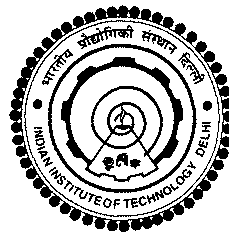


Modeling and Mapping of Multimedia Applications on Multiprocessor Architectures

A Thesis
submitted in partial fulfillment
of the requirements for the degree
of
Master of Technology
in
VLSI Design Tools & Technology
by
Basant Kumar Dwivedi

Under the guidance
of
Prof. M. Balakrishnan
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
and
Dr. Jan Hoogerbrugge **Dr. Paul Stravers**
IST, Philips Research, Eindhoven



Department of Computer Science and Engineering
Indian Institute of Technology Delhi
December 2000

Certificate

This is to certify that the thesis titled **Modeling and Mapping of Multimedia Applications on Multiprocessor Architectures** being submitted by **Basant Kumar Dwivedi** for the award of **Master of Technology in VLSI Design Tools and Technology** is a record of bona fide work carried out by him under our guidance and supervision at the **Department of Computer Science & Engineering, Indian Institute of Technology, Delhi** and **Philips Research, Eindhoven**. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Prof. M. Balakrishnan

Department of Computer Science & Engg.
Indian Institute of Technology, Delhi

Dr. Jan Hoogerbrugge

IST, Philips Research
Eindhoven, The Netherlands

Dr. Paul Stravers

IST, Philips Research
Eindhoven, The Netherlands

Acknowledgments

I am greatly indebted to my supervisors Prof. M. Balakrishnan of IIT Delhi, Dr. Jan Hoogerbrugge and Dr. Paul Stravers of IST, Philips Research, Eindhoven for their invaluable technical guidance and moral support during the course of project. I am grateful to Prof. Anshul Kumar for his suggestions during our weekly project meetings. I am also thankful to Prof. G. S. Visweswaran for his help during the project.

I would like to thank Anup Gangwar for his feedbacks during our informal discussions and my other colleagues for their cooperation and support. I would also like to thank the staff of Philips laboratory, IIT Delhi for their help.

Abstract

The main objective of the project was to expose the parallelism present in MPEG-2 decoder application and identify modeling issues, which can further be used as guidelines to create parallel models and improve performance of other applications. The application MPEG-2 decoder was modeled using YAPI (Y-Chart Application Programmer's Interface). The application model was mapped onto multiprocessor architecture (SpaceCAKE architecture). Performance data was collected at both application and architecture level to identify potential bottlenecks. Then MPEG-2 decoder model was fine tuned to enhance the performance.

Contents

List of Figures	v
List of Tables	vii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Methodology	2
1.4 Outline of Report	3
2 YAPI	5
2.1 Introduction	5
2.2 Kahn Process Network	5
2.3 YAPI	7
2.3.1 Process Network	7
2.3.2 Process	7
2.3.3 Communication	8
2.3.4 Workload Analysis	8
3 MPEG-2	11
3.1 Introduction	11
3.2 MPEG-2 Video Sequence	12
3.3 MPEG-2 Video Sequence Decoding	13
3.3.1 Algorithm	13

3.3.2	Scope of Parallelism	15
4	MPEG-2 Decoder Modeling	17
4.1	The Early Structure	17
4.1.1	The Configuration	17
4.1.2	Drawbacks of the Model	22
4.2	Modeling Issues	23
4.2.1	Amount of Parallelism	23
4.2.2	Unidirectional Communication	24
4.2.3	Communication Overhead	24
4.2.4	Granularity of Operation	24
4.2.5	Balanced Pipelines	25
4.2.6	Synchronization Overhead	25
4.2.7	Multiple Instance of Processes	26
4.2.8	Memory Usage	27
4.2.9	Scalability	27
4.3	Refined MPEG-2 Decoder Model	28
4.3.1	Top level view	28
4.3.2	Process Network MPEG-2 Decoder	29
4.3.3	Process Network TsliceDec	32
4.3.4	Process Network Tmc	34
4.3.5	Process Network Tiq_idct_add	35
4.4	Salient Features of The Model	36
5	Experimental Environment	39
5.1	Introduction	39
5.2	Thread Scheduler	40
5.3	The SpaceCAKE Architecture	41
5.3.1	The Architecture	41
5.3.2	The Tile	41

6 Experiments and Results	45
6.1 YAPI Level Simulations	45
6.1.1 Computation Overhead	45
6.1.2 Communication Overhead	48
6.2 TSS Level Simulations	48
6.2.1 Application Behavior	49
6.2.2 Architecture Behavior	53
7 Conclusions	57
7.1 Conclusions and Contributions	57
7.2 Further Research	58
Bibliography	61
Index	63

List of Figures

1.1	Y-Chart approach	3
2.1	Kahn Process Network	6
3.1	Data organization in MPEG-2 video sequence	13
3.2	Simplified representation of MPEG-2 video sequence decoding	14
4.1	A simple MPEG-2 decoder model in YAPI	19
4.2	A process network demonstrating two pipelines	26
4.3	A process network demonstrating multiple instance of process P	27
4.4	Top level view of complete application	28
4.5	MPEG-2 decoder model	30
4.6	Process Network TsliceDec	33
4.7	Process Network Tmc	35
4.8	Process Network Tiq_idct_add	37
5.1	Layers of experimental environment	39
5.2	Thread scheduling	40
5.3	The SpaceCAKE Architecture	41
5.4	Structure of tile	42
6.1	Number of cycles vs Number of CPUs	50
6.2	CPI vs Number of CPUs	51
6.3	Bus wait cycles vs Number of CPUs	52
6.4	Number of snooping requests vs Number of CPUs	53

6.5	Number of coherence writes vs Number of CPUs	54
6.6	Number of cycles vs Parallelism Number	54
6.7	Normalized architecture parameters vs CPU id	55

List of Tables

6.1	Computation workload	47
6.2	Comparison among various application models	48

Chapter 1

Introduction

1.1 Motivation

Recent advances in network and microprocessor technology have made it possible to introduce a new set of applications and services. High Definition TV (HDTV), Broadcast Satellite Service, Video-conferencing, Interactive Storage Media etc. are few typical examples. These applications need huge amount of data processing both in video and audio domain. The high data rates involved in the applications make computation very time consuming.

Designers mainly follow two architectural approaches for signal processing systems, dedicated and programmable. Dedicated architectures target an algorithm or a set of algorithms. These architectures fully exploit the computational features of algorithm and VLSI implementations of dedicated architectures are optimized for area, power and performance. A good overview of architectural approaches for signal processing systems has been given in [1]. Though dedicated architectures provide good performance, they lack flexibility to further extend the algorithm set.

On the other hand the programmable approach offers a number of advantages. Programmable solutions offer greater flexibility, as the target algorithm set can further be extended by applying proper software modifications. Since a large number of applications can run on the same hardware, per application

hardware cost is reduced. On the other hand, as computational properties of algorithms are not fully exploited, it requires that both the architecture and application models be faster. Architecture can be made faster by employing multiprocessing strategy. Hence a parallel model of application running over a multiprocessor architecture offers one of the potential solutions.

The computational resources of a multiprocessor architecture can be exploited fully only when the application model has sufficient parallelism. Parallelism present in signal processing applications can be made explicit using models based on Kahn process networks [10]. Several variants of this model have been reported including [13] and [3]. The drawback of models based on Kahn process network is that they cannot model reactivity. This limitation is overcome in control flow models such as Statecharts, Esterel and Polis. Once a parallel model of the application is built, significant speed up can be achieved when application runs over a multiprocessor architecture.

1.2 Objectives

The main objective of this work was to identify several modeling issues involved in creating a parallel model using YAPI [3], which can further be used as a starting point to expand the application set and estimate the performance of other application models.

1.3 Methodology

In this project, the *Y-chart* [12] approach has been followed. Figure 1.1 describes this approach. Hence, the work was divided into four stages:

1. Modeling of application MPEG-2 decoder using YAPI.
2. Mapping of application onto a specific multiprocessor architecture being developed within Philips. This architecture (SpaceCAKE architecture) has been described in Chapter 5.

3. Collection and analysis of performance data from simulations to identify potential bottlenecks.
4. Fine tuning of application to match architecture closely and repeating various stages for satisfactory performance.

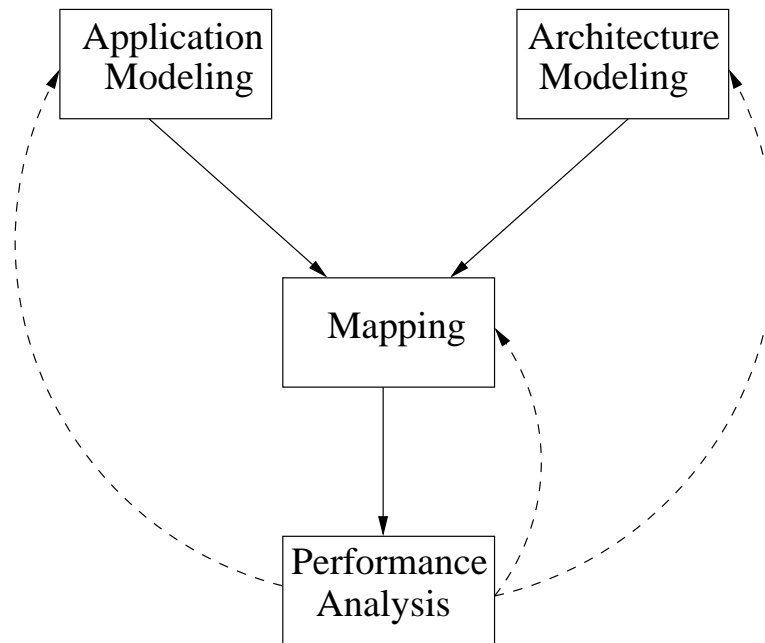


Figure 1.1: Y-Chart approach

1.4 Outline of Report

Chapter 2 discusses YAPI and Kahn process networks. Chapter 3 describes MPEG-2 video sequence organization and discusses scope of parallelism in MPEG-2 decoder application. Chapter 4 discusses various modeling issues and describes MPEG-2 decoder application modeling in particular. Chapter 5 gives details of experiment set up. Chapter 6 provides details of analysis of performance data and its feedback and Chapter 7 concludes.

Chapter 2

YAPI

2.1 Introduction

Most of the signal processing applications are quite computation intensive. These applications process large amount of data at high data rates. This is especially true for video processing applications. Throughput requirements of these applications can be met by exploiting inherent parallelism present in the application. At this end YAPI [2, 3] facilitates exposing the parallelism present in the application.

Application modeling is the first step in Y-chart [12] which is essentially the functional specification of the system. Using YAPI the application can be modeled based on Kahn process network [10] . This model can be reused across various platforms. YAPI also provides a run time environment for efficient execution on a workstation. The application model can be further used as an input for platform dependent mapping and performance analysis.

