational network dec is created which has computation nodes FRONTEND, DECODER and BACKEND. FRONT- and BACKEND communicate with the environment to get the input data and to store it the output data, and shall not be used when calculating the concurrence measures. Because the supply of the input data and the storing of the output data is very platform dependent, these processes are not wanted to participate in the concurrency measures. DECODER does al the computational work, in this case the H263 decoding step. This first step of the exploration tries to extract the functionality's described in section 2.4.2, from the DECODER process.

To get more familiar with the code, first the IDCT process was created. Although not the most logical step, it was easy to implement. The IDCT process was implemented as a loop starting and ending at the DECODER. It didn't present much improvement, because the DECODER was still waiting on the data just after it was passed to the IDCT process. The logical step would be to extract a variable length decoder process (VLD) which would tell the DECODER process what to do. All the designs that were created until this point, were used to get rid of options in the application, to clean up the code and to implement a correct functioning network in YAPI. After the VLD process was created in design 16 (See Appendix for network graph), it was CAST that showed that the DECODER was still the bottleneck with respect to the restart.

On functional level again, the IDCT process previously driven by the DECODER, was then driven by the VLD in design 17. The VLD only steered the DECODER in design 16. Design 18 was the result of extracting a STOREFRAME process between DECODER and BACKEND. This process gets the functionality that orders the output data in the correct format. The ordering of the output data should not be used when evaluating the concurrency measures. This STOREFRAME process can therefore not

be found in the functional schema in Figure 2.2.4. You can see the effect of this step on the increased restart of the decoder, which indicates that is necessary to extract this part out of the DECODER. In analog to the functional schema of the decoder, also a DEQUANT process was extracted, but that was from the VLD in design 19. These last three steps increased the restart of the IDCT with roughly a factor 7. But the restart of the DECODER remained low, increasing at most with roughly factor 2,5. This difference in improvement is thought to be caused by the huge amount of computation that the DECODER has to do. This narrowed the influence of little pieces of functionality extracted from the DECODER. The VLD and the DEQUANT had a restart 5 times as large as the one of the DECODER. At this point of the exploration a functional split up is made of the H263 decoder. Figure 2.2.4 shows a forward-, backward motion compensation and a select block, however these functionality's are still incorporated in the DECODER process. There has been chosen to start with the next step of the exploration, because the extraction of these two or three processes are large steps to take. Smaller steps can be taken first if these processes are extracted later on the exploration. The knowledge of the code will be increased at that point in time, so larger steps can be made more easily. Figure 3.3.1 shows that the DECODER is the bottleneck with respect to the restart in the created designs.

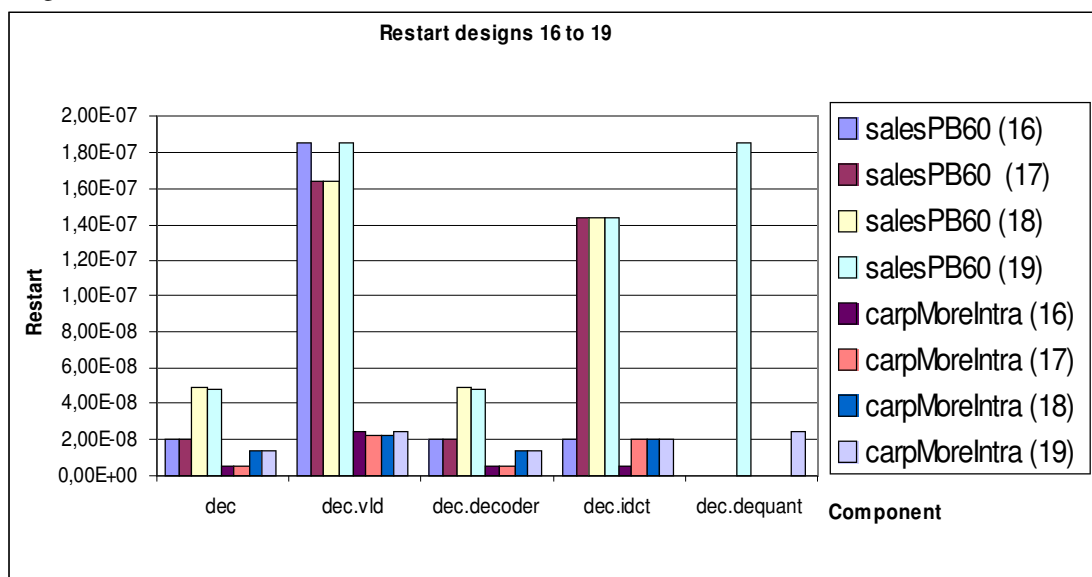


Figure 3.3.1 Restart of designs 16 to 19 per component

The created designs (up to design 19) ended up in a functional split up of the H263 decoder more or less as specified in the functional schema in Figure 2.2.4. These split up basically did the first step of the exploration strategy, which is task splitting. Therefore this step of the proposed exploration strategy is skipped, and in the next step some data parallelism will be explored. The end this first step, the figures of this design, design 19 are shown as well as the network that acts as the starting network.

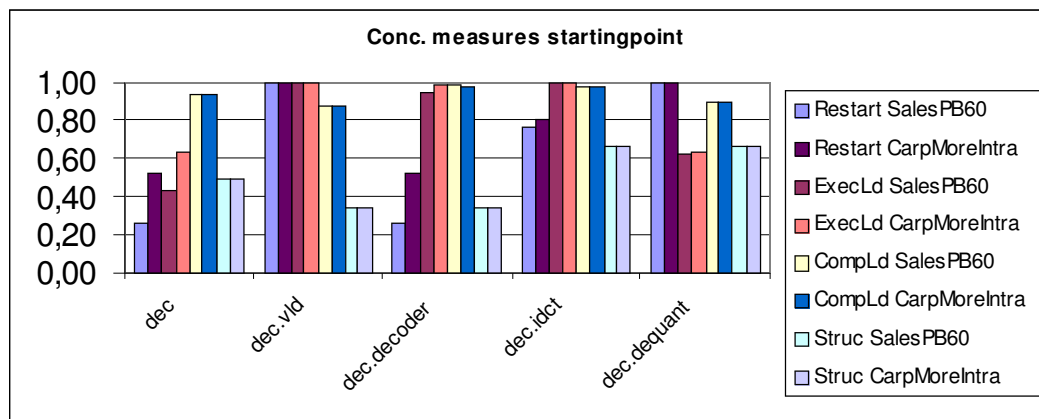


Figure 3.3.2 Concurrency measures at starting point

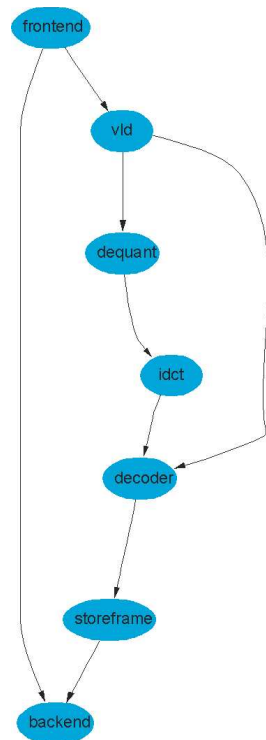


Figure 3.3.3 Design 19, Starting point

The restart measures in Figure 3.3.2 are scaled to the maximum restart measure per design. For the carp sequence this was $2.45e-8$ and for the sales sequence the maximum restart was $1.86e-7$.

3.1.2. Data splitting

The point where data parallelism is available is at the path between the VLD and the DECODER. In design 21 this path has been split into two paths. One for intra blocks and one for inter blocks. This step increased the structure of the design, as could be expected from [1]. This is shown in Figure 3.3.4 and Figure 3.3.5.

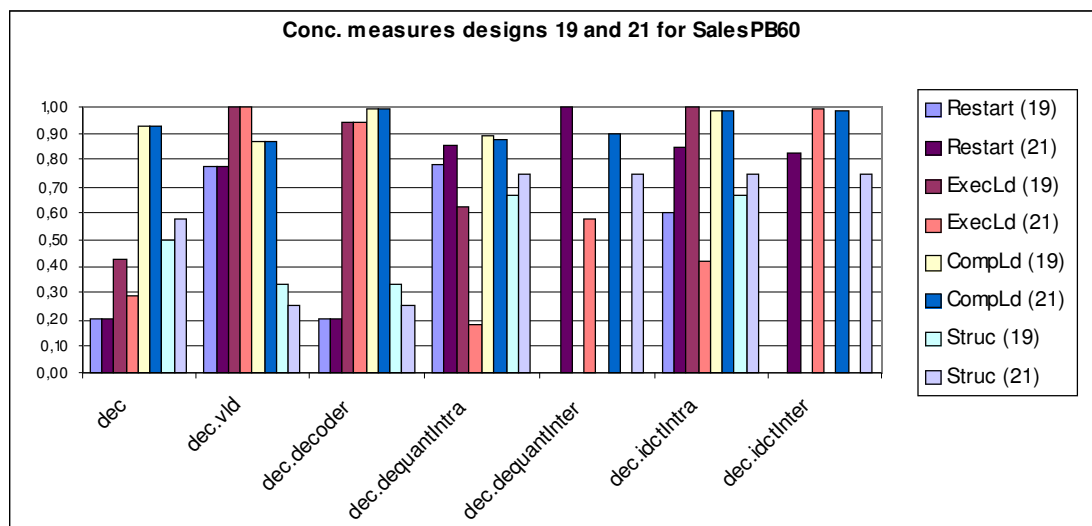


Figure 3.3.4 Concurrency measures designs 19 and 21 for SalesPB60 sequence

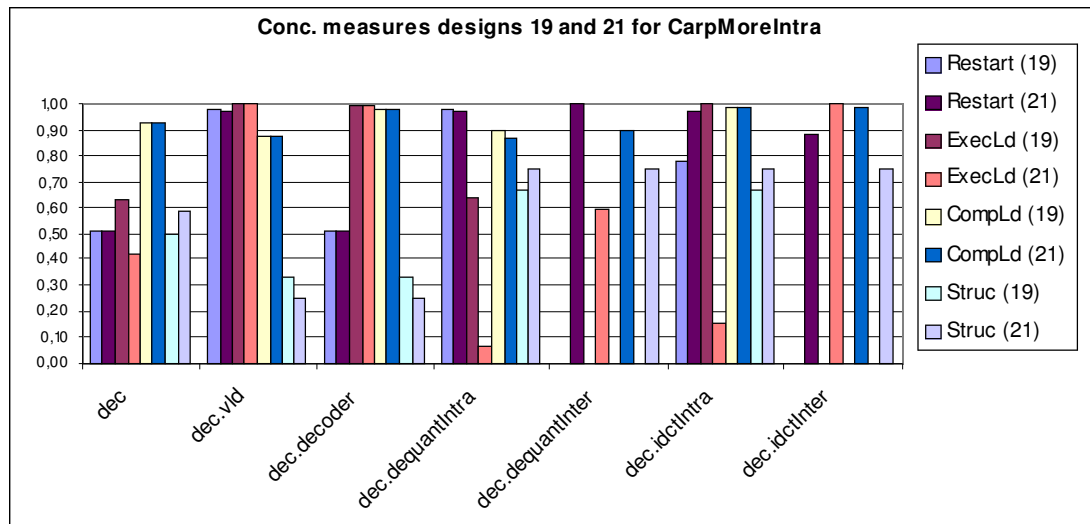


Figure 3.3.5 Concurrency measures designs 19 and 21 for CarpMoreIntra sequence

The execution load of the design decreased in this step. This is due to the idle times in the intra path, which pulled the execution load of the nodes in that path down. The other concurrency measures, restart and computation load stayed the same as expected. The synchronization for the two designs, designs 19 and 21, also stayed the same. That means that the two paths either are faster than the VLD produces the data and the DECODER can use the data, or the amount of data passed through both paths is badly balanced (e.g. one path gets lots of data while other path is mainly idle). The first explanation of very fast paths is true when the restart of the DECODER is considered. Both the two paths as the VLD have a restart almost twice the one that the DECODER has. The second argument also holds when looking at the execution loads of the processes in the two paths. The intra path is hardly used, whereas the inter path is much more used. As can be seen from the execution load of the IDCT process in the inter path. This last fact, that the both paths do not get the same amount of data is caused by the used test sequences. In the used test sequences the first frame consists solely out of intra blocks. In the other frames, only few intra blocks were present. The intra blocks only occur when large motion in the sequences is available, and these sequences show relatively few motion. Therefore the relative number of intra blocks in the sequence drops dramatically when the sequence increases in length. The correct ratio between the number of intra and inter blocks for a representative sequence is hard to choose, and needs more study. This has not been carried out due to the lack of time for performing this case study. The result of this step is therefore difficult to interpret, but it looks like a tradeoff between structure and execution load. In the upcoming designs, the intra and inter path are used. In a later stadium of the exploration these two paths can change separately with respect to the other path.

3.1.3. Task splitting

In the proposed exploration strategy communication granularity should be the next step to perform. The order of the steps in the proposed exploration strategy is the ideal order. But due to the huge code length and therefore the difficulties in understanding the program code, made the performed exploration far from ideal. Because of that, the proposed order of these steps has not been carried out. There is tried however, to follow the coarse lines of the exploration, by first splitting and at the end doing the merging. This subsection then, describes the next step taken, differentiating from the proposed exploration and performing task splitting.

In order to increase the restart of the DECODER and with that the smoothing of the execution loads, the FORWARDMC process is extracted from the DECODER. This results in design 23.

The results are not what hoped for. The restart of the DECODER decreases as well as the execution load. (See Figure 3.3.6 and Figure 3.3.7) This means that the DECODER passes work to the FORWARDMC and waits for the FORWARDMC process to pass the results back. The wish is that the FORWARDMC should pass the results to STOREFRAME, so DECODER can start computing on the

next data when previous results have been send to FORWARDMC. This is not implemented, because there was no clue how to implement this.

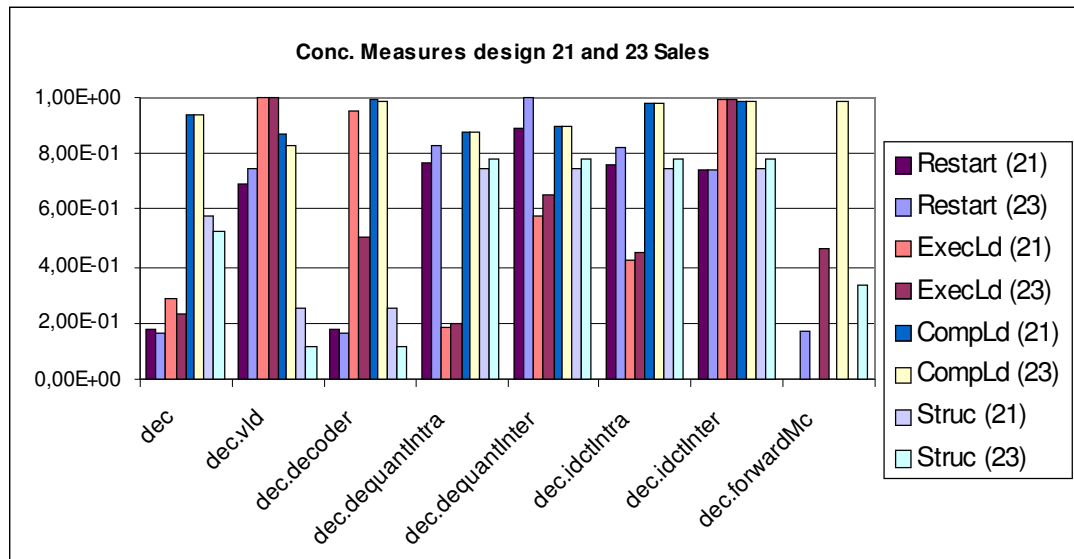


Figure 3.3.6 Concurrency measures for designs 21 and 23 for the Sales sequence

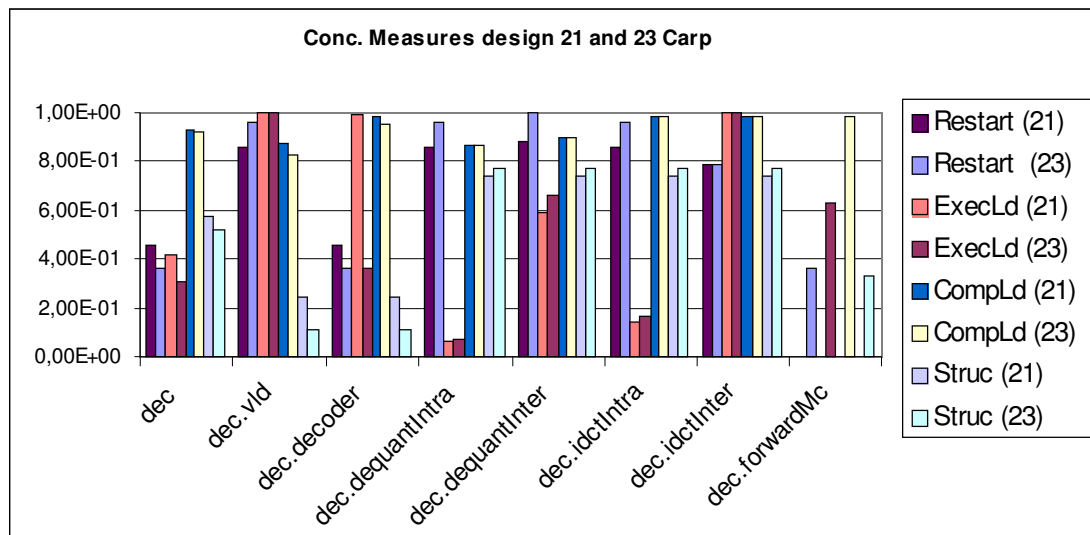


Figure 3.3.7 Concurrency measures dor designs 21 and 23 for the CarpMoreIntra sequence

In the next step it was chosen to extract a BIDIRECTMC only used when PB-frames were available in the test sequence. The choice of extracting this process was based on the fact that the execution load of the total design was rather low. Say 25%. The execution load of a process contributes to the total execution load with the inverse of the restart of that process. Because the restart of the DECODER and the one of the FORWARDMC were still the bottlenecks of the design, the restart was tried to increase by extracting the BIDIRECTMC process. This new BIDIRECTMC process was introduced in design 24. The restart of the DECODER did change again, but also in the wrong direction. It dropped to half the restart of before the extraction of the FORWARDMC process. Also the execution load remains very low for the DECODER, the FORWARDMC and the BIDIRECTMC process. This means that both processes, BIDIRECTMC and FORWARDMC are waiting on the DECODER for data and the DECODER waits until the results of one of these two processes get back. In CAST the event traces showed that this reasoning was correct. The visualization software, that was under development during this trainee ship, also provided the visual indication for this reasoning. This is shown in Figure 3.3.8 for the FORWARDMC process. This same picture can be found when evaluating the BIDIRCTMC process.

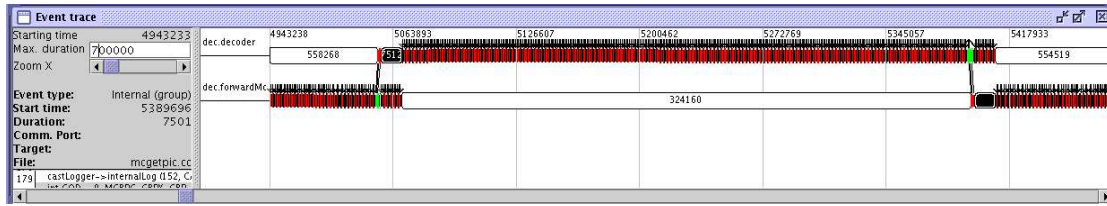


Figure 3.3.8 Event trace of design 24, showing the waiting on the data by FORWARDMC

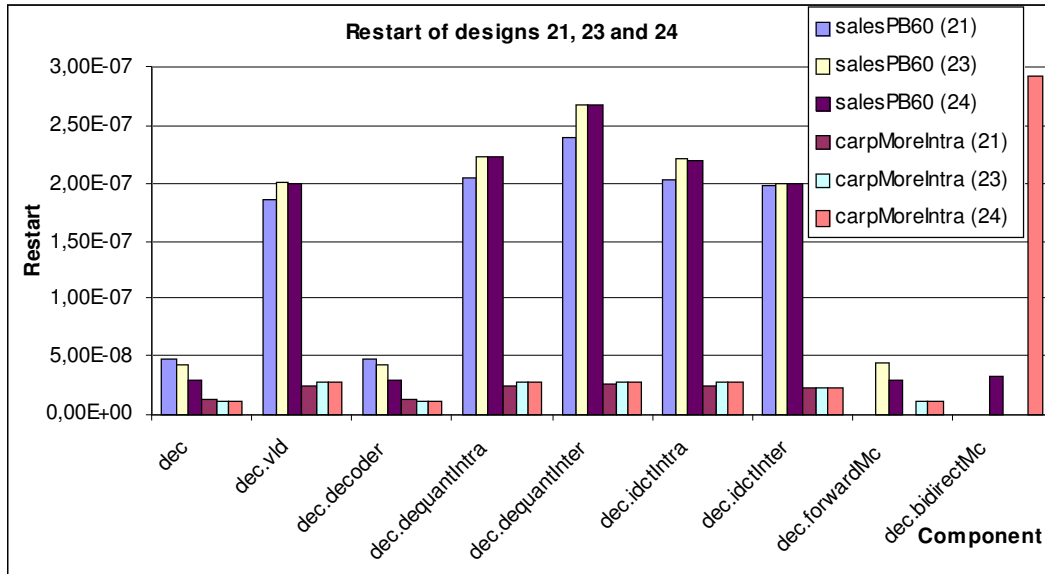


Figure 3.3.9 Restart of designs 21, 23 and 24 per component

3.1.4. Communication granularity

The way to improve the low execution load caused by the idle processes and to boost, is to communicate in smaller parts. In design 25 the communication between the DECODER and the BIDIRECTMC changed, passing smaller amounts of data. This meant that the restart of the complete design slightly increased, as well as the execution load of the complete design.

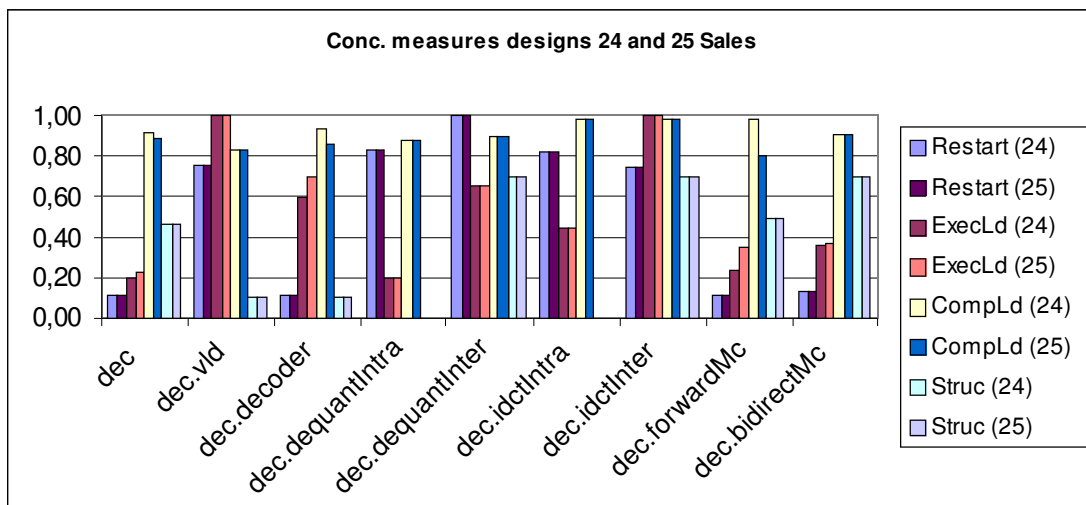


Figure 3.3.10 Concurrency measures designs 24 and 25 for Sales sequence

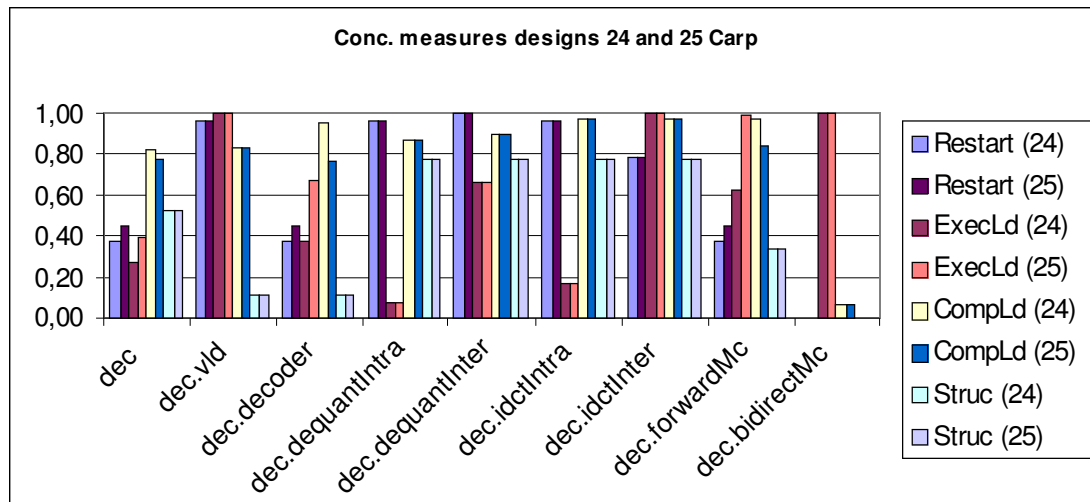


Figure 3.3.11 Concurrency measure designs 24 and 25 for CarpMoreIntraSequence

Figure 3.3.10 and Figure 3.3.11 show the results. Both sequences show improvement with respect to the execution load of the DECODER and the FORWARDMC process.