2.2 Kahn Process Network

The Kahn Process Network is a model of parallel computation. In this model a number of concurrent processes communicate through unidirectional FIFO channels (Figure 2.1). A process follows sequential execution. At any given

time, a process is either computing, waiting for information on one of its input lines or sending a token.

Process Network

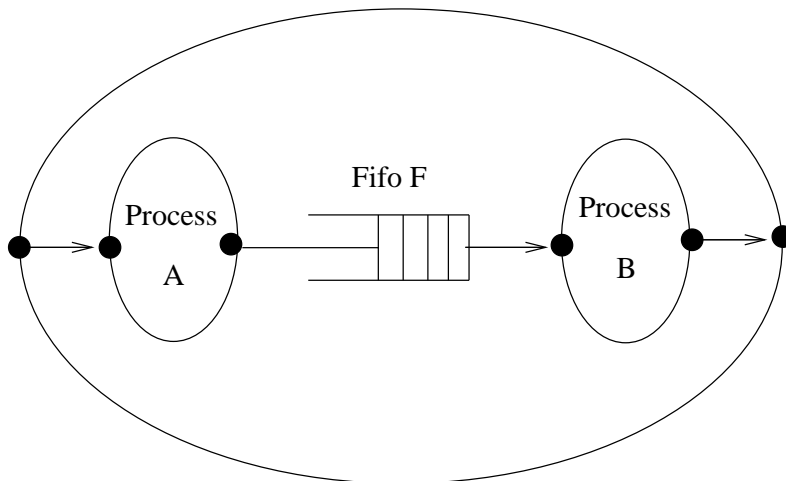


Figure 2.1: Kahn Process Network

Communication channels are the only way of communication among processes. A communication channel transmits information within an unpredictable but finite amount of time. Moreover only one process is allowed to put information into a particular channel and only one process is allowed to receive information from that channel. Reads on the channels are blocking.

The behavior of communication channels makes the Kahn process network deterministic. The information moving in a channel is determined by only input data. The order in which processes are being executed does not affect this. This deterministic property can be exploited to create parallel models for signal processing applications irrespective of order of execution of processes.

Kahn process network does not model reactivity. The reason is that non deterministic events cannot be made known to processes. This limitation is overcome in control flow models such as finite state machines. In [13] several variants of Kahn process network are discussed.

2.3 YAPI

YAPI is a C++ library with a set of rules which can be used to model signal processing applications as a process network.

2.3.1 Process Network

A process network in YAPI is an instance of class *ProcessNetwork*. Its member objects are a set of processes, process networks and FIFOs. In a process network a process represents a computing station and a FIFO represents communication channel through which two processes interact with each other.

A process network can be part of another process network, an hierarchy of process networks can be created as per requirements of the application. Furthermore, a process network does not have any member function to define its functionality, rather its definition comes from its member objects (processes and process networks). Figure 2.1 shows a simple process network consisting of two processes, A and B communicating over FIFO F.

2.3.2 Process

A process is an instance of class *Process*. It has two parts, interface and behavior. The process defines its behavior in the member function *main*, which can call any other member function. The process follows sequential execution and interacts with its environment through input and output ports. A process can be in one of the three states:

1. **Running state** This indicates that process is doing some computation.
2. **Blocked state** A process is blocked when it tries to read some data at its input port and FIFO connected to that port is empty or when it tries to write some data at its output port and FIFO connected to that output port is full.

3. **Ready state** In this state the process is no longer in blocked state, but it is waiting for its turn to get scheduled.

The interface part of the process is described by its constructor. The argument list of the constructor includes input and output streams. An input stream is an object derived from template class *In* and an output stream is an object derived from template class *Out*. The process actually communicates with its environment using ports. Port types are *InPort* and *OutPort*. When a process is created input and output streams defined in constructor argument list are bound to input and output ports respectively. These ports are connected to FIFOs in the process network at higher level.

2.3.3 Communication

Processes communicate with each other via unidirectional FIFO channels. A channel is an instance of template class *Fifo*. It has one input and one output end. There is exactly one process which reads from the channel and exactly one process which writes into the channel. There are three kinds of primitives provided by YAPI for communication:

Read This is used to read data from input channels.

Write This is used to write data into the output channel.

Select This is used for non deterministic applications. At run time this primitive identifies the port that is connected to the process and that has committed to communicate. The function terminates if such a process can be selected, otherwise it blocks.

2.3.4 Workload Analysis

Two important properties of the application are its computation and communication workload, given the input data. In YAPI the communication workload is measured by counting the number of tokens that are written into

FIFOs. To get computation workload, thread corresponding to each process provides the time taken by that process to complete its job. This workload analysis gives important feedback to fine tune the application. Functions *printCommunicationWorkload()* and *printComputationWorkload()* are used to get workload information.

Chapter 3

MPEG-2

3.1 Introduction

Development of MPEG-1 [16] standard was motivated by the need to remove the incompatibilities present in the systems for video and audio applications and to propose a standard widely accepted by different research communities and industries. MPEG-1 was mainly aiming at bit rates around 1.5Mbits/s and therefore it is more suitable for storage media applications. MPEG-2 [14] evolved from MPEG-1 and addresses more diverse applications such as High Definition TV (HDTV), Video Conferencing, Broadcast Satellite Service and many more.

The main reason of widespread use of MPEG standards is very high technical quality. Even after a number of years, no errors are found in the specification. This led to a large number of implementations based on these standards.

Most of the video processing applications require large amount of data processing and they have to meet real time deadlines because of high bit rates. Though there are constraints such as large amount of data processing and real time deadlines, researchers have been exploring implementations fully in software [5, 9, 11, 15]. Motivation of this is not only to find methodologies (such as parallel models) to speed up the application (e.g. MPEG-2 encoder

and decoder) but also to explore the large design space of next generation computer architectures. In this project, we further extend this exercise by creating a parallel model for MPEG-2 decoder and using this model to study the behavior of SpaceCAKE architecture (Subsection 5.3.1).

This chapter looks at MPEG-2 standard and tries to find places where parallelism can be extracted to create a parallel model for MPEG-2 decoder. Section 3.2 discusses structure of bit stream. Section 3.3 looks at algorithm and discusses the scope of parallelism.

3.2 MPEG-2 Video Sequence

Figure 3.1 gives an overview of different layers of MPEG-2 video sequence. An MPEG-2 video sequence starts with a sequence header immediately followed by `extension_start_code`¹. After the required sequence extension, Group of Pictures (GOP) headers and picture(s) repeat. The sequence ends with *sequence_end_code*.

The GOP header is always optional². GOP header may be followed by user data. At least one picture always follows each GOP header.

A picture header is always followed by the picture coding extension, other extensions, optional user data and picture data. There are three different types of pictures, I, P and B. I picture (intra-coded picture) is coded independently. P picture (predictive-coded picture) obtains predictions from preceding I or P picture. B picture (bidirectional-coded picture) obtains predictions from the nearest preceding and/or upcoming I and P pictures in the sequence. The picture data consists of several slices. Each slice has its own header followed by several macroblocks.

A macroblock has 6 to 12 blocks depending on chroma format. For 4:2:0 chroma format there are 6 blocks, 4 luminance blocks and 2 chrominance blocks. For the 4:2:2 or 4:4:4 optional formats an additional two or six blocks

¹Absence of `extension_start_code` indicates MPEG-1 sequence.

²Fields followed by * in Figure 3.1 are optional.

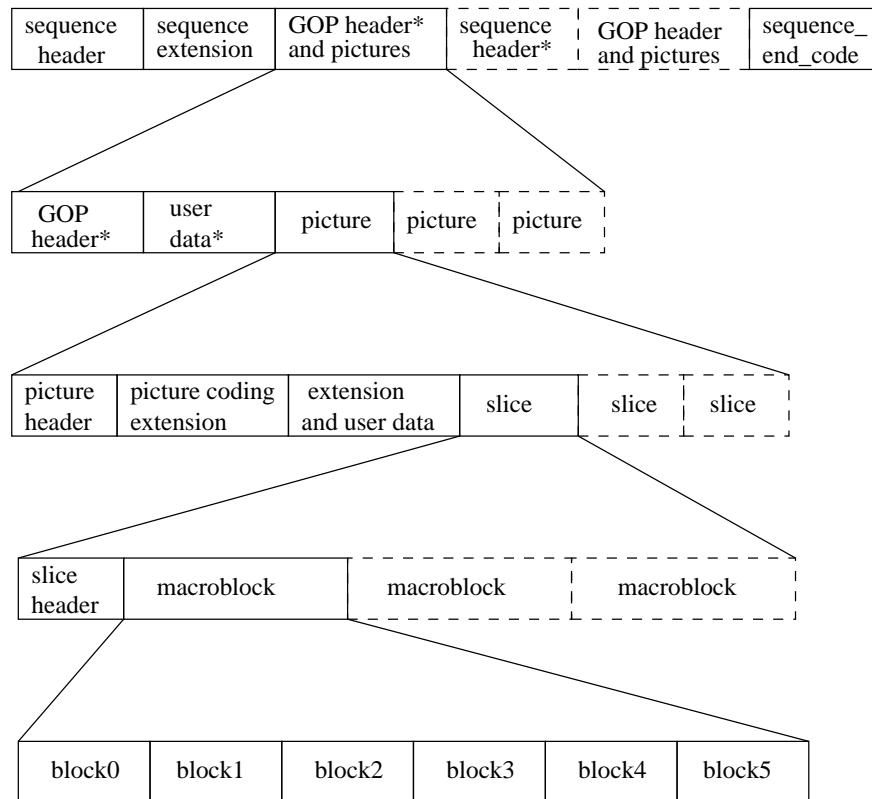


Figure 3.1: Data organization in MPEG-2 video sequence

may be coded. Blocks make the bottom layer of sequence. If it is an intra-coded block, differential DC coefficient is coded first. Rest of the coefficients are coded as runs and levels. *end_of_block* (EOB) terminates the variable length codes for that block.

3.3 MPEG-2 Video Sequence Decoding

3.3.1 Algorithm

Figure 3.2 gives a simplified MPEG-2 video sequence decoding algorithm, skipping all header details. In this figure 'C' like commenting style has been used. So anything between '*' and '*\' is a comment.

```
mpeg-2_decode(){
    get_seq_header();
    get_seq_ext();
    do{
        /* decode GOP */
        get_gop_header();
        next_start_code();
        while(pic_start_code){
            /* decode picture */
            get_pic_header();
            get_pic_ext();
            next_start_code();
            while(slice_start_code){
                /* decode slice */
                get_slice_header();
                do{
                    /* decode macroblock */
                    do{
                        /* decode block */
                        get_block();
                    }while(any block within macroblock is left)
                }while(any macroblock within slice is left)
            }
        }
    }while(end_of_seq)
}
```

Figure 3.2: Simplified representation of MPEG-2 video sequence decoding

It can be seen from the algorithm that MPEG-2 decoding is essentially a nested loop algorithm which gives enough room to exploit parallelism. Quite a few approaches are possible to extract parallelism present in the algorithm. Compilers are there that are able to extract instruction level parallelism at a very fine level of granularity. The drawback is that they are unable to exploit coarse grain parallelism offered by the architecture (presence of coprocessors such as IDCT).