The computation load, which should be the main lead while performing this step, is not addressed. This is due to two reasons. The first is the interpretation of this communication granularity step. In [1] this step is mainly used for addressing communication costs for every read or write function. In this case study, this step is also interpreted as efficient communication between two process on a more functional level. E.g. communicate in blocks, which can be done at place A in the code, or communicate in frames, which can be done at place B in the code. This naturally influences the communication costs (computation load), but also the execution load and perhaps even the restart, because of the functional difference of the two communication choices. (Communicate from place A or B).

The second reason why the computation load was not looked at, at this point in time, was the absence of knowledge about the correct delay parameters in CAST that control the costs of communication. In the generation of the results, these cost have been taken in account for. That's the reason why the computation load is not one in the results of the designs. The parameters that are chosen for the communication costs, are 30 units for the initialization per read or write call. Every data element that passed the communication line, had a cost of 1 just as most C statements. These parameters assume that the request and initialization to start passing data takes more time, than the actual passing of the data. This can be realistic if a broad data buss is used to let the processes communicate with each other. No steps have to be taken to improve the computation load, looking back, because this measure is quite nice for the generated designs.

3.1.5. Task merging

The fact remained that the DECODER had a low restart value and the IDCT process in the inter path had more to do than most other processes. The low restart value of the DECODER was taken granted at this point of the exploration. There was no way of extracting more functionality out of the decoder. Therefore the execution load of the process with the lowest restart measure should be as high as possible. Otherwise a small execution load of a process with a low restart would pull the total execution load down by a great amount. Of course this effect on the total execution load is what is wished for. That is, if a process has a relative low restart then it is long idle during the total execution of the complete design. The execution load of the total design should therefore be decreased. Even if that process with a low restart has a high execution load, because of the previous mentioned reason. When this process has also a low execution load, then the idle time should of course be visible in the total execution time of the design.

To prevent this low execution load by badly scaled restart, the FORWARDMC was merged back into the decoder again from design 26. This merger also arose from the fact that the DECODER's computation load dropped with 20% when the forward motion compensation was extracted. The execution load of the DECODER increased from 70% to 80-99% from design 25 to design 26 as can be seen in Figure 3.3.12. This increase depended on the test sequence. When PB-frames were present,

a execution load of 80% could be obtained. When no PB-frames were present in the sequence, the execution load increased to almost 1. From this point in the exploration, it was tried to optimize the execution load by getting the value as close to one for all the processes in the design. The first merger that took place, was the merger between the DEQUANT processes. The summation of both execution loads were for every simulation close to 1. The merger of both processes ended up with a DEQUANT process with a execution load between 75% and 80% as can be seen from design 27 and Figure 3.3.12. Looking at the graph representing the decoder at this point, only the IDCT processes were possible candidates for merger. The IDCT process in the inter path had already an execution load of nearly 1. Because of the available test sequences, it was decided not to merge the IDCT processes. When the test sequences were long enough in duration, than the intra path is expected to be obsolete. But that was not the case with the available sequences. So far the execution loads of the complete design was pushed back by the differences in restart measures of the computation nodes in the design. So an other split up had to be made, in order to get more flexibility in smoothing the restart. Therefore an other option of data splitting was examined next. This could be done at the start of the exploration. For instance after the splitting of the intra and inter path. Because of the enormous code length, this split option was delayed until this point.

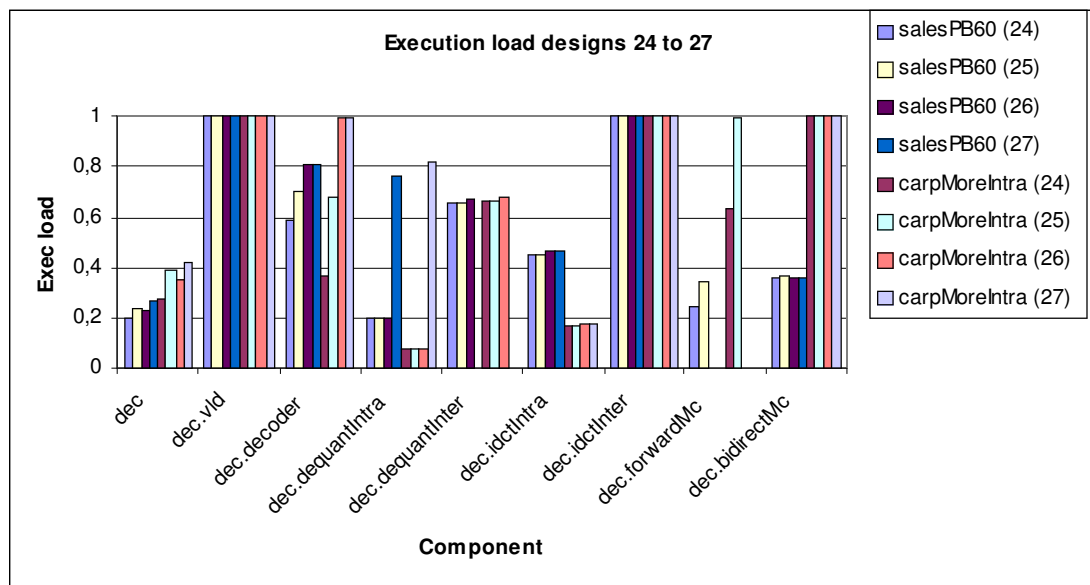


Figure 3.3.12 Execution load for designs 24 to 27

Looking back at the created designs so far, there are two different design paths visible. The first ends at design 21, where there was a different path for intra- and inter blocks created. Then a kind of new start of the exploration started trying to make the DECODER less of a bottleneck by extracting the FORWARDMC and BIDIRECTMC from the DECODER. After communication optimization, it was best to merge the FORWARDMC back into the decoder. Ending up at design 27. This is schematically shown in the figure on the right. (Figure 3.3.13) Every gray bullet is the end of one design path. Next another design path is created starting from design 21.

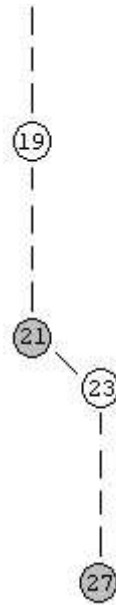


Figure 3.3.13 Mapping trajectory in design numbering

3.1.6. Data splitting.

The path between the dequantizer (DEQUANT) and the store process, was split up in three paths. One for the luminance and two for the chrominance. This means that the DECODER process is now split up into 3 processes. One process per color component. In the resulting design, 28 (Figure 3.3.16), the DECODER for Y and the BIDIRECTMC process for Y were the bottlenecks with respect to the restart value. The difference with the other processes were roughly a factor 2.5, focusing on the design with PB-frames. This factor 2 is strange. One could expect a factor 4, just looking at the specific data every color component needs. This is not the case. It is thought that this lower factor is the result of the many options that the every color component DECODER needs to process. This processing can not be neglected to processing for the actual building of the frame. In the paths of the chrominance the execution loads were lower than on the luminance path. This shows that the chrominance paths should be merged.