Other approach is to represent the algorithm as a network of concurrently executing processes (Kahn process network). This representation makes parallelism explicit and makes mapping of algorithm onto architecture easier by putting the processes either on microprocessor or its coprocessor. Both automatic and manual construction of process network is possible from sequential specification. In [6] a methodology has been described, which tries to build process network using very aggressive data dependency analysis of *single assignment code*. However, in this project we do not try to explore such approach, rather we try to extract data parallelism using manual analysis at levels such as slice, macroblock or block in MPEG-2 decoder application.

3.3.2 Scope of Parallelism

A number of nested loops can be seen in Figure 3.2. These loops are present at all the levels i.e., at GOP, picture, slice, macroblock and block level. This suggests possibility of extracting parallelism at all levels.

The first picture of a GOP should be an I picture or B picture and the last picture should be an I picture or P picture. It is possible to process two GOPs in parallel provided GOPs are closed³. If the GOP is not closed, data dependency is created between two consecutive GOPs and decoding of GOPs in parallel is not possible. In [5] a parallel model for MPEG-2 decoder has been described assuming closed GOPs in video sequence.

Since P and B pictures depend on other pictures for their decoding, par-

³A GOP is said to be closed when the first picture is either I picture or a B picture which does not depend on pictures in previous GOP.

allelism at picture level is not possible. Within a picture each slice is coded independently. Moreover a slice is identified with `slice_start_code` followed by a header. So slices have proper boundaries. This makes parallelism possible at slice level.

Within a slice each macroblock is independent, but there is no macroblock header to define boundaries of macroblocks. So macroblocks and blocks within a macroblock can be processed in parallel only after variable length decoding. In [9] and [4] parallel models for MPEG-2 decoder have been described exploiting parallelism at slice and macroblock level. We further extend this exercise by incorporating parallelism present at both slice and macroblock level in our MPEG-2 decoder model.

Chapter 4

MPEG-2 Decoder Modeling

Application modeling is the first step in Y-Chart [12]. In this chapter various aspects of application modeling (MPEG-2 decoder in particular) using YAPI has been discussed in detail.

4.1 The Early Structure

The basic tasks required for MPEG-2 video sequence decoding are sequence header extraction, variable length decoding, inverse quantization, inverse DCT, motion compensation (This can be further divided into motion vector decoding and prediction) and frame memory management. Based on these tasks, a parallel model for MPEG-2 decoder was suggested in [15] using YAPI (Figure 4.1). In YAPI each node is a Process and each communication channel is a FIFO . The application model described in this section is our starting point. The MPEG-2 decoder described in Section 4.3 has been derived from this model.

4.1.1 The Configuration

Description of all the processes in Figure 4.1 is as follows:

Process Tinput

The process Tinput has an exclusive access to the MPEG-2 encoded video sequence. This process is responsible for reading the bit stream and passing bit stream to the process Tvld for further processing. No other process has direct access to the bit stream. Other processes get access to bit stream through buffering mechanism.

Process Thdr

The process Thdr is responsible for extracting and interpreting header information from the bit stream. To get access to the bit stream, Thdr handshakes with the process Tvld. Thdr parses bit stream and builds picture and sequence properties. Picture and sequence properties are transferred to different processes for further operations.

Process Tvld

The process Tvld stands for variable length decoding. Responsibilities of the process Tvld are:

- Giving access to bit stream to the process Thdr through handshaking.
- Extracting motion vector, prediction and DCT coefficients properties by parsing the bit stream.
- Variable length decoding of DCT coefficients.

After extraction, motion vector and prediction properties are transferred to the process TdecMV. DCT coefficient properties and variable length decoded coefficients are transferred to process Tisiq. These transfers are at macroblock level.

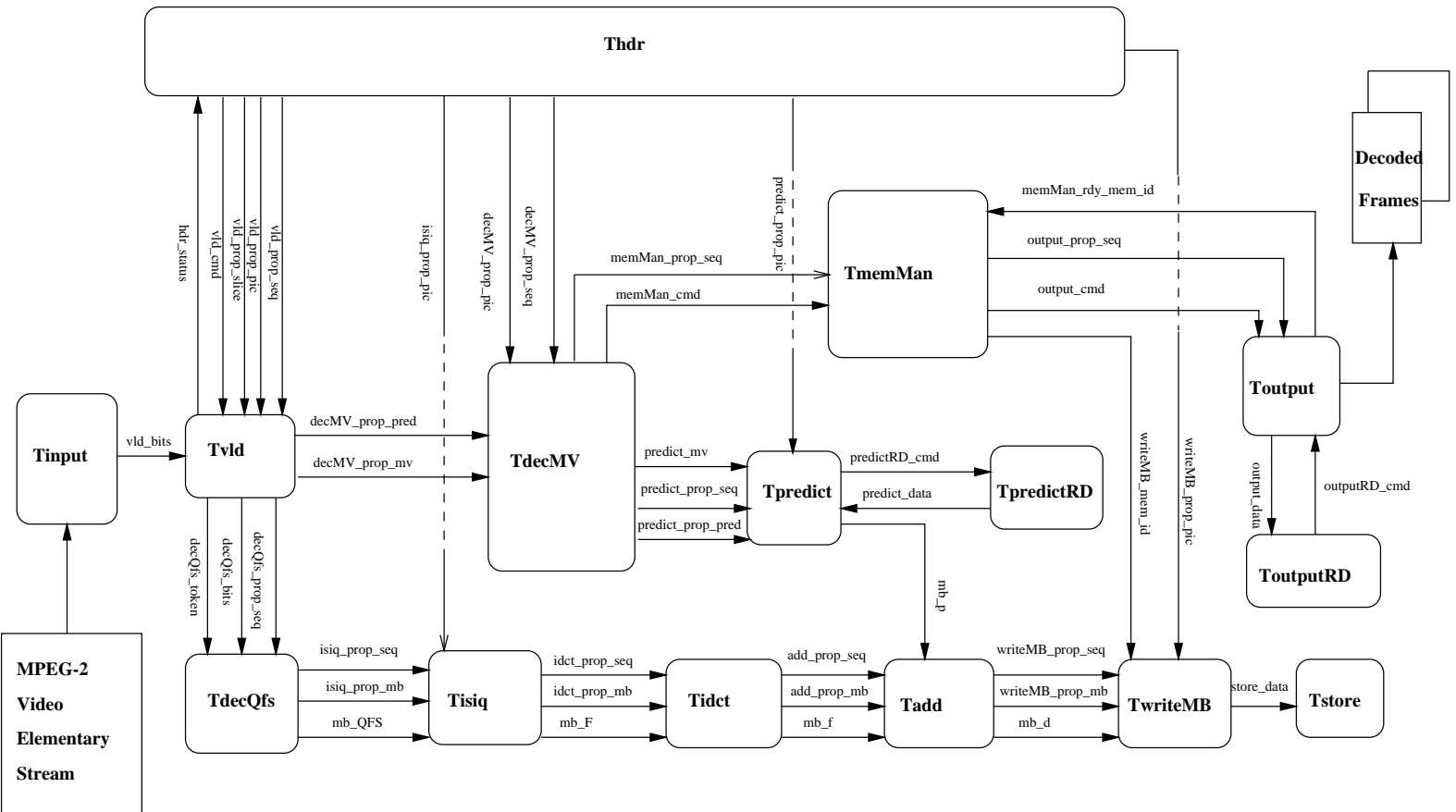


Figure 4.1: A simple MPEG-2 decoder model in YAPI

Process Tisiq

The process Tisiq has following responsibilities:

- Extracting DC¹ DCT coefficients for intra-coded macroblocks
- Inverse scan of AC DCT coefficients
- Inverse quantization of DCT coefficients

After inverse quantization, DCT coefficients are transferred to the process Tidct for IDCT operation.

Process Tidct

The process Tidct is responsible for inverse DCT operation. After the processing of macroblock, it is transferred to the process Tadd for addition with prediction component.

Process TdecMV

The process TdecMV decodes motion vectors. These motion vectors are transferred to the process Tpredict on macroblock basis for prediction.

Process Tpredict

The process Tpredict receives motion vectors from the process TdecMV and provides necessary prediction. Since intra-coded macroblocks are coded independently, so prediction is provided only for non-intra macroblocks. Tpredict receives addresses of reference frames from the process TmemMan.

Process Tadd

The process Tadd simply adds motion compensated component with inverse DCT component and constructs fully decoded macroblock. This macroblock

¹The first coefficient of intra-coded block is DC coefficient. Rest of the coefficients are AC and they are coded using runs and levels.

is transferred to the process TwriteMB which writes them into the frame memory.

Process TwriteMB

The process TwriteMB calculates the position of macroblocks within the frame and writes them into the frame memory. The address of current frame is given by the process TmemMan and calculation of position of macroblock is based on macroblock properties.

As soon as TwriteMB receives a macroblock property token with start_of_pic tag, indicating decoding of new picture, it sends request for current frame address to the process TmemMan. Once TwriteMB receives the address, it can start writing the macroblocks into the frame memory.

Process Toutput

The process Toutput reads the frame from frame memory specified by TmemMan and outputs in some standard format (e.g. PGM format). As soon as Toutput finishes its job, it notifies TmemMan that the address given to it can be reused for other purpose (such as writing new picture macroblocks into it).

Process TmemMan

The process TmemMan keeps track of reference frames, frame which is being currently read by the process Toutput and frame where the process TwriteMB is writing macroblocks. TmemMan ensures that the process TwriteMB writes at the frame address which is neither used as reference frame nor being read by the process Toutput. When TwriteMB requests for frame address, TmemMan searches space for new picture in the frame memory.

Processes Tstore, TpredictRD and ToutputRD are dummy processes. These processes don't contribute towards actual decoding.

4.1.2 Drawbacks of the Model

The model for MPEG-2 decoder described in Figure 4.1 breaks the functionality into several processes and combines them in a pipelined manner. Though the model shows a certain degree of parallelism, it has some drawbacks.

Less Parallelism

The model does not fully exploit the independence present at slice and macroblock level. Though different macroblocks are present at different stages in the pipeline, number of such macroblocks is less at any time.

Bidirectional Communication

There are two processes Thdr and Tvld which require access to the bit stream. Thdr gets access to the bit stream by request and grant mechanism with the process Tvld. This handshaking may be quite expensive in a multiprocessor environment where these two processes might be running on different processors and there is a constant contention for communication resources. Moreover request for number of bits by Thdr at any time is not very high. This leads to more number of requests and more communication cost.

Unbalanced Pipelines

In Figure 4.1 there are two paths emerging from the process Tvld. The upper path corresponds to motion compensation and lower path corresponds to inverse quantization followed by inverse DCT. Experiments show that latency of lower path is higher than upper path. Moreover, since intra-coded frames don't require motion compensation, computation in the upper path is quite less in this case. This latency difference reduces effective parallelism and puts further requirement to reduce latency in the lower path.