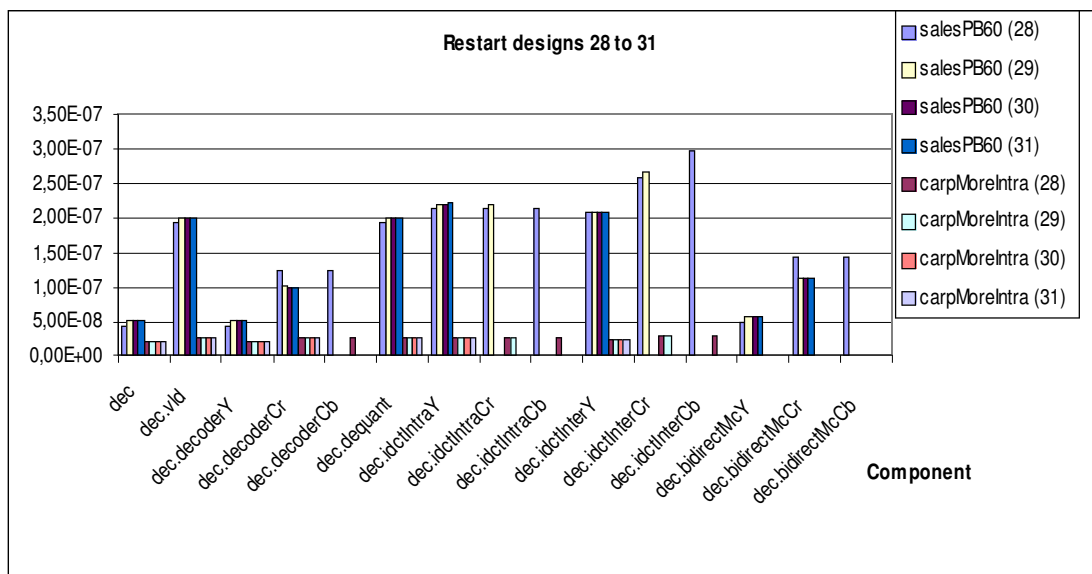


Figure 3.3.14 Restart designs 28 to 31 per component

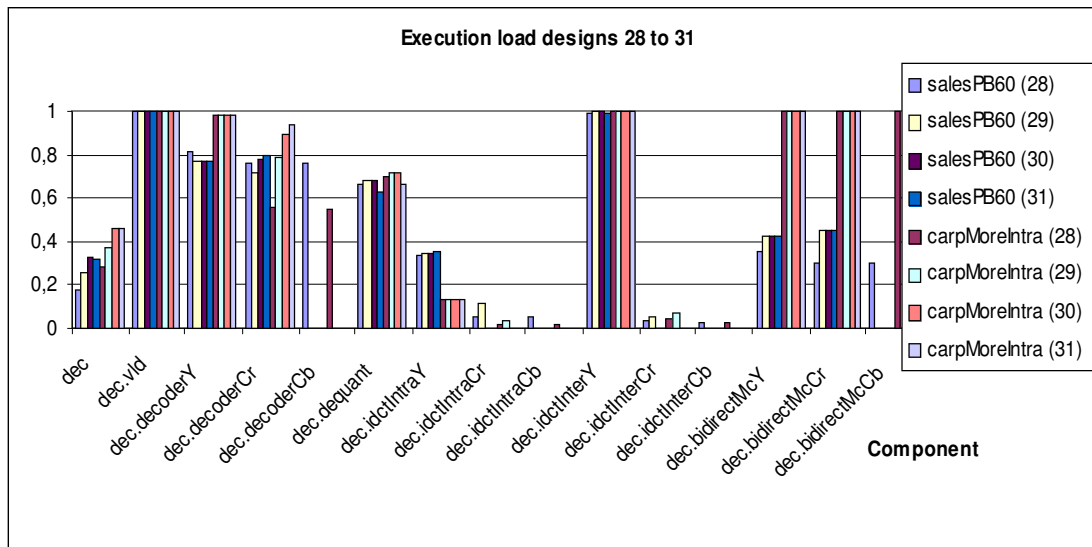


Figure 3.3.15 Execution load Designs 28 to 31 per component

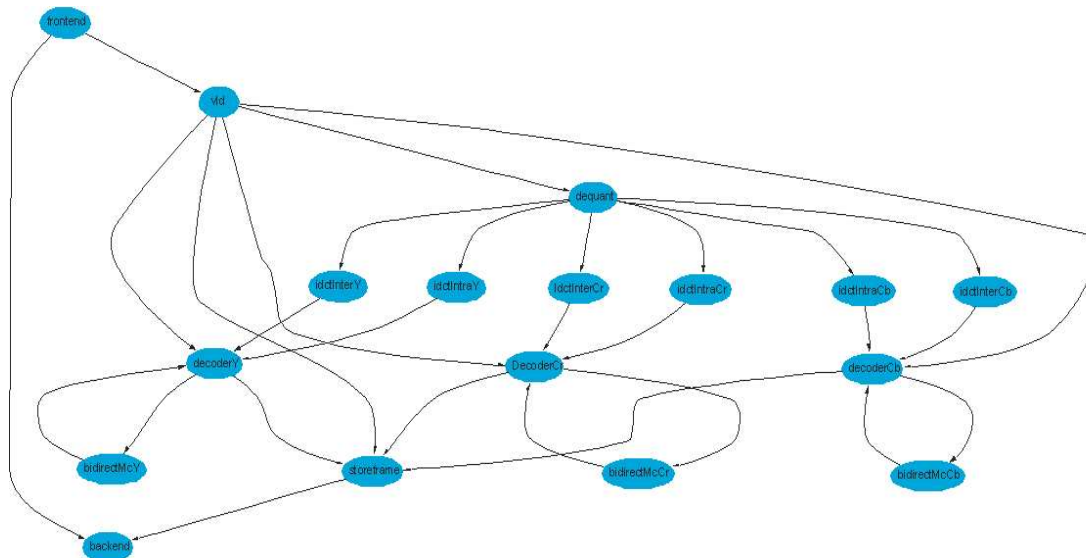


Figure 3.3.16 Design 28

3.1.7. Data merging.

In design 29 the two chrominance paths were merged. As expected, the execution loads of the merged IDCT process was the sum of the IDCT processes separately (See IdctCr component in Figure 3.3.15). Due to the overhead on processing options, the execution load of the resulting DECODER and BIDIRECTMC process in the chrominance path increased with 50%. The IDCT processes of the chrominance path were still the bottlenecks with respect to the execution load. The restart value of the resulting chrominance DECODER was twice as high as the one of the luminance path. This could be expected. Before merging of the two chrominance paths, the ratio of the restart between luminance and one chrominance path was also in the order of 2. This is hard to explain. The most likely explanation is the overhead in options needed to process for both luminance paths. The fact that the ratio stays at 2, indicates that the existence of two separate chrominance paths is obsolete. This ratio of 2 in restart value, indicates that the chrominance path is still too fast with respect to the luminance path. Although the splitting of the IDCT process in the inter path of the luminance could increase speed in that part of the design, the DECODER in the luminance path will always be the bottleneck with respect to the restart. Therefore it is tried to slow down the chrominance path to even the restart measures.

3.1.8. Task merging

To slow the chrominance path down, first the IDCT processes are incorporated into the decoder of the chrominance path in design 30. This doesn't affect the restart of the DECODER in the chrominance much. This can be explained by the higher restart value of the IDCT processes than the one of the DECODER. The execution load benefits substantially by this step. Before the merger of the IDCT processes in the DECODER, the IDCT processes in the chrominance path had, by far, the lowest execution load. This is gone back to the DECODER and BIDIRECTMC process in the chrominance path. Although the work is more balanced, the speed of both paths still differ with maximum a factor of 2. The last possibility to slow down the chrominance path is the merger of the DEQUANT functionality into the DECODER in the luminance path. In design 31 this merger has been implemented. This step doesn't improve the design. The restart of all the nodes hardly changes. The only measure that changes is the execution load of the DEQUANT process in the luminance process. This values decreases as the result of not communicating and computing for the chrominance path. This pulls down the total execution load. In this final design, there still is an unbalance in speed between the chrominance path and the luminance path. See Figure 3.3.17. The execution load of this design is also still unbalanced. The VLD has an execution load close to one, just as the IDCT in the inter path. The BIDIRECTMC process does not exceed the 50% when looking at the execution load. Also the DEQUANT process and the IDCT process in the intra path show a too small execution load. It could be useful to merge the IDCT process of the intra path of the luminance into the DEQUANT process. The execution load of the resulting node should be close to one. This step is only useful when considering the length of the test sequences and namely the relative number of intra blocks in the sequence. This means that the test sequences that are long and contain relatively few intra blocks, should benefit most from this merger. The merger of the DECODER and the BIDIRECTMC in the luminance path should increase the execution load to almost one. The restart of both processes indicates that this isn't a smart thing to do.

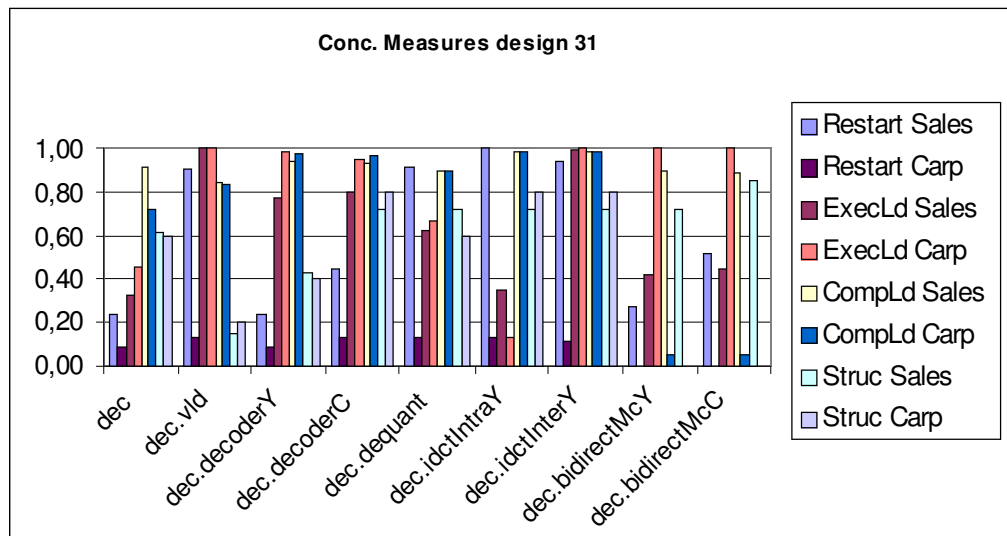


Figure 3.3.17 Concurrency measures for design 31

At the end of the exploration it can be noticed that the values structure and synchronization have not been used. The exploration only focused on the restart, execution load and the computation load. The computation load was ok for most of the designs, which meant that every node in a design is more computing than communicating. The BIDIRECT process influenced the computation load in a negative way when no PB-frames were present in the test sequence. More comments on this issue can be found in the Problems section. The global numbers of the concurrency measures for the designs in total are shown in Appendix .

3.1.9. Result

The result of the exploration is not the best decoder possible with respect to the introduced measures. It is just time that was needed to come to a better solution. The remarks at the end of the previous section are indications of the actions that can be taken to even the unbalance between the two paths. Or perhaps this division between luminance and chrominance paths wasn't the best thing to do. That depends on the results that could be obtained performing the hints at the end of the previous section.

The total exploration resulted in three minor explorations. This is depicted in Figure 3.3.18. This figure shows that design 21, 27 and 31 are the end points of the three minor explorations. These designs are shown in Figure 3.3.19 to Figure 3.3.21 and their concurrency measures are plotted in Figure 3.3.22. The concurrency measures for the other designs can be found in Appendix .

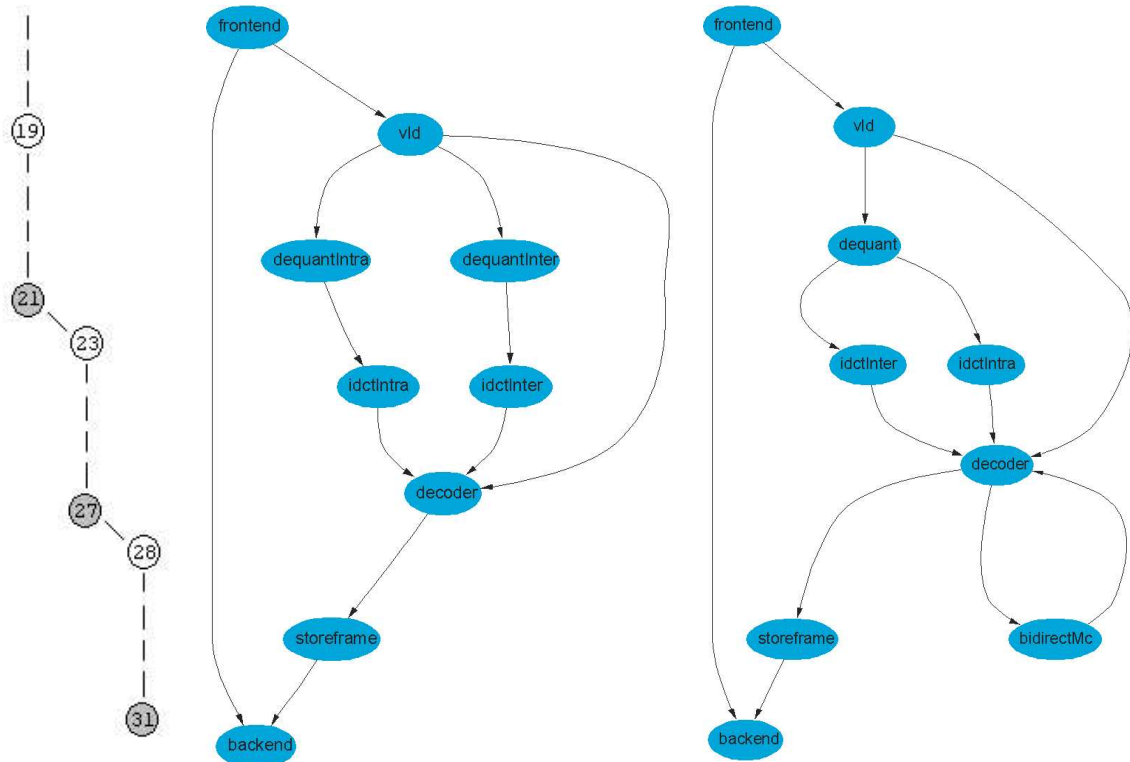


Figure 3.3.18 Design trajectory Figure 3.3.19 Design 21

Figure 3.3.20 Design 27

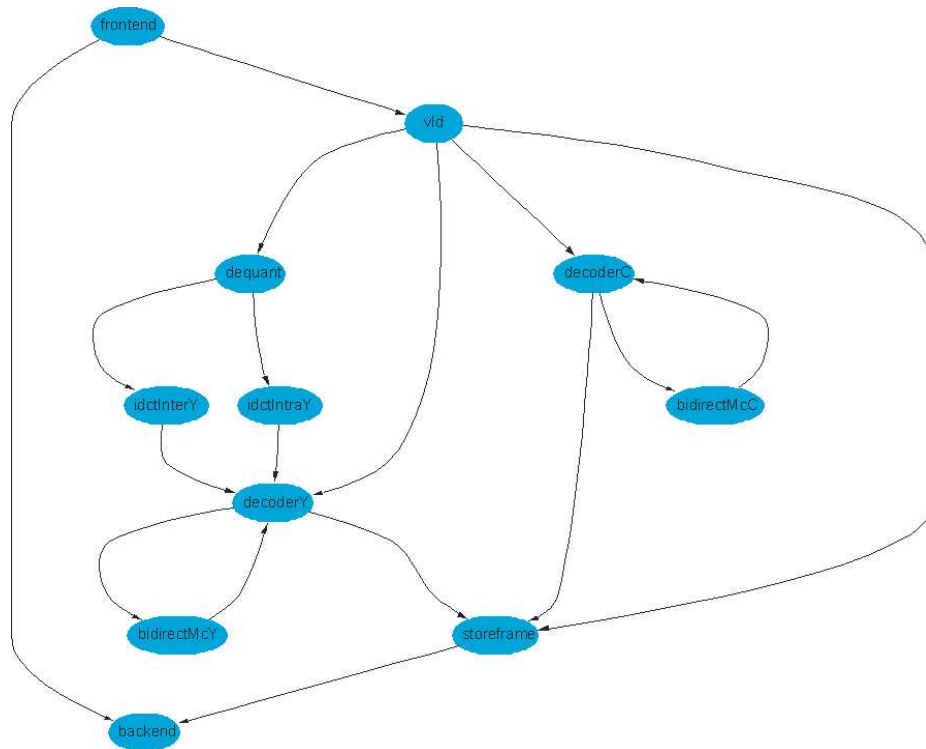


Figure 3.3.21 Design 31

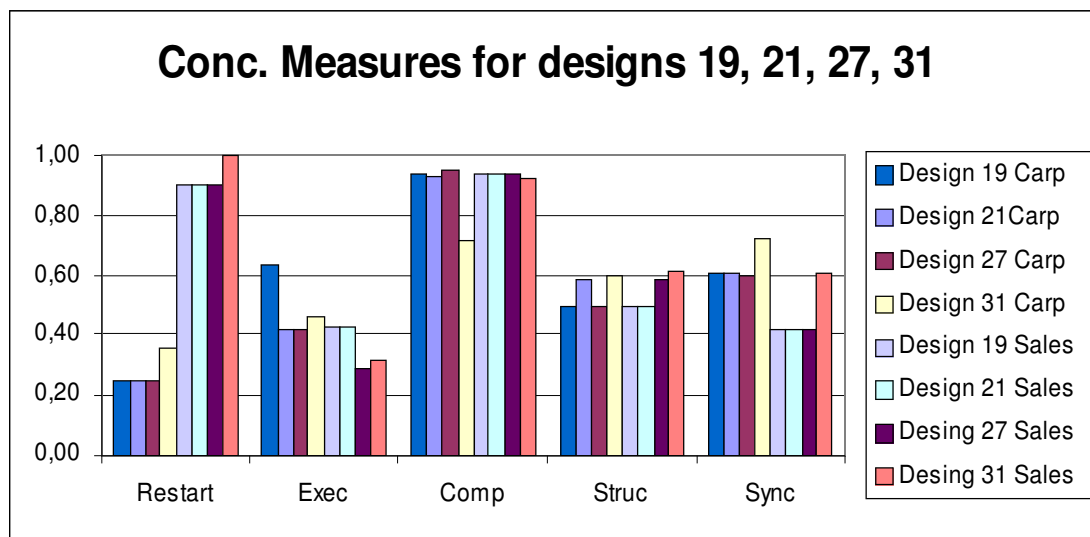


Figure 3.3.22 Concurrency measures for designs 19, 21, 27 and 31

The results of Figure 3.3.22 shows that the restart has been improved from the start design 19 to all the end designs. The structure and the synchronization also improved with respect to the start design.

When looking at these results, it can be noticed that the execution load drops after design 19. The execution load of later designs does not exceed the values of the first designs. This is due to the fact that the values of the restart measure were not evenly distributed in these later designs. The scaling of the contribution of the execution load per node with the restart of that node when calculating the overall execution load, makes these execution load values end up at this relatively low values

The computation load dropped considerably in the last minor exploration. From design 27 to 31 that is and only for the CarpMoreIntra sequence. This is caused by a strange phenomenon of CAST that has

to be sort out. The BIDIRECTMC process, does not do any real computation when no PB-frames are available. Which is the case in the CarpMoreIntra sequence. Normally, the computation load of such a process, is low at that moment. In the first designs that contain the BIDIRECTMC process, CAST gives a computation load for this process close to one. From designs 29 CAST suddenly decreases this value, without changes in the communication has been made in the BIDIRECTMC process. The exact reason is not know yet. Looking at the CAST log file, it can be found that this BIDIRECTMC process get to the same point in the code for designs 28 and 29. Unfortunately, no exact reason for this phenomenon can be given at this point in time. When the computation load of the BIDICTMC process is ignored, the computation load of the complete design should be in the same order as the computation load values of designs 27 and 21. This can be concluded by the results of the Sales sequence.

The global measures of all the designs indicate that the separate data paths of luminance and chrominance do contribute to faster performance. It doesn't indicate that these split up is the best split up. For longer sequences however, this choice outperforms the data split up of inter and intra blocks. That is clearly demonstrated by the figures of designs 21 to 27 for the inter/intra option, and designs 28 to 31 for the luminance/chrominance option.

The exploration method introduced in [1], has been followed in the most practical sense. That means that the order of the exploration was the not practical way to do the exploration. The start with the functional description of the design was a logical first step. The next step of splitting tasks to enhance the restart was also found to be applicable. But this was mostly done in the functional description of the H263 decoder. The closure of all previous steps, before continuing to the next step of the exploration, was however not practical. This meant that the task and data splitting, communication granularity and the data and task merging were interleaved. Of course the overlap in the steps can become much smaller when a large insight of the code is available. Also a larger time span to conduct the exploration method can help to perform the exploration according to the steps ordered as proposed. With the knowledge gained by experiencing one exploration a refined exploration strategy is proposed. This strategy is not followed during the performed exploration, but reflects the thoughts that have been developed during the exploration. The proposed order of the exploration strategy is described next. Hereafter some remarks about this altered version of the exploration strategy are made.

1. Starting point
 - Create computational network according to the functional schema of the computation.
2. Task splitting
 - If the restart in the design is not evenly spread for all the processes, do this step as proposed in the original exploration strategy. If the restart is evenly spread over the designs, go to next step.
3. Communication granularity
 - This step is needed after every splitting task. Check if the computation load of nodes is low, while having a high execution load. This indicates that processes are waiting a long time with respect to their computational effort. Then perform this step as described in the original exploration strategy. When this step is performed, thus if a communication behavior has changed, go back to step 2. Otherwise go to step 4.
4. Data splitting
 - When tasks are split in step 2, then it may be possible to split data. Do this data splitting as much as possible as described in the original exploration strategy, in order to increase the structure. If data paths are created, go back to step 3. Otherwise continue with step 5.
5. Data merging
 - The paths created in the previous steps, can now be merged together to some extend, in order to increase the execution load. Do this as described in the original exploration strategy.
6. Task merging
 - Decrease the number of nodes in computational paths, in order to increase the execution load, and perhaps, even the restart a little when the processes heavily depend on each other. Perform this step as presented in the original exploration

strategy. If this step and the previous steps resulted in any change of the design, go to step 2. Stop the exploration if no step changes anything to the design.

This altered form of the exploration strategy uses the same steps as the original strategy with the explicit mentioning of the starting point. This starting point has found to be very important during the design exploration. That is the reason that it is mentioned explicit. A good functional split up of the design, can give a head start in the exploration. In this case study, the starting point already inhabited the task splitting of the first minor exploration. It can be imagined that this functional split-up will push the designer in the wrong direction. It is not said therefore that the functional schema of an application is the best split-up that can be made.

The ordering of the steps of the strategy is a bit different and the transition to a next strategy step is more specified. Instead of a flow from step 1 to step 6 without loops, this strategy becomes a flow with loops. In Appendix a flow chart is drawn, that visualizes the steps described above.

The iterative nature of this design trajectory could lead to a long design trajectory. Of course this trajectory should be as short as possible. The loops introduced in this altered strategy are mainly caused by the unfamiliarity with the code of the application and the needed programming skills. So a better understanding of the code should make the number of loops the designer makes decrease enormous. If the loops should be banned form the exploration strategy is something that is not clear yet. My opinion is that the splitting and merging steps are not optimizing just one concurrency measure per step. This introduces the loops. But if the designer id able to overlook the steps that have to be taken and the steps that are already taken, then the number of loops that are necessary can be decreased.

The order of the splitting task is rather random. During the exploration no mentionable advantage or disadvantage of the proposed order were encountered. So the order has been the same as in the proposed exploration. The time to perform the communication granularity has changed. In this alter version, after every splitting task communication granularity is proposed. After the task splitting task, it is obvious that the size of the to communicating blocks are optimized. After the data splitting task this is different. Because after the task splitting step an optimal way of communication has been obtained, so when creating extra parallel paths this should be optimal as well. This sounds reasonable, but it's not convincing enough. The data split step could make it for a few paths perhaps easier to communicate in different amounts. When looking at the data path in the performed exploration between chrominance and luminance, then perhaps working on frame bases could slow the chrominance path down to smoothen the two paths with respect to the restart. But that has not been explored at this point.

In the latest proposed exploration scheme in [2] the two merging steps were combined into one merge step. The altered version of the exploration does not inherit this, because at the time the two steps were merged, There was no knowledge of the change in the exploration strategy. Also it was not found to be convenient when combining these two steps. If these two steps are observed separately a more intuitive strategy is formed. Although the goal of the two merging steps is roughly the same, which is in favor of combining the two steps.

3.2Problems

This section describes in brief the problems encountered while pursuing the exploration.

The first and most time taking issue was the lack of a well documented manual of the source code. The source code used was public software made available by Telenor Telecom. At the start of the exploration there is tried to obtain a manual for the source code, but Telenor didn't offer it anymore, even when trying to contact them personally. The absence of the manual made it difficult to exactly understand the source code, and during the exploration its absence came back many times when wrong assumptions were made on the way the code worked. This was particular the case when extracting the VLD out of the DECODER and when the BIDIRECTMC was extracted from the decoder. These two steps took the largest amount of time during the exploration.

Another fact that slowed the exploration down was the lack of programming skills. Although in the first two years at the university programming in C and C++ was discussed, it was demonstrated clearly during the design trajectory that the programming skills weren't developed enough to start this kind of exploration. Many thanks in that respect to colleagues who helped me to develop my programming skills. This thanks also applies for the fruitful discussions on how the software could work, this with respect to the previously mentioned issue.

The true problems during the exploration were the extraction of the VLD and the extraction of the BIDIRECTMC out of the DECODER. Starting with the VLD, this step was crucial in understanding the structure of the software. As stated above, no documentation was available at that point in time. The solution of extracting the VLD was to copy the DECODER with VLD functionality, throw away this functionality in the DECODER. Then use the already available buffer techniques and option flow in both the DECODER and VLD in order to let them communicate with each other. In this case that meant that the buffers in the DECODER were now read by the decoder with fixed number of bits. While in the VLD the buffers, naturally, were read with possibly varying word lengths. The structure of the buffers however stayed the same, in order to keep the changes as small as possible. This implies that the code was not optimized during the extraction of the VLD process and new attempts of extracting the VLD from the DECODER can result in more practical and manageable code.

When extracting the BIDIRECTMC it was necessary to know how the frames were represented in the software. Because of the absence of the manual, this also introduced a considerable time loss. It was possible however to implement this process rather flexible. By flexible it is meant that a lot of possible design choices were available. This mostly concentrated on the way of communicating the desired data from the DECODER to the BIDIRECTMC. After some steps it showed that communicating blocks of 64 pixels was the smartest thing to do. The DECODER could process other block in the mean time, while the BIDIRECTMC could already start with the available block. Because the BIDIRECTMC worked on blocks, this was the minimum amount of data needed by the BIDIRECTMC process to efficiently work.

These two exploration steps were the largest obstacles in the design trajectory. As can be concluded from the fact that both obstacles were splitting steps, splitting steps were found to be the hardest ones. For splitting a process more knowledge about the implementation of the software is required then for the merger of processes.

To end this chapter one remark is made on the behavior of CAST. When evaluating designs 25 and 26, it was found that the computation load of one process had a surprisingly different value in one design than in the other design. Nothing was changed between the designs with respect to the amount of data passed to the process and also the order of data passing had remained the same. Because this difference doesn't seem to be caused by the input of CAST, it is believed that this phenomenon is caused by CAST. The exact reason of this difference is not known yet, but it is under consideration by Sander Stuijk.

4. Conclusion

This section tries to answer the goals stated in section 1.3. Section 4.1 addresses the relevance of the concurrency measures. In the next section, 4.2, the conclusion on the proposed strategy is presented, whereas section 4.3 gives some comments on the used and under construction being graphical user interface (GUI). This chapter ends in section 4.4 giving some final remarks on the final design, the possible next steps and on the case study itself.

4.1 Relevance of the concurrency measures

Because the exploration didn't result in an optimal design, yet, it is difficult to answer the question if all the computation measures are of relevance when optimizing an application with respect to concurrency. A few remarks are given with respect to this question anyhow. Whether the measures are of relevance of the optimization of concurrency is left aside for most of the measures.

The computation load had only little influence in the design exploration. This is mainly caused by the fact that most of the processes had a computation load close to one during the exploration. Whenever a process had a significant lower computation load value than one, then this problem was dealt with right away. Because of the nature of the design, relatively many computations per data element, it was not likely that the computation load could be the bottleneck of the design. However it is though that this measure is of importance when optimizing a design for concurrency when other designs are addressed. Designs that could be split up in smaller computation parts for example.