Presence of Feedback Loop

Processes `TwriteMB`, `TmemMan`, `Tpredict` and `Tadd` make a small feedback loop. Since the process `TmemMan` has to ensure that `TwriteMB` writes at a valid location, `TmemMan` might have to wait until `Toutput` finishes reading the frame which is being output. This may cause additional delay. The process `Tpredict` does not get addresses of reference frames until `TwriteMB` is granted its request. Since `TwriteMB` cannot proceed, it may cause some processes to get blocked in both upper and lower path. This will increase number of context switches and more delay in decoding. Therefore this feedback loop should be broken and `TwriteMB` should be given the frame address for new frame slightly ahead of time (e.g. as soon as `Tvld` knows that a new picture has arrived).

Single Decoder

The model does not suggest how multiple MPEG-2 decoders can be run in a bigger application set up.

4.2 Modeling Issues

Discussion in the previous section highlights various issues. Based on these issues different models can be evaluated and optimized for better performance.

4.2.1 Amount of Parallelism

The amount of parallelism present in the application depends on application itself. Thorough data dependency analysis of application is required to fully extract the data and functional parallelism present in the application. For example, in MPEG-2 decoder application, several independent tasks such as variable length decoding, motion compensation, IDCT etc. can be identified and data parallelism present at slice, macroblock or block level can be exploited by providing more instances of these tasks. It is very likely that an

application model, having more parallelism, will improve performance, provided total number of tokens communicated among processes do not increase much.

4.2.2 Unidirectional Communication

An application model where processes are handshaking with each other (request & grant of tokens between two processes), may lead to a deadlock. Handshaking also increases number of tokens transferred and violates the streaming behavior, where data flows in one direction.

An application model, which has unidirectional communication among processes, is likely to give better performance. This improvement comes from two sources, reduction in number of context switches and number of tokens being communicated. Since data flows only in forward direction in unidirectional communication, it also reduces synchronization overhead and debugging time.

4.2.3 Communication Overhead

In a multiprocessor configuration, there are conflicts for communication resources. As the number of processors increases in the architecture, communication becomes more expensive. So an application model, optimized for reduced communication, will give better speed. This can be achieved by carefully choosing token structure and controlling the number of tokens transferred over communication channels.

4.2.4 Granularity of Operation

Granularity of operation significantly affects structure of application model. Compilers are there which are able to extract instruction level parallelism at a very fine level of granularity. Here we mainly concentrate on coarse grained parallelism.

In signal processing applications parallelizing at finer granularity may have the following effects:

- Data parallelism present at finer granularity can be exploited.
- Communication primitives are called more frequently, as less number of tokens are transferred during any transfer. So communication workload increases.
- It is more likely that total amount of data transfer also goes up.

Hence, at finer granularity, data parallelism can be increased, but it also increases communication workload. So performance improves only when computation workload relatively decreases.

4.2.5 Balanced Pipelines

In a process network, processes communicate with each other in a pipelined manner. The slowest process in the pipeline determines its throughput. If the computation is not properly balanced among processes, it will reduce effective parallelism present within the application model.

Figure 4.2 shows two pipelines which converge at process *Sink*. If the latency of lower path is more, because of finite space in FIFOs, processes in the upper path will get blocked more often. This will increase the number of context switches and reduce the effective parallelism in the application. Hence, in such a situation, an application model will perform better which has balanced pipelines.

4.2.6 Synchronization Overhead

In YAPI an application is modeled as a process network. Processes communicate with each other over FIFOs by transferring tokens. Then it becomes important at what place a process sends or receives tokens. Further, there is need for mechanisms using which global events can be notified and actions

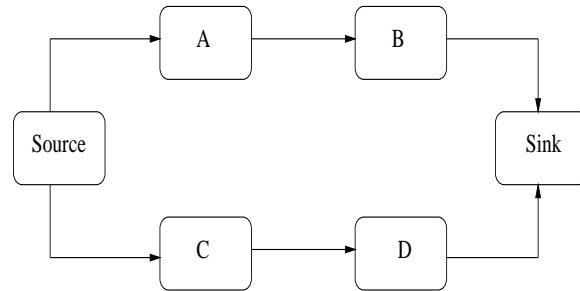


Figure 4.2: A process network demonstrating two pipelines

can be started. For example, arrival of new picture in MPEG-2 decoding is a kind of global event. All the processes within MPEG-2 decoder model should be notified about this, so that they can update picture properties required for correct decoding. So, the application programmer has to take care of the following:

- Identification of the points at which, a token required for synchronization should be sent.
- Mechanism of notifying processes about global events. One solution is to employ *broadcast mechanism*, in which case some special tokens are communicated to all the relevant processes.
- Mechanism of communicating (sourcing or sinking) tokens with the processes repeating same operation (e.g., IDCT).

4.2.7 Multiple Instance of Processes

The process network in Figure 4.3 has n instances of process P . In this process network, throughput at this stage of process P should increase roughly by a factor 'n'. However, this scheme increases synchronization overhead. Now the process *Source* must adopt some policy to distribute the tokens properly among processes P_i and similar policy must be adopted by the process *Sink* to collect tokens.

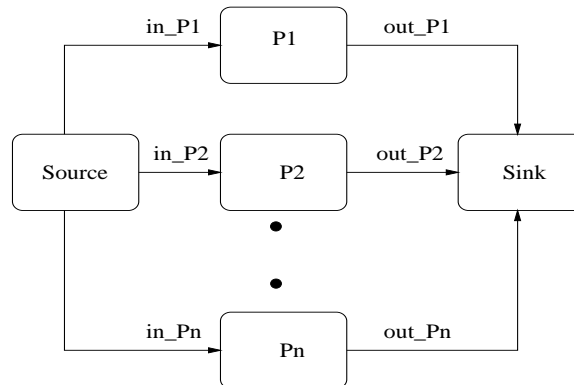


Figure 4.3: A process network demonstrating multiple instance of process P

If 'n' becomes large, processes *Source* and *Sink* may not respond to demands of processes P_i quickly, which may further lead to increase in context switching. So for a large value of 'n' performance improvement may get saturated.

4.2.8 Memory Usage

On chip memory speeds up system performance, but it is limited in amount. This suggests memory usage should be optimized. Increase in parallelism increases number of processes and communication channels within the application model. This further increases memory requirements. So there is a trade off between size of parallelism and space requirements. Though this cannot be eliminated fully, the problem can be reduced by fine tuning the sizes of FIFOs after few experiments.

4.2.9 Scalability

Scalability of the model defines its capability to grow. An application model is scalable if more data parallelism can be provided by just plugging more instances of processes (e.g. by providing more IDCT processes in MPEG-2 decoder application).

4.3 Refined MPEG-2 Decoder Model

We created a new MPEG-2 decoder model using YAPI starting from the model described in Figure 4.1. We have tried to optimize the model based on modeling issues discussed in Section 4.2.

4.3.1 Top level view

Figure 4.4 gives the overview of complete application.

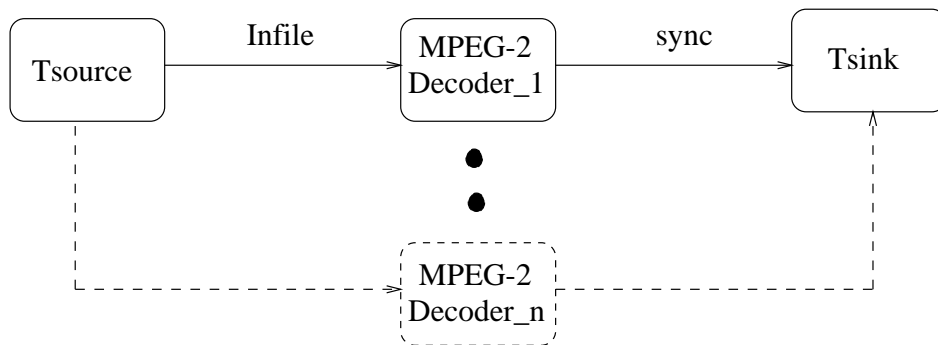


Figure 4.4: Top level view of complete application

Process Tsource

The process Tsource has been put so that multiple MPEG-2 video sequences can be decoded in parallel. Tsource reads initialization file for locations of MPEG-2 video sequences and transfers this information to corresponding MPEG-2 decoder.

Process Tsink

The whole application should terminate only when all the MPEG-2 decoders have finished their jobs. The process Tsink collects sync token from all the MPEG-2 decoders. After collecting all sync tokens, Tsink gives profile information for the whole application and terminates it.

4.3.2 Process Network MPEG-2 Decoder

Figure 4.5 gives the configuration of MPEG-2 decoder. In this configuration, a number of instances of the process network *TsliceDec* can be created. By instantiating more *TsliceDec*, parallelism in application model can be increased. Furthermore, *TsliceDec* can also be made more parallel by instantiating more process networks *Tiq_idct_add* within it.

Process Tinput

The process *Tinput* receives video sequence location from the process *Tsource*. Then *Tinput* reads the sequence, buffers the bit stream and transfers bit buffer to the process *Thdr*. Thus only process *Tinput* has direct access to the bit stream.

Process Thdr

The process *Thdr* has following responsibilities:

- Parses the bit stream and extracts sequence and GOP header information.
- Extracts slices including slice headers and transfers them to process networks *TsliceDecs* on round robin basis (i.g., If there are only two *TsliceDecs*, first slice is given to *TsliceDec1*, second to *TsliceDec2* and third is given back to *TsliceDec1*).
- Sends *start_of_picture* command to the Process *TmemMan* and thus, activates *TmemMan* to determine frame address for new picture.

Process TmemMan

The process *TmemMan* gets *start_of_picture* command from the process *Thdr*. Then it sends reference frame addresses and frame address, where the next frame is to be written after decoding, to the process networks *TsliceDecs*.