It was found that the dependency of the execution load on the restart was difficult to overlook. When optimizing the execution load, the restart values could change, which after improving the restart values changed the execution load in the wrong direction. This can be explained when looking at the definition of the two measures in [1]. The execution load of one design is the restart of that process multiplied by the execution time. That is the time that a process is computing and communicating. The restart is de inverse of the run time of a process. The runtime is de execution time and the communication idle time of the process. So when the restart of a process is increased, the execution time could be lowered. But that lowers the execution load. (According to the definition of the restart and the execution load measure [1].) Also the dependency of the restart and the execution load of one process with respect to the execution load of the complete design was tricky to deal with. This dependency causes that if the restart of the design is not well balanced, that the execution load of the complete design is low because of the idle time that the process has due to the communication. That's why during the complete exploration the restart was tried to smoothen as much as possible. This was also the major cause of the loops in the altered exploration strategy.

The relevance of the synchronization for the exploration method was very little. It is however a good indication whether speed up has been made. And this is of course an important goal, when optimizing applications. Of course throughput of the design is also very important in streaming applications, generating enough frames per second in video applications. The speedup makes delays in real-time applications shorter. These delays are important for real-time applications, such as the H263 decoder used for videoconferencing systems, when delays of a few hundreds of milliseconds are found to be very annoying. Therefore the opinion is that the synchronization could be a useful indication, although it is not used in the design strategy.

Evaluating the use of the structure measure during the exploration there can be concluded that this measure was not used. Just looking at the description of this measure, the assumption arises that this measure is a good indication for the amount of parallelism in the design. Because the lack of time, this relevance could not have been given a more firm fundament during this case study. Also in a rather late stage of the case study the parameters were found that produced the desired structure value. So the data splitting steps were all based on the restart values, as can be concluded from sections 3.1.1 to 3.1.8.

An important claim of the used concurrence model [1], states that little knowledge about the application is necessary to get good results. This case study did not required real in depth knowledge

about building blocks of the H263 decoder. For instance no knowledge about the IDCT or Variable length coding was necessary. But extensive knowledge was needed to on the way the building blocks communicated with each other. So the specific knowledge about functional implementation was not found to be necessary, but a clear overview on the coherence of the different building blocks were found to be very important.

Summarizing it can be stated that all the values could be valuable when optimizing the design with respect to concurrency. The dependencies between measures makes the design exploration more difficult and the dependencies also introduce loops in the design strategy. Not every measure is used during the presented improved exploration strategy, but perhaps the measures that aren't used, can be incorporated in the strategy at later points in time.

4.2 Exploration strategy

The proposed exploration strategy assumes an ideal exploration. The performed exploration was all but ideal. Because of the issues mentioned in section 3.2, the proposed order could not be followed. Therefore an altered version of the exploration strategy is presented 3.1.9. This altered version is the result of the thoughts that have been formed during the performed exploration and is not claimed to be better than the proposed strategy but merely a refining. The main reason for the loops in the altered version of the exploration strategy is the difficulty to overlook the enormous amount of code. This task should be better to perform when the designer is more experienced. Then the loops perhaps can be left out, because then all the steps can be over seen. With the little available experience it is hard to argue whether these loops can become obsolete. The exploration does not fully optimize all concurrency measures, such as the synchronization. Because speedup is an important goal in optimizing concurrency perhaps more research could be aimed at precise steps to optimize this measure.

4.3 The GUI

If time was left, the GUI should be reviewed. Unfortunately there was no time left. However there has been some interaction with the GUI during the design exploration. That is the reason why in this section some remarks will be made addressing the GUI.

First of all, the GUI is a good start to visualizing the design in an intuitive way. The used colors for the display of the network mappings as well as the sizes of the nodes when finding the bottlenecks are quite useful. Also the possibility to show the graph is in itself a convenient way of visualizing the design.

The possibility to look at the event traces has not been used frequently. But the one time it has been used, it showed its value. This also can be said about the code accessing possibility from the displayed event trace.

Unfortunately the exporting of all the graphics was not fully implemented yet. That is also the reason that the results in this report are shown in excel bars, and not by the bar charts of the GUI. Another practical comment with respect to the exporting is the file format of the exported graphics. In the used version of the GUI, only scalable vector graphics (SVG) could be exported. When using this relatively new kind of graphics it was hard to find any conversion tools. Therefore it could speed up the reporting of the design exploration if the exporting of the graphics were also possible with better supported graphics standards like JPEG, PNG or even EMF.

The fact that only 12 designs could be mounted by the GUI is also something that could be changed. Especially when evaluating a large part of the exploration it can be very useful when a lot of designs could be mounted by the GUI.

The utility to display the event traces is a valuable utility to find communication problems. This tool has the disadvantage that the number of events that can be displayed is limited. Maybe it is possible that the maximum and minimum gap size (computation idle time) are displayed in the GUI at the event

traces. The user could then get the possibility to jump to every gap larger of smaller than an user defined value. In this way it is easier for the designer to jump over the enormous parts of events, which can very enormous in length, that are not of interest for him.

Other, more debug related, information was passed to the designer of the GUI during the exploration. This information was focused on the logical functioning of the GUI and shouldn't be mentioned in detail in this report.

To conclude the remarks on the GUI it can be stated that this tool is a convenient and intuitive way of absorbing and interpreting the measures of ones designs. Some work has to be done however in perfecting this tool qua functionality and debugging.

4.4 Future work

The resulted design of the exploration did not end up to be the optimal design. To improve the design, it would be best to choose whether PB-frames are used in the sequences. In that way, the influence of the backward motion compensation does not cause problems with sequences without PB-frames. The statement that the number of options should be minimized when optimizing a design sounds logical. If fewer options need to be incorporated in the design it is more likely to get a more optimized solution. This case study reinforces that statement. This applies mostly to studies that evaluate the concurrency measures and strategy. But in general you can say that more options in an application will result in a larger compromise in the resulting optimization. Therefore in studies it is best to optimize the basic version of the application. For optimizations at the end of the development of the exploration strategy, many extra options of an application should be taken into account with the optimization.

Another way to improve the design is to re-implement the DECODER completely. In that way some data dependency with the VLD could be lost. This step could take a huge amount of time, so no attempt has been made to try to re-implement the DECODER completely.

This case study did not vary the cost of communication. In [1] it was stated that the cost of communication could have a great influence on the computation load. In order to test this, it could be useful to vary the costs of communication in the various designs. It is expected that due to the enormous amount of computation done by the processing nodes, the influence of the communication cost is not as dramatically as assumed in [1]. The fact that the current processes of the design do relative much computation on one data element, it is not thought that this measure will drop very far. Assuming that realistic cost parameters are used. When one designs an application where a process can do little computation on one data element, the assumption that the cost have a great influence on the computation load is correct. But that is not the case in the application under study. The influence of the communication cost parameters on the application under study can change, when one manages to break the current processes down into a lot of smaller processes especially the DECODER process.

There has not been looked at the influence of the implementation architecture when performing the exploration. This could be done for example by varying the costs of communication. But also the tuning of the code for certain architectures could be one of the possibilities. This last possibility makes the outcomes of CAST not platform independent. But if the platform for the design is already known, this could be a way to get a better optimized design.

The results of the exploration show improved measures for concurrency. These results have not been tested on a real implementation, that show that the improved measures are in fast improvements for the performance of the application. The exact way of testing these results are not known by the writer of this report at this point in time. It must be noted that these tests are needed in order to validate the outcomes of this case study

Bibliography

- [1] Sander Stuijk. *Concurrency in computational networks* (Master thesis) Technical University Eindhoven, October 2002
- [2] S. Stuijk and T. Basten. *Analyzing Concurrency in Computational Networks* (Extended Abstract)
In Formal Methods and Models for Codesign, 1st ACM & IEEE International Conference, MEMOCODE'2003, Proceedings. Mont Saint-Michel, France, 24-26 June, 2003. IEEE Computer Society Press, Los Alamitos, CA, USA, 2003.
- [3] Kock, E.A. de, and G. Essink. *Y-chart Application Programmer's Interface. Application programmer's guide version 1.0.1*. Philips Research, Eindhoven , 2001.
- [4] S. Stuijk, T. Basten and J. Ypma. *CAST - A Task-Level Concurrency Analysis Tool* (Tool paper.)
In Application of Concurrency to System Design, 3rd International Conference, ACSD 03, Proceedings. Guimarães, Portugal, 18-20 June, 2003. IEEE Computer Society Press, Los Alamitos, CA, USA, 2003.
- [5] ITU-I Recommendation H.263, February 1998
- [6] Andre Carmo. *A study on the transformation of sequential streaming algorithms into parallel YAPI programs* (Stage report). Technical University Eindhoven
- [7] Karl Olav Lillevold. *TMNDecoder source code*. Telenor R&D, Norway, 1996
- [8] Brian Kernighan, Dennis M. Ritchie. *The C programming language, second edition, C Handboek*, Prentice Hall. Mei 1990
- [9] Mark Allen Weiss. *Data structures and problem solving using C++, second edition*. Addison-Wesley, 2000

Appendix A

Flow chart of altered exploration strategy.