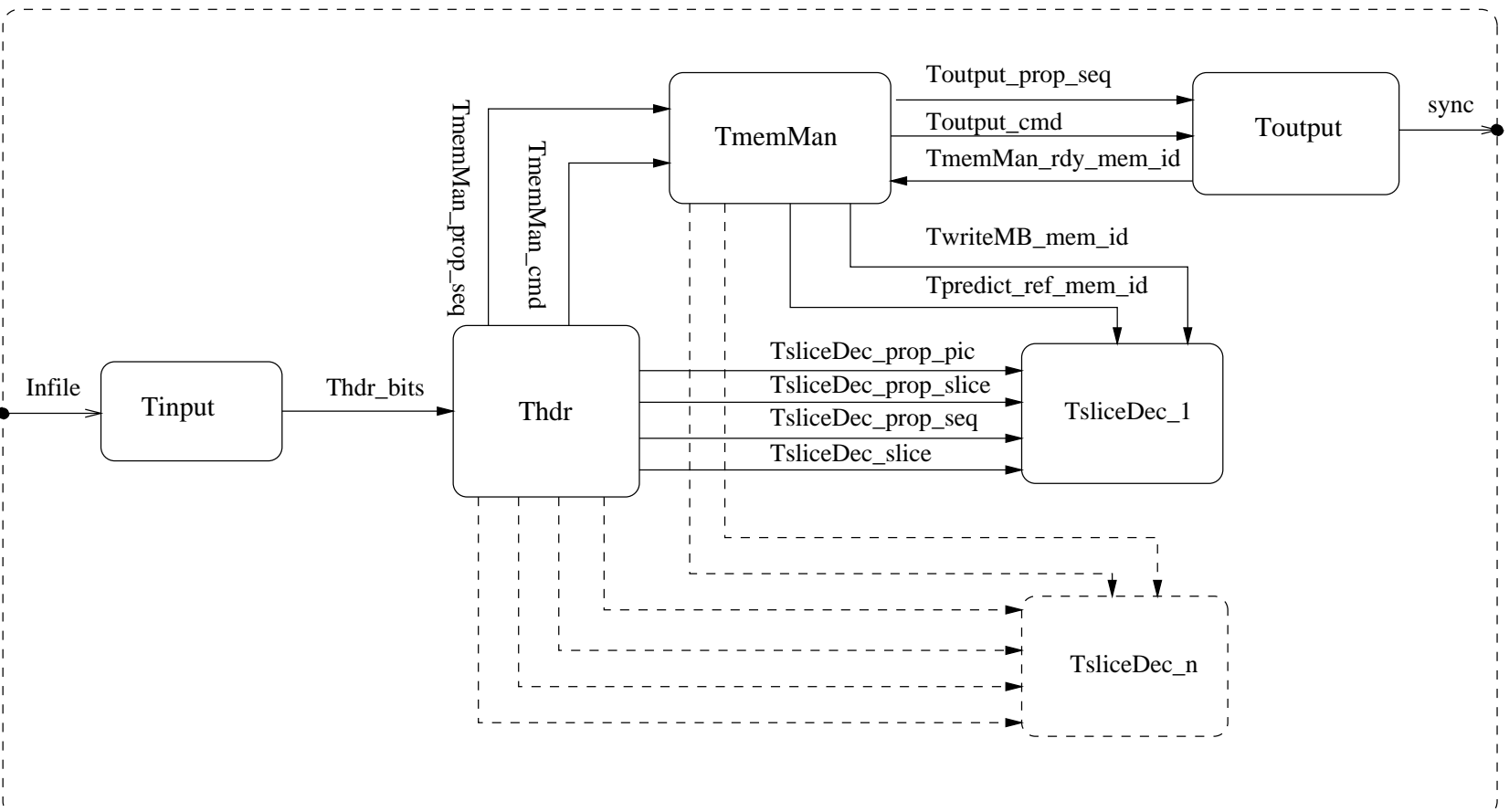


Figure 4.5: MPEG-2 decoder model

In present configuration, TmemMan determines the frame addresses much ahead of the time, when decoded macroblocks can actually be written into the frame memory. This might create situation when the Process Toutput is reading and one of the the process networks *TsliceDec* is still writing decoded macroblocks at the same location. To avoid this situation, TmemMan does not give the address of current frame being decoded to the process Toutput at the start of next frame, rather it waits for one more frame to arrive. Here TmemMan assumes that this two frame interval will give enough time to the process network *TsliceDec* for decoding of at least first frame.

Let us assume that the sequence of decoded frame is {f0, f1, f2, ...}. In this scheme, address of f0 is given to the process Toutput only when the frame f2 has already arrived before decoding. This time interval is sufficient for processing of frame f0 at all *TsliceDec*. Though this scheme introduces delay of one frame for sending frame address to Toutput in the beginning, afterwards frames are displayed at regular intervals.

This configuration removes request-grant mechanism present in Figure 4.1 for frame addresses. This not only speeds up decoding, but also facilitates instantiation of multiple process networks *TsliceDec* easier, by making MPEG-2 decoder process network simpler.

Process Toutput

This process receives address of frame to be displayed from TmemMan. It sends sync token to the Process Tsink as soon as it receives *end_of_seq* token. Furthermore this process notifies process TmemMan about end of reading the frame by sending a token over FIFO *TmemMan_rdy_mem_id*.

Process Network TsliceDec

The process network TsliceDec gets slices (This includes slice header) and decodes it. Its structure is described in Subsection 4.3.3. The purpose of creating this process network is to exploit parallelism at slice level.

All the process networks *TsliceDec* need properties of the picture for correct decoding. In the configuration where several *TsliceDecs* are present, it becomes necessary to notify all *TsliceDecs* that new picture has arrived. To implement this, a *broadcast mechanism* has been adopted. Special tokens are transferred, for new picture, over all the FIFOs corresponding to slice property or macroblock property (FIFO *TsliceDec_prop_slice* in Figure 4.6 and FIFOs with suffix *prop_mb* in Figure 4.6 and Figure 4.8). Thus creating multiple instances of processes or process networks may impose new synchronization requirements.

4.3.3 Process Network *TsliceDec*

Figure 4.6 describes the process network *TsliceDec*. *TsliceDec* receives slice (including slice header), extracts macroblocks and writes them into the frame memory. Within *TsliceDec*, it tries to exploit parallelism present at macroblock level. At this level, model has flexibility to increase number of instances of the process network *Tiq_idct_add* to enhance parallelism present in the model.

Process *Tvld*

Responsibilities of the process *Tvld* include:

- Extraction of slice header
- Extraction of motion vector and macroblock properties²
- Variable length decoding of macroblock coefficients (this includes differential DC coefficient of intra coded blocks) and transfer to process networks *Tiq_idct_add*. This transfer is on *round robin basis*.

²Motion vector properties include fields such as *motion vector format*, *motion code*, *dual motion vector* etc. and macroblock properties include fields such as *DCT type*, *quantization scale code*, *intra dc precision* etc.

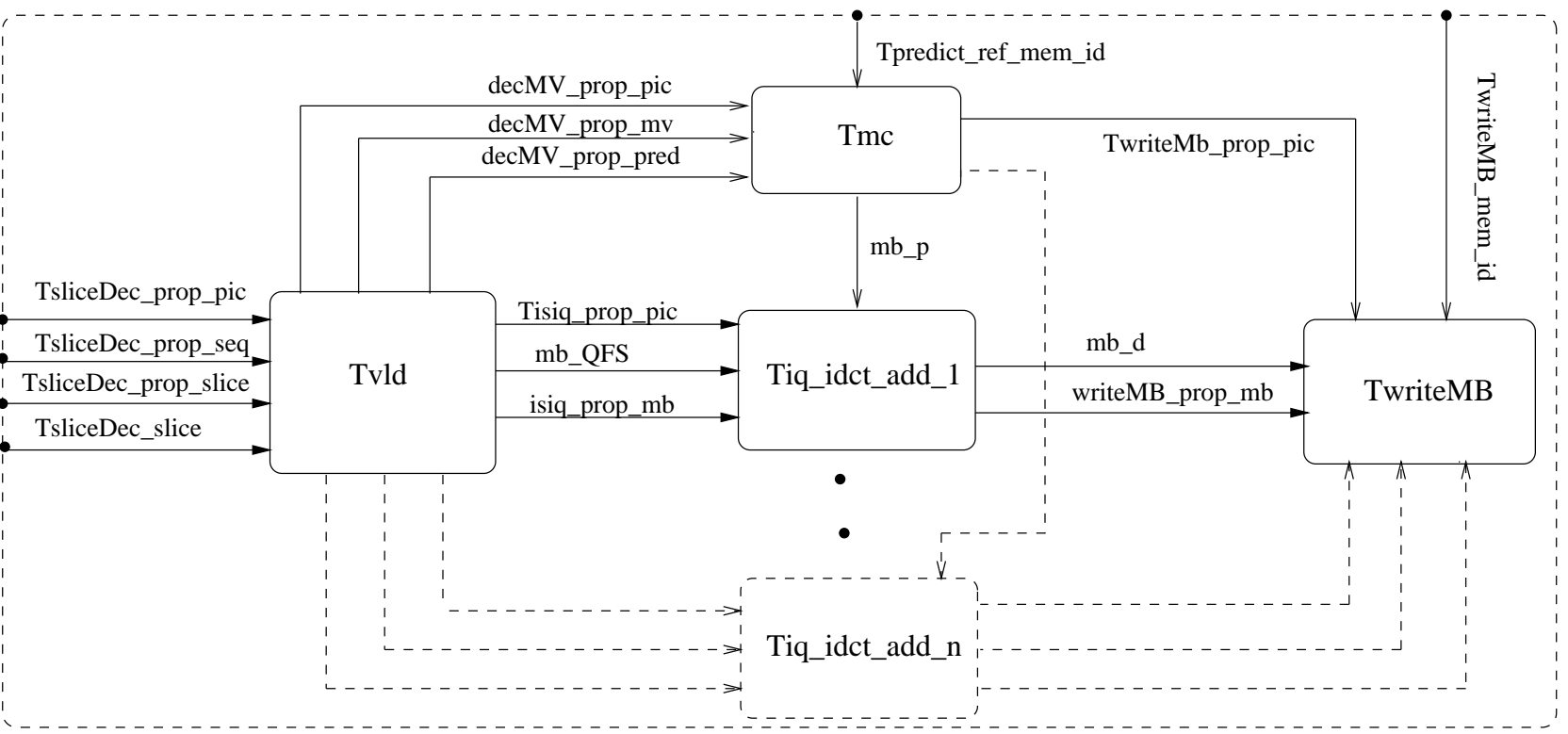


Figure 4.6: Process Network $T_{sliceDec}$

To reduce communication overhead, only non zero DCT coefficients are transferred to process networks *Tiq_idct_add*.

Process Network *Tiq_idct_add*

The activities of the process network *Tiq_idct_add* include inverse quantization, inverse DCT and addition with prediction blocks. Its structure has been described in subsection 4.3.5.

Process Network *Tmc*

The process network *Tmc* receives motion vector properties from *Tvld* and provides motion compensation. Subsection 4.3.4 discusses more about *Tmc*.

TwriteMB

The process *TwriteMB* receives decoded macroblocks from process networks *Tiq_idct_add* and picture properties from *Tmc*. It adopts the same *round robin policy*, as adopted by *Tvld*, to communicate with process networks *Tiq_idct_add*. Now there is no request to *TmemMan* for frame addresses.

4.3.4 Process Network *Tmc*

The process network *Tmc* (Figure 4.7) provides motion compensation.

Process *TdecMV*

The process *TdecMV* is responsible for motion vector decoding.

Process *Tpredict*

The process *Tpredict* gets motion vectors from *TdecMV* and provides prediction. It receives information, regarding which *Tiq_idct_add* process network should be selected to write predicted blocks, from *TdecMV*, which in turn receives that information from *Tvld*.

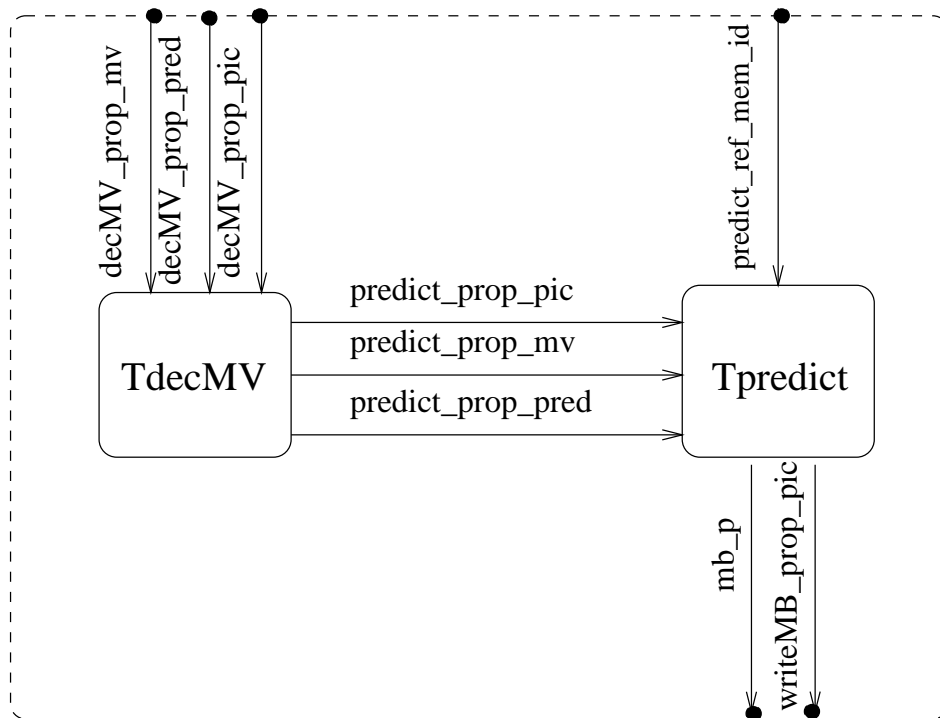


Figure 4.7: Process Network Tmc

4.3.5 Process Network Tiq_idct_add

Figure 4.8 describes the structure of the process network Tiq_idct_add. Since the process Tidct (this process performs IDCT) is quite computation intensive, multiple such processes are provided to increase the speed. The number of Tidct processes, present in the process network, is flexible.

Process Tisiq

Inverse scan and inverse quantization is performed by the process Tisiq. This process has two additional features:

- This process communicates (transfer of macroblocks) with a number of Tidct processes using round robin policy.
- DCT coefficient matrix of a block is sparse in nature. Since a number of entries are zero, only non-zero coefficients are transferred to the process

Tidct. This transfer takes place after compression of blocks using simple sparse matrix representation. Each non-zero entry maintains two fields, *value* and *index*. Later, process Tidct does decompression and extracts the macroblock.

The mode of macroblock transfer increases computation overhead for compression and decompression, but it gains more by reducing communication overhead.

Process Tidct

The process Tidct performs inverse DCT operation.

Process Tadd

The process provides necessary addition after motion compensation. It receives macroblocks from all the Tidct processes using round robin scheme, same as the process Tisiq.

4.4 Salient Features of The Model

- The process network MPEG-2 decoder is identified by *MPEG_ijk*. Number of slice decoders (process network TsliceDec) is indicated by 'i', 'j' indicates number of Tiq_idct_add process networks within the process network TsliceDec and 'k' indicates number of Tidct processes within process network Tiq_idct_add. Different configurations can be created by instantiating MPEG_ijk corresponding to different 'i', 'j' and 'k' combinations.
- In the MPEG-2 decoder process network parallelism has been extracted at slice level and within the process network TsliceDec, parallelism at macroblock level has been exploited.

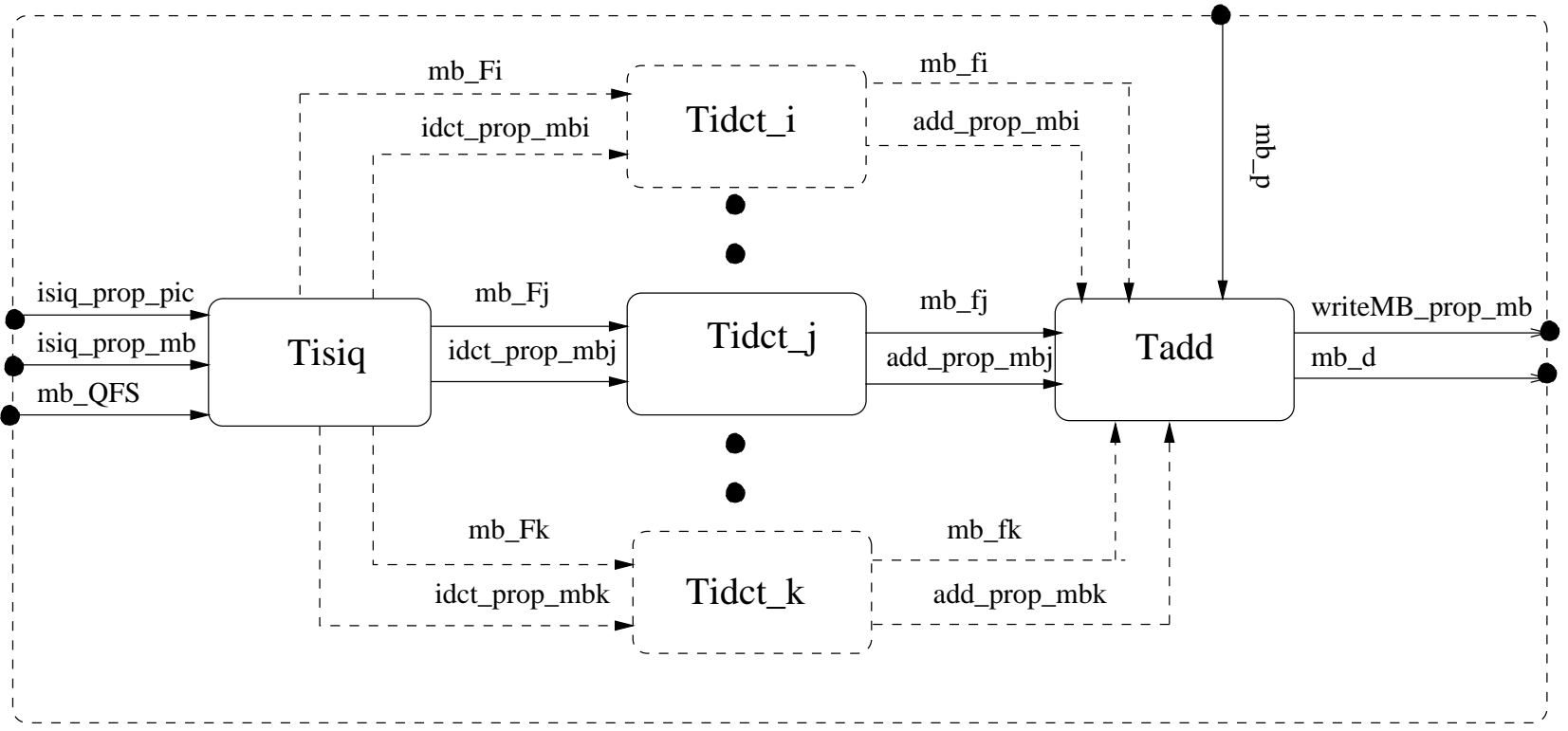


Figure 4.8: Process Network $T_{iq_idct_add}$

Chapter 5

Experimental Environment

5.1 Introduction

The simulation setup is composed of 4 layers (Figure 5.1). Uppermost layer is application. Application has been modeled using primitives of YAPI (Chapter 2 discusses YAPI in detail). YAPI further uses thread scheduler to provide its services. Thread scheduler is responsible for thread management activities such as context switches, maintaining status of threads etc. The bottom layer is architecture which runs under control of thread scheduler.

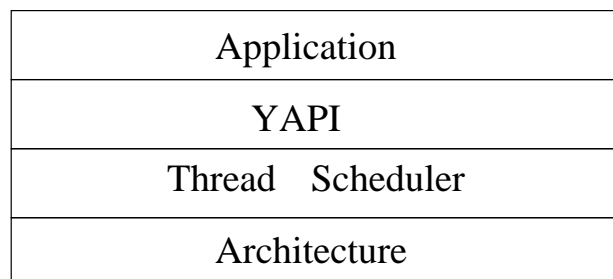


Figure 5.1: Layers of experimental environment

5.2 Thread Scheduler

Figure 5.2 gives an overview of thread management.

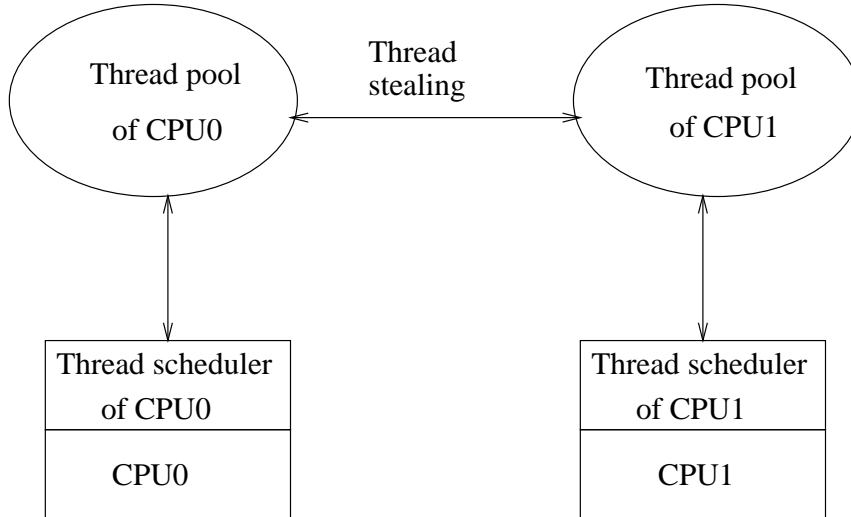


Figure 5.2: Thread scheduling

Activities of thread scheduler are as follows:

- Creating threads and maintaining thread pool for each processor.
- Facilitating context switches. At the time of initialization, all the threads go to the thread pool of one processor. When thread scheduler of any processor looks at its thread pool for a thread to schedule and its thread pool is empty, it tries to steal a thread from thread pool of other processor. This thread stealing mechanism enables all the processors to build their thread pool after initialization.
- Maintaining the status of threads.
- Implementation of semaphore operation.

Every CPU has its own thread scheduler. The thread scheduler is like a small operating system, which manages the resources of the CPU.

5.3 The SpaceCAKE Architecture

The SpaceCAKE Architecture is a homogeneous multiprocessor architecture. This architecture addresses re-usability and scaling and targets very computation intensive applications such as video processing.