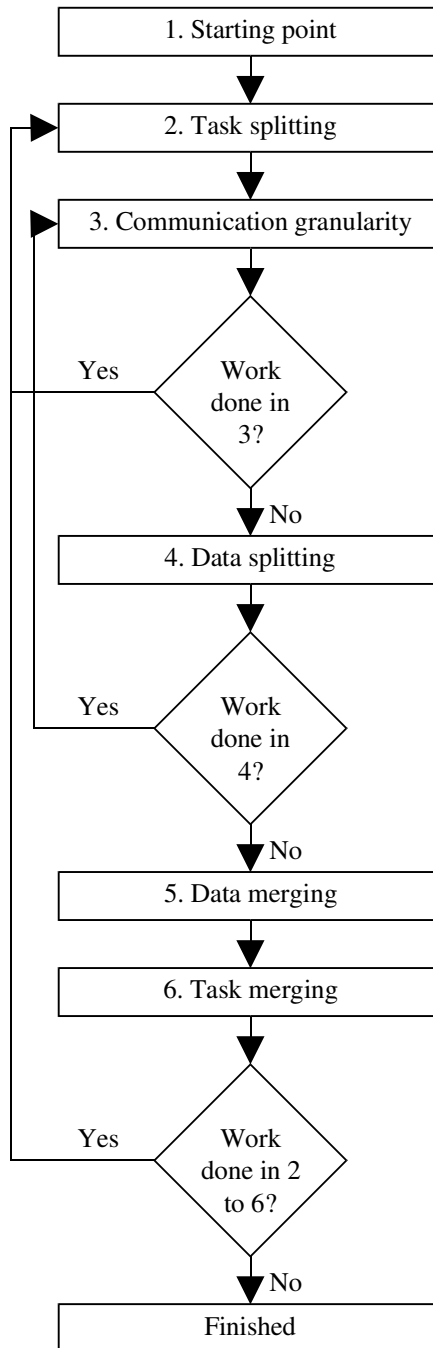


Figure 4.1 Improved design scheme

Appendix B

Global results of design 16 to 31

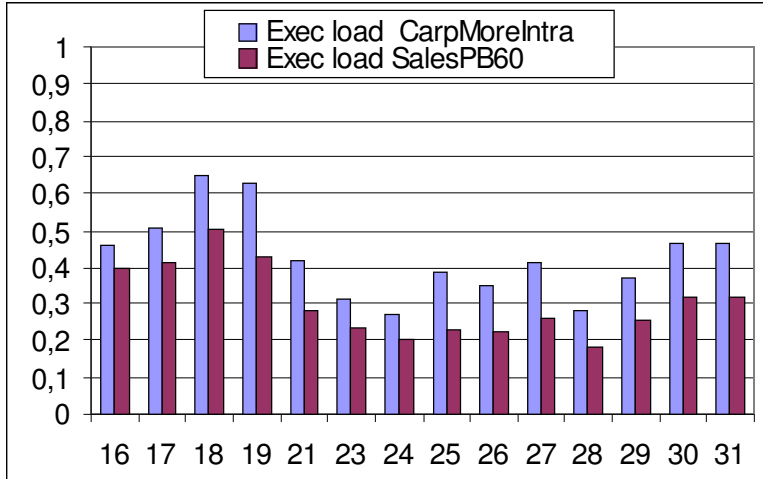


Figure 4.1 Execution load designs 16 to 31

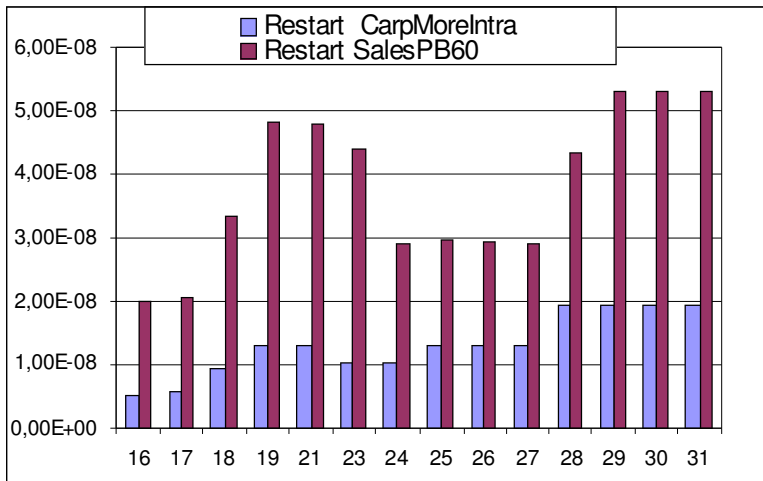


Figure 4.2 Restart measures designs 16 to 31

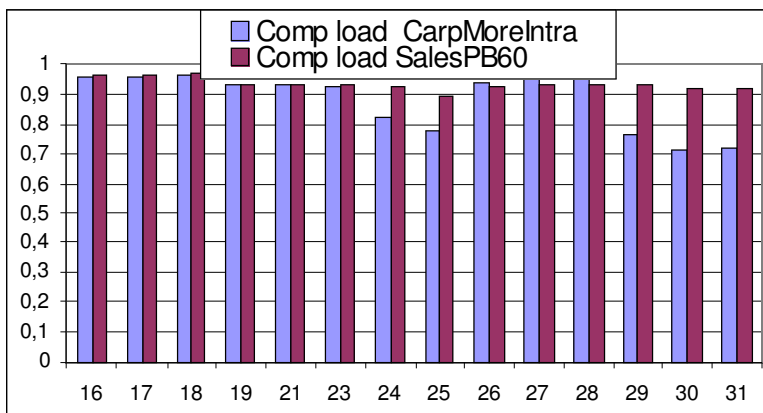


Figure 4.3 Computation load designs 16 to 31

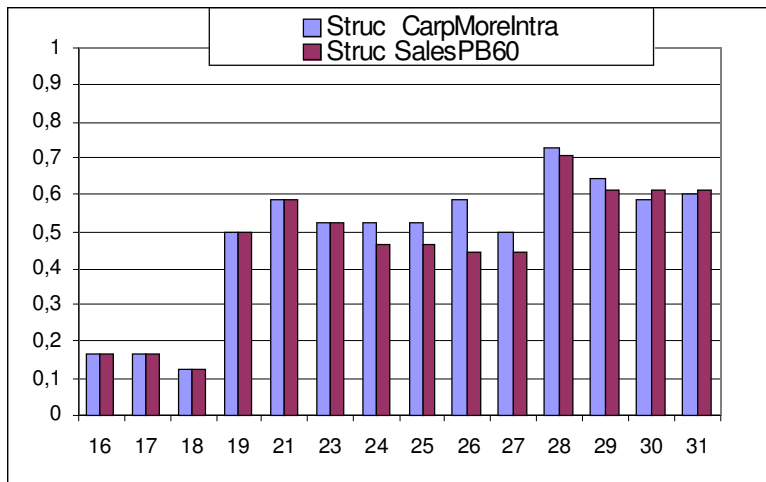


Figure 4.4 Structure measures designs 16 to 31

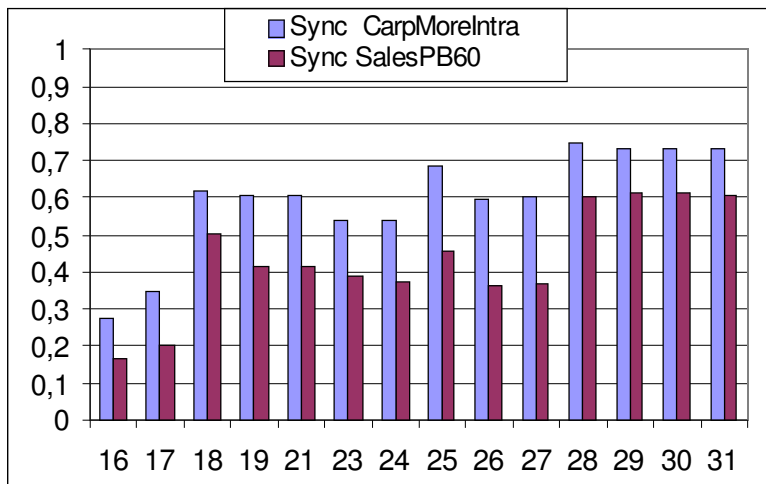


Figure 4.5 Synchronization measures designs 16 to 31

Appendix C

Network graphs graphics

This appendix shows the relevant network graphs that were created during the design exploration.

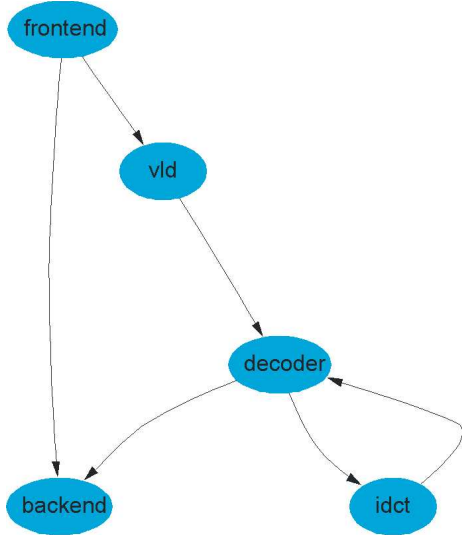


Figure 4.1 Design 16

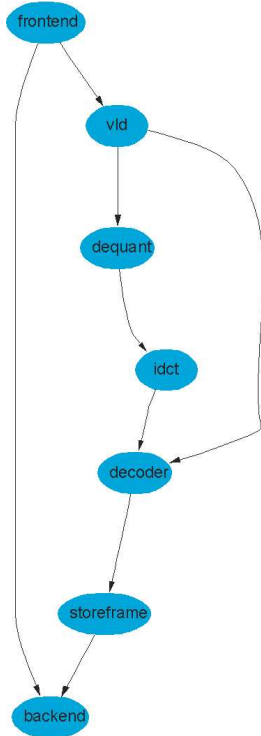


Figure 4.3 Design 19

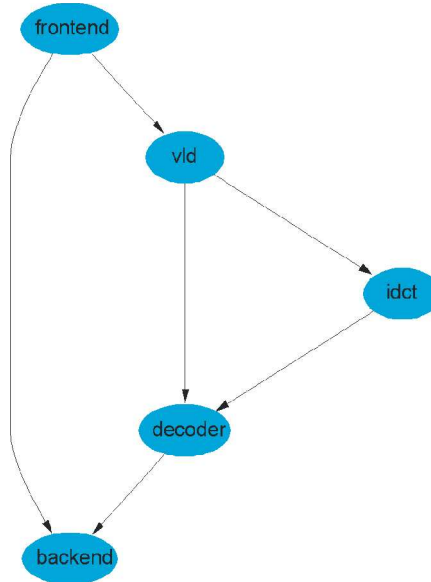


Figure 4.2 Design 17

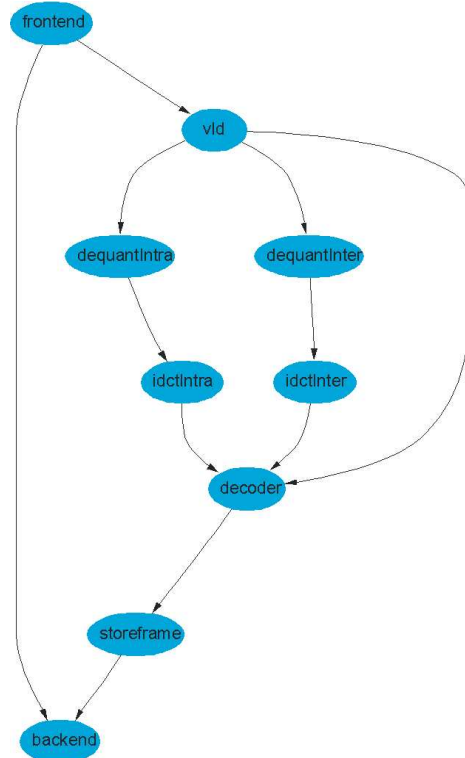


Figure 4.4 Design 21

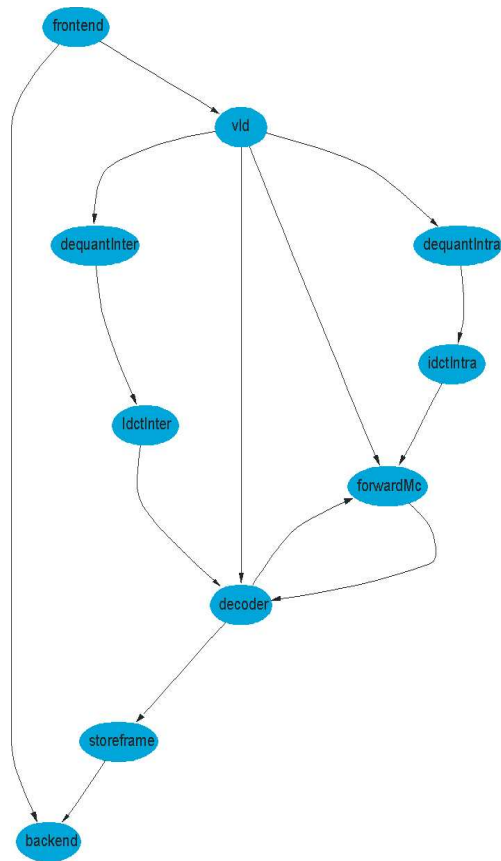


Figure 4.5 Design 23

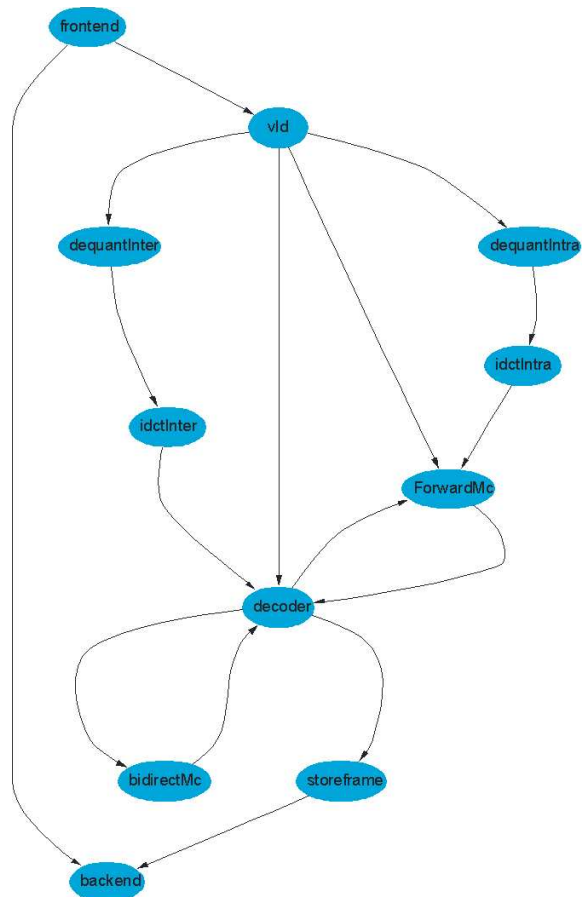


Figure 4.6 Design 24

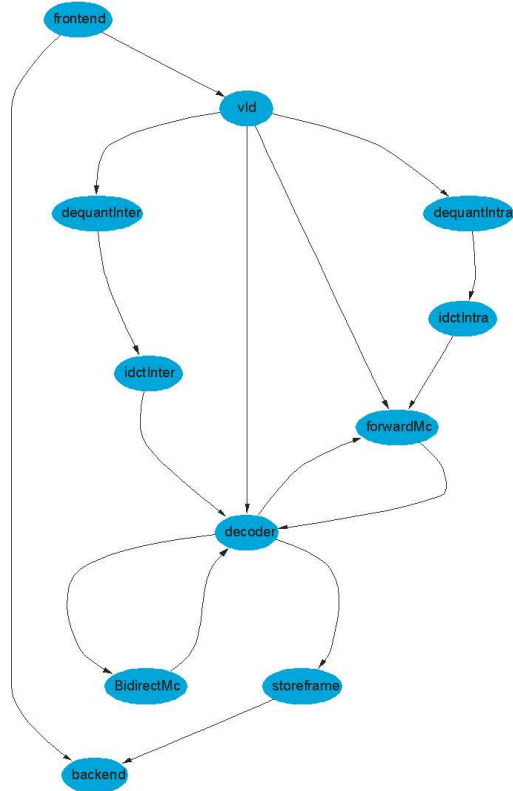


Figure 4.7 Design 25

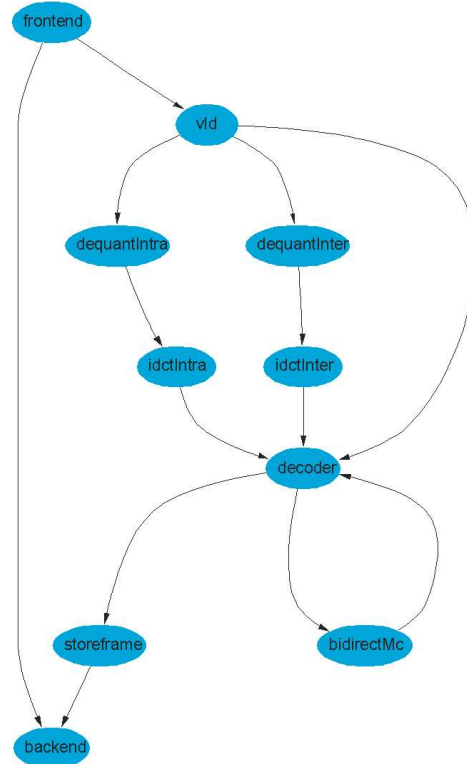


Figure 4.8 Design 26

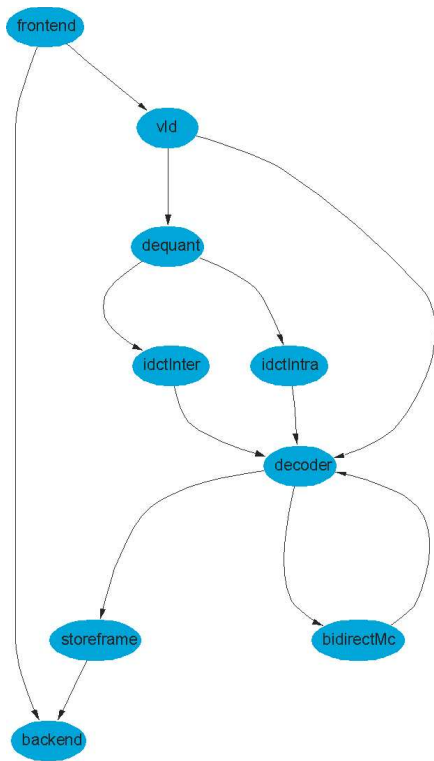


Figure 4.9 Design 27

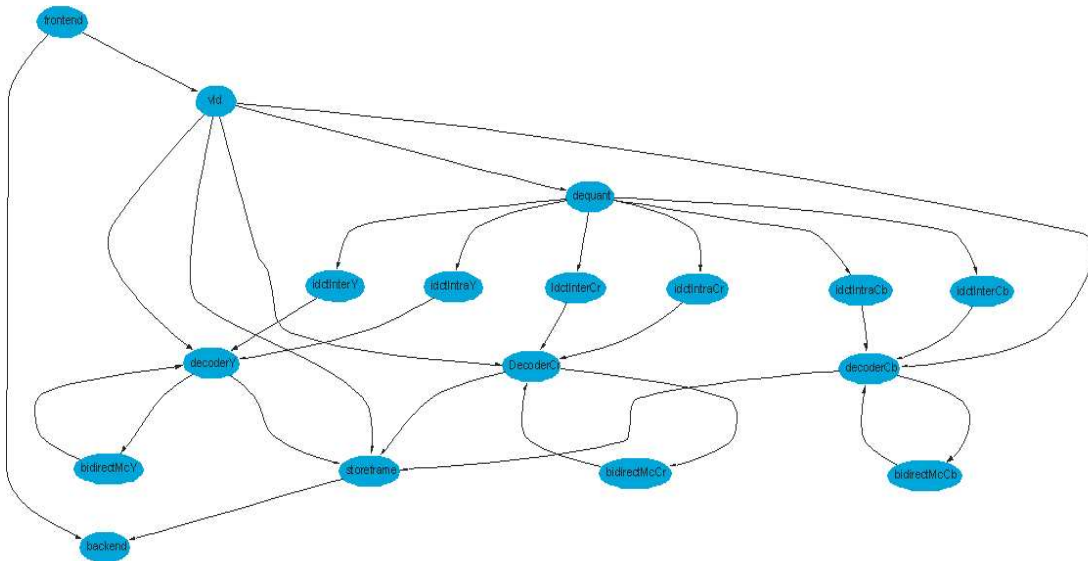


Figure 4.10 Design 28

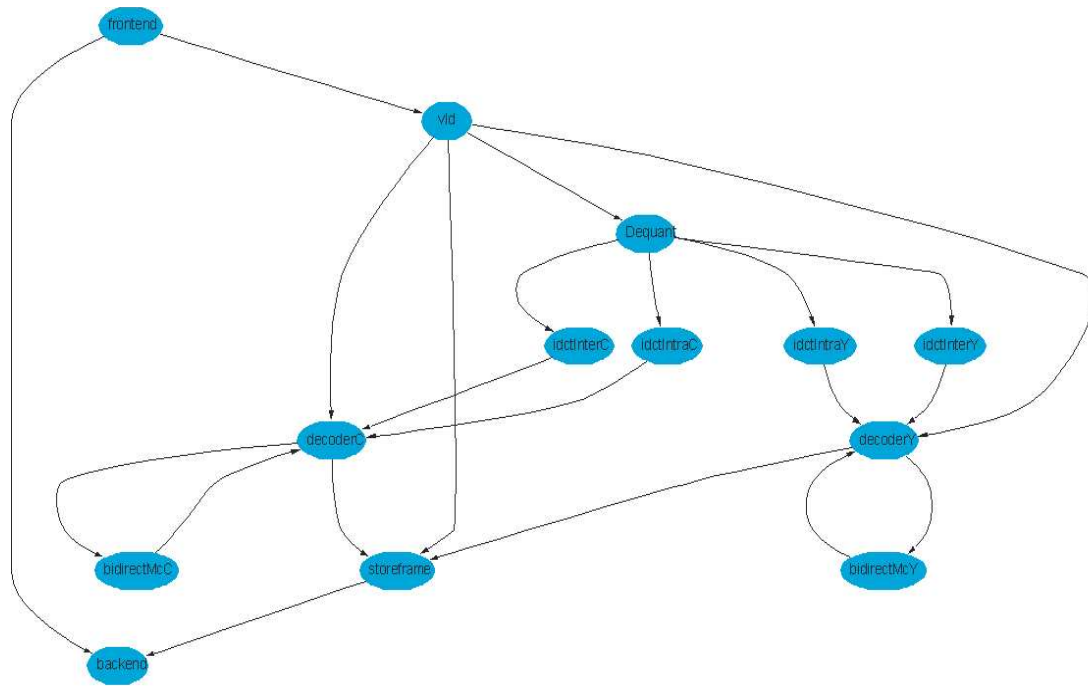


Figure 4.11 Design 29

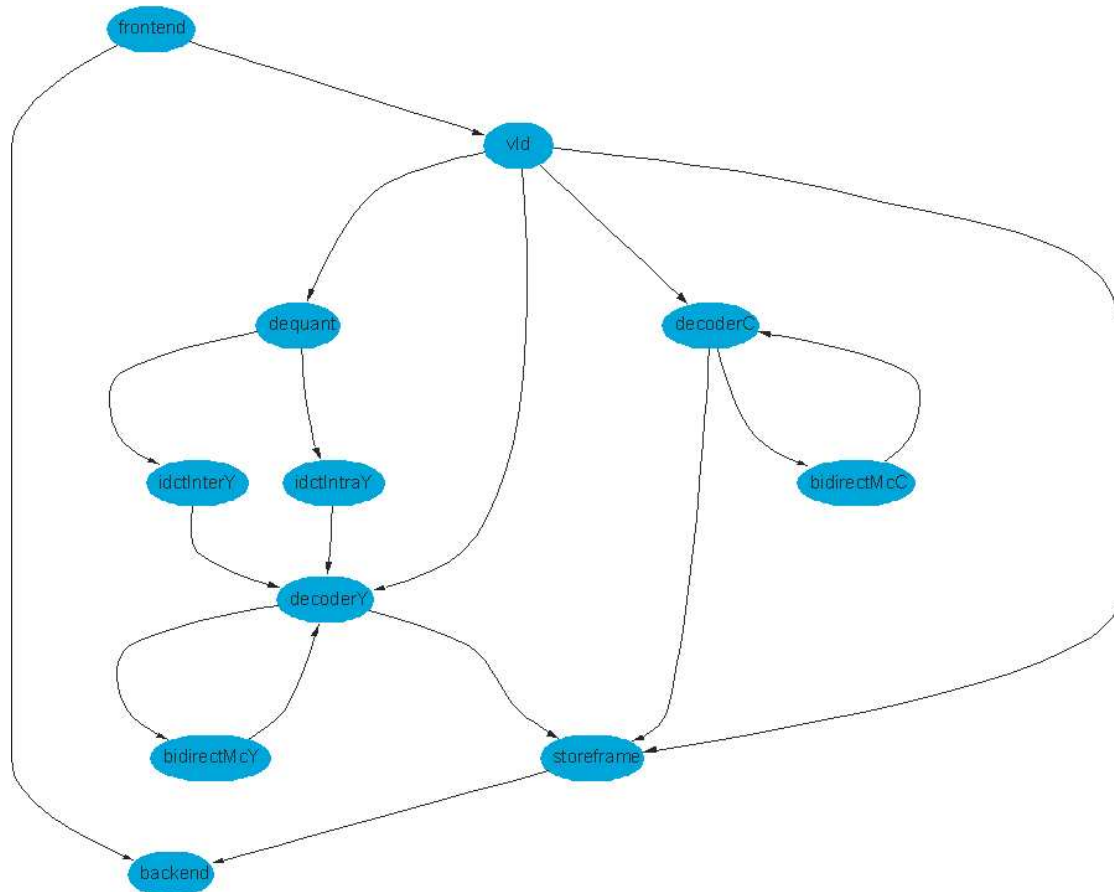


Figure 4.12 Design 30

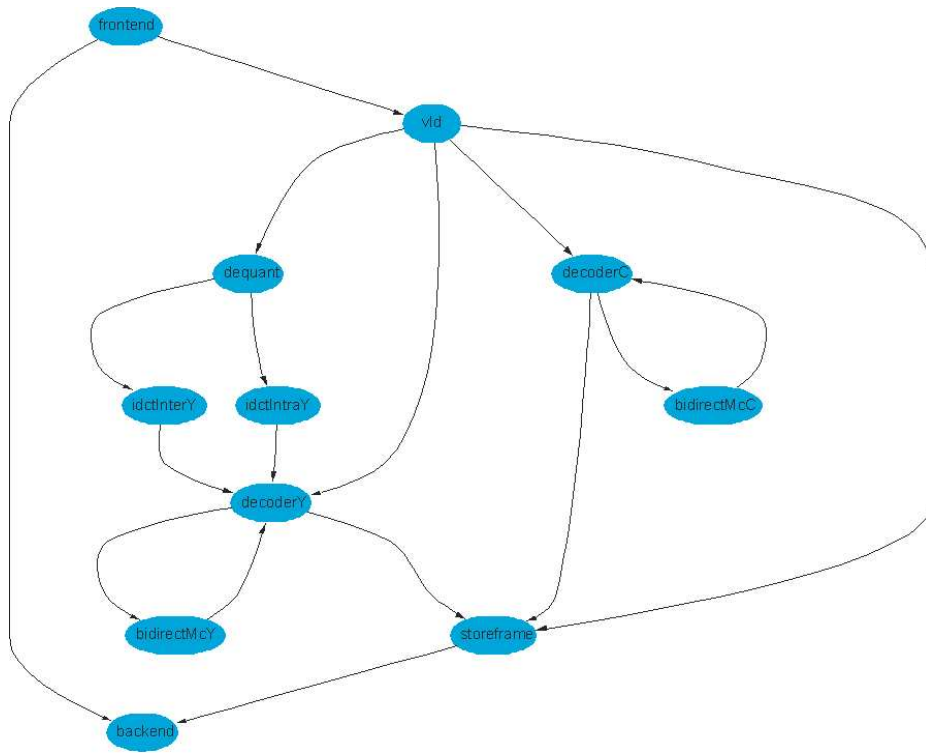


Figure 4.13 Design 31