5.3.1 The Architecture

The SpaceCAKE Architecture (Figure 5.3) is essentially an homogeneous multiprocessor architecture. The basic unit of repetition is a tile. A tile consists of a heterogeneous mix of memories, general purpose processors (like the MIPS 1900 or ARM), DSPs, configurable function units and high-performance special-purpose functional units. All tiles on a chip are exactly the same and they are laid-out in a 2-dimensional fashion with a high-speed message passing network to interconnect them. The interface of the tile and interconnection network are further topic of research.

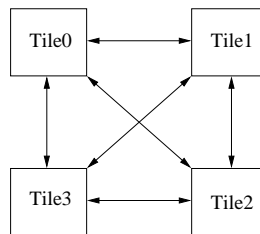


Figure 5.3: The SpaceCAKE Architecture

5.3.2 The Tile

Tile is the basic unit of repetition in SpaceCAKE Architecture. Figure 5.4 shows the present structure of tile. It essentially consists of a number of general purpose processors, memory bank, a snooping bus, a BCU (bus control unit) and a proxy (to implement system calls).

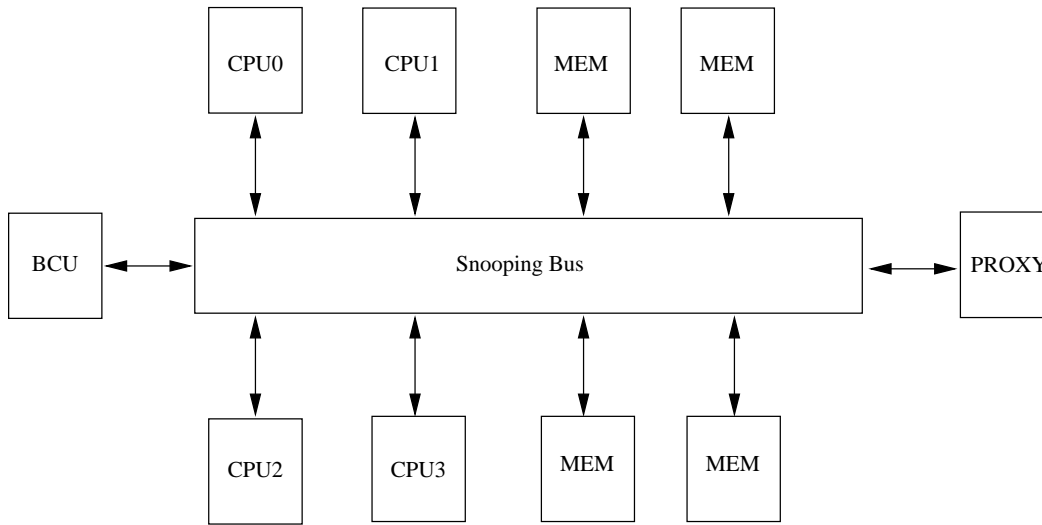


Figure 5.4: Structure of tile

The CPU

The CPU present in the tile is a MIPS-32 micro-controller, which is close to PR1910 [17]. Apart from supporting normal PR1910 features, it also supports MSI¹ cache coherence protocol [8], so that it can be used as node in a cache coherent multiprocessor system. Moreover this also supports upto two bus adapters that can be active concurrently. Bus adapters either connect to a 32-bit PI-bus or to a 64-bit DVP-DMA bus.

Bus

The snooping bus shown in Figure 5.4 is Peripheral Interconnect (PI) Bus [19]. Apart from supporting clock synchronous and multi-master bus operation, it also supports memory coherency. Support for multi master bus operation and memory coherency enables it to be used in a multiprocessor configuration.

¹MSI stands for Modified Shared Invalid

Bus Control Unit

Bus Control Unit (BCU) controls the operation of bus. Functions are:

- Bus ownership assignment and address dependent slave selection
- Initiation of memory coherency protocol
- Unification of acknowledgment control information
- Detection of illegal addresses and time-out bus faults

PROXY

This component of tile handles system calls. Whenever any CPU has any system call, CPU sends request to PROXY to implement that. If there is only one request, the request is granted immediately. If more CPUs are sending requests, PROXY selects one CPU at a time and grants its request.

Memory

This component of tile refers to storage available within the tile besides cache memory inside the CPUs.

The present configuration of tile shows only MIPS CPUs, the configuration can be extended later to accommodate other components also such as DSPs, coprocessors etc.

Chapter 6

Experiments and Results

Motivation behind doing experiments, using YAPI application model, is to fine tune the application model to increase its speed and to collect performance data to explore the architecture design space. Experiments were done at both application level (YAPI provides information about computation overhead and communication overhead of the application) and system level using TSS. Section 6.1 analyzes application level and Section 6.2 analyzes system level performance data.

6.1 YAPI Level Simulations

Computation and communication overhead is obtained from YAPI, when the application is run on a workstation in a multi-threaded environment. The data referred in this section are obtained by running application on *sun* platform (SunOS 5.7) and input MPEG-2 video sequence is *tennis.m2v* (8 frames, frame size 576x704).

6.1.1 Computation Overhead

Threads are created corresponding to each process in the application model. Computation overhead refers to the total time taken by the whole application (decoding of video sequence *tennis.m2v*) and time distribution among

processes (threads). This distribution refers to the total time taken by any process to perform computation on data, which it gets during decoding of whole sequence (*tennis.m2v*, 8 frames, frame size 576x704). This information can be used in the following way:

- Different models can be compared for speed up.
- Percentage of time taken by different processes can be used to find out which process takes more time and is more computation intensive. Moreover timing information enables to find out which pipeline has more latency and can be a source of bottleneck.

Table 6.1 gives time distribution of processes for model described in Figure 4.1. This model has two parallel paths emerging from the process Tvld. First corresponds to motion compensation and second corresponds to inverse DCT. They finally merge at process Tadd. From the table it can be seen that second path takes more time to complete its task. The first path takes total 18.57%, whereas second path takes total 27.89% time. So, to speed up the application, second path should be made faster (by providing more processes).

YAPI provides more information about application by giving *Number of context switches* (total number of switches among threads on the CPU) and *Parallelism number*.

Number of context switches This is the total number of thread switching on the processor. A context switch takes place when a process gets blocked. So this indicates how often processes get blocked.

Parallelism number This is the average number of processes (threads) in the ready list of the processor. This gives an indication of effective parallelism present in the model. If N_{sw} indicates Number of context switches and N_{thr}_i indicates number of threads in the ready list of the processor at i^{th} instance of context switch, *Parallelism number* is calculated as follows:

Name of process	Time taken	% Computation time
ToutputRD	0.114250s	1.48
Toutput	1.074277s	13.9
TmemMan	0.031361s	0.4
TpredictRD	0.409410s	5.2
Tpredict	1.293022s	16.8
TdecMV	0.136052s	1.77
Tstore	0.058771s	0.65
TwriteMB	0.610476s	7.9
Tadd	1.045679s	13.5
Tidct	1.276872s	16.6
Tisiq	0.867145s	11.29
Tvld	0.708408s	9.22
Thdr	0.013470s	0.17
Tinput	0.041562s	0.53

Table 6.1: Computation workload

$$Parallelism\ number = \frac{\sum_{i=1}^{N_{sw}} Nthr_i}{N_{sw}} \quad (6.1)$$

Total time This is the total time taken by the application when run on multi-threaded workstation given the input video sequence (tennis.m2v). This is not a very accurate measurement of speed of application, but it gives feedback about relative speed up. More accuracy is obtained when application is run in TSS environment (TSS is a cycle accurate simulator).

Table 6.2 shows that *Number of context switching* comes down by increasing the value of 'i' in models *mpeg_ijk*, but increases when j goes from 2 to 4. This implies that it is quite possible that after a certain point by providing more instances of processes or process networks will increase *Number of context switches* even if *Parallelism number* increases.

Model name	Number of context switches	Total time taken (sec)	Parallelism number
old_model	346763	7.68	3.87
mpeg_122	21786	5.78	7.81
mpeg_222	20506	5.71	13.58
mpeg_242	31826	6.04	22.89
mpeg_422	19919	5.77	23.87
mpeg_442	31965	6.02	39.6

Table 6.2: Comparison among various application models

6.1.2 Communication Overhead

Communication overhead refers to total number of tokens transferred and hence, total amount of data transfer over FIFOs. This information can be used to change the sizes of FIFOs to reduce memory usage and to reduce number of times processes get blocked. For example, size of FIFO, which transfers less number of tokens, can be decreased and size of FIFO, which transfers more number of tokens, can be increased. This resizing should reduce the number of situations, when a process gets blocked on write. Furthermore, resizing of FIFOs can increase memory used by the application, so there is a trade off between space and speed.

6.2 TSS Level Simulations

The application is mapped onto the architecture to get architecture performance data. Different YAPI models of MPEG-2 decoder were mapped onto the tile (Figure 5.4) of SpaceCAKE architecture (Figure 5.3). Since, at present, tile has a number of MIPS CPUs, we compiled different configurations of MPEG-2 decoder for MIPS and obtained executables. Then this is run on TSS [18] (Tool for System Simulation) model of architecture (single tile within SpaceCAKE architecture). We used *tennis.m2v* (8 frames, frame

size 576x704) as MPEG-2 video sequence. We simulated for three different *tile* (Figure 5.4) configuration, with 2, 4 and 8 CPUs.

6.2.1 Application Behavior

Number of Cycles

Figure 6.1 shows simulation results for total number of cycles, taken by the MPEG-2 decoder model to decode the video sequence *tennis.m2v*. The plot shows normalized values at Y-axis. These are normalized against total number of cycles during simulation of *old_model* (Figure 4.1) with 8 CPUs.

$$\text{Normalized number of cycles} = \frac{\text{cycles}_{\text{model_name}}}{\text{cycles}_{\text{old_model_8cpus}}} \times 100 \quad (6.2)$$

Except two models (*old_model* and *mpeg_122*¹) other models show decrease in number of cycles with increasing number of processors in architecture. The possible reasons why *old_model* and *mpeg_122* models show deviation, could be:

1. Less parallelism
2. More communication cost. Processors wait for data for significant amount of time.

The gain in speed, while going from 4 CPUs to 8 CPUs, is less compared to gain from 2 CPUs to 4 CPUs. The reason is more conflict to get access on communication resources (bus). This is verified by increase in *bus_wait_cycle* (Figure 6.3). Among shown application configurations model *mpeg_222* is the fastest.

¹The application model *mpeg_ijk* indicates that there are 'i' *TsliceDecs*, 'j' *Tiq_idct_add* process network within *TsliceDec* and 'k' *Tidct* processes within *Tiq_idct_add* process network.

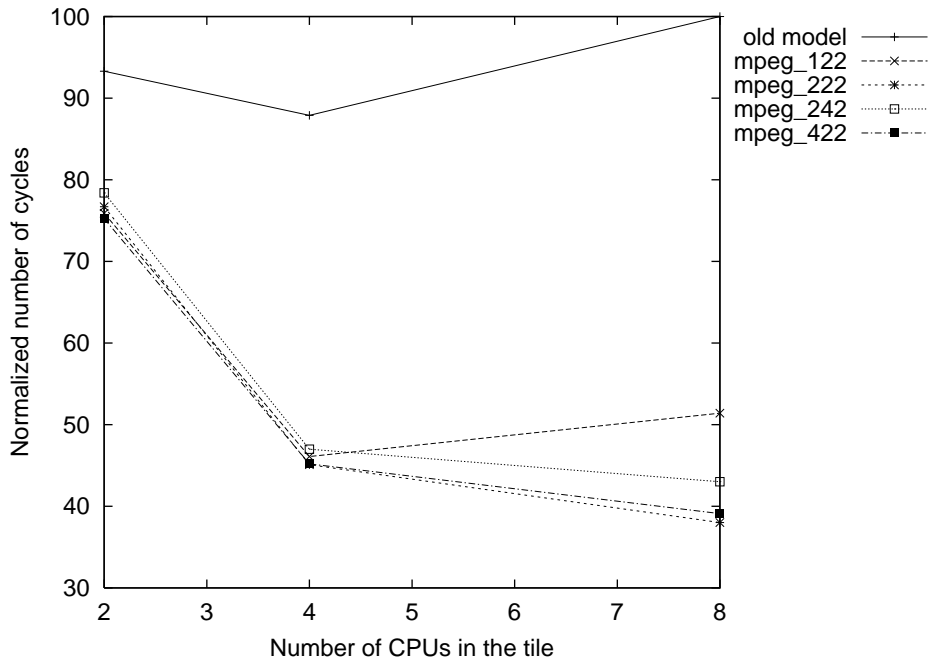


Figure 6.1: Number of cycles vs Number of CPUs

Cycles per Instruction (CPI)

CPI^2 is the ratio of total number of cycles over total number of instructions, taken by the application to decode *tennis.m2v* during TSS simulation. In Figure 6.2, except old_model all other models show increase in CPI, which indicates decrease in total number of instructions relative to total number of cycles.

Bus Wait Cycles

Figure 6.3 shows variation of bus_wait_cycles for different configurations. In this plot, bus_wait_cycles indicates the average number of cycles a CPU is blocked while a cache fill request is in progress. Except old_model all other models show the same kind of behavior and variation is not very high. The curve is nearly straight line. So increase in bus_wait_cycles is almost 9 units

²This is the average CPI value over all the processors.

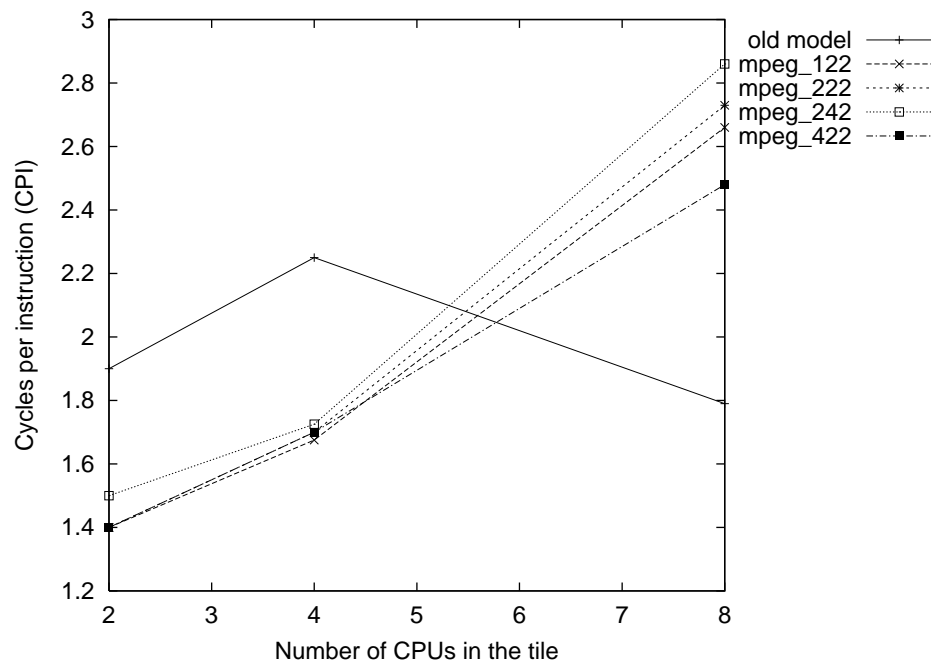


Figure 6.2: CPI vs Number of CPUs

per processor, whereas `old_model` show slightly different behavior and increase in `bus_wait_cycles` is 6.4 units per processor. The reason for increase in `bus_wait_cycles` is that now there are more CPUs to share the only communication resource (bus).

Cache Coherence

Figure 6.4 shows the total number of snooping requests³ made by CPUs for cache coherence. The TSS model of the CPU uses MSI cache coherence protocol for cache coherence among CPUs. It is clear from the plot that as the number of CPUs is increased, required snooping requests also go up. The curves are again nearly straight lines. Increase in snooping request is approximately 3.83 units per CPU for `old_model` and 1.1 units per CPU for others. Hence relative decrease, in number of snooping requests in new models compared to `old_model`, is by a factor of 3.4, which indicates there is

³This is the average number of snooping requests made by all the CPUs

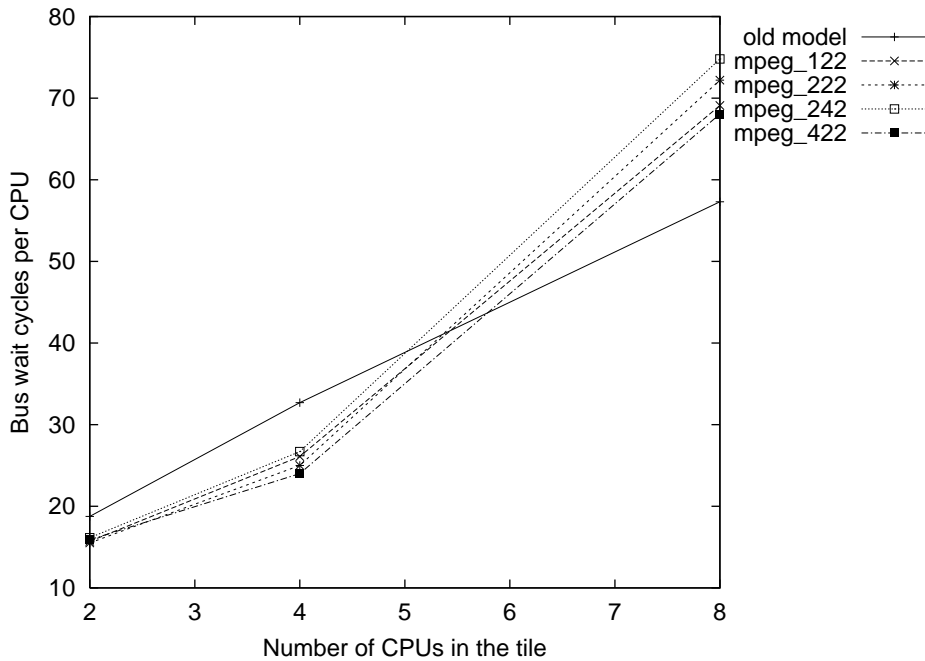


Figure 6.3: Bus wait cycles vs Number of CPUs

relatively much less cache coherence activity compared to `old_model`.

Number of coherence writes can be seen in Figure 6.5. As expected, this increases by increasing number of CPUs in the tile, as with more CPUs it is more likely that two communicating processes are running on different CPUs, necessitating coherence write traffic from the producer to the consumer. Further, *Number of coherence writes* reduces drastically for our MPEG-2 decoder model because of lesser communication overheads. However, by increasing parallelism in *mpeg_ijk* models, coherence writes are reduced but not much. This indicates that if the number of tokens transferred does not increase much, increasing parallelism helps to reduce cache coherence writes.

Parallelism Number

In Figure 6.6 *Parallelism number* have been taken from YAPI level simulations. It can be seen that variation in the total number of cycles taken during the TSS simulations is not large for a particular number of CPUs. This im-

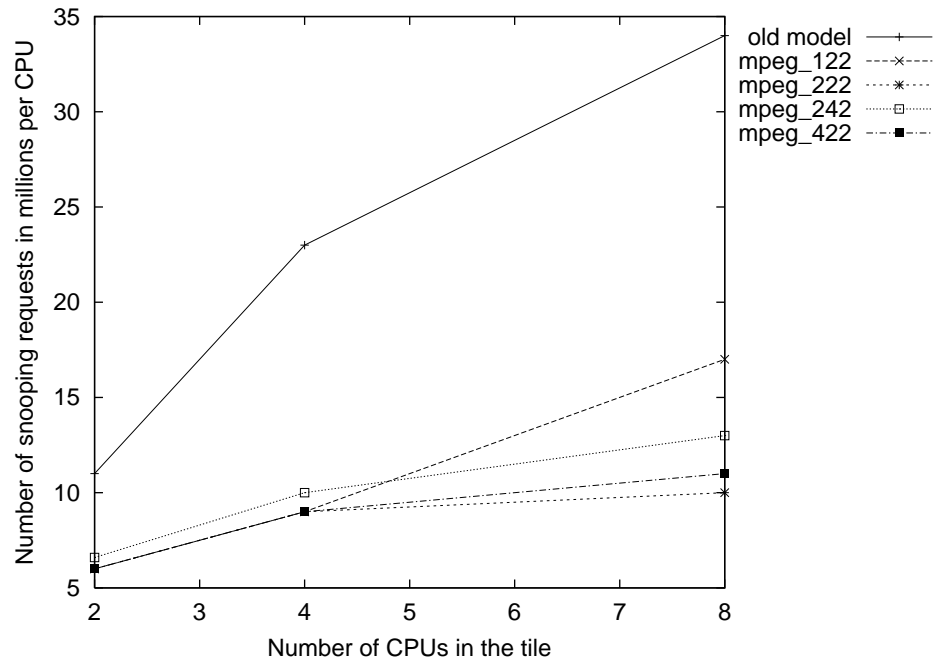


Figure 6.4: Number of snooping requests vs Number of CPUs

plies that ideally total number of cycles taken during TSS simulations should decrease as *Parallelism number* increases, but the communication bottleneck prevents this.

6.2.2 Architecture Behavior

Figure 6.7 shows variation of architectural parameters for application mpeg_222 model when run on 8 CPU tile configuration given input video sequence *tennis.m2v*. CPI is almost same for first few CPUs and then rises in a linear manner. Number of snooping operations performed by all the CPUs are nearly equal. The most surprising variation is seen in number of bus wait cycles. The curve indicates that few of the CPUs seem to be waiting much more than other CPUs. Further this variation is linear.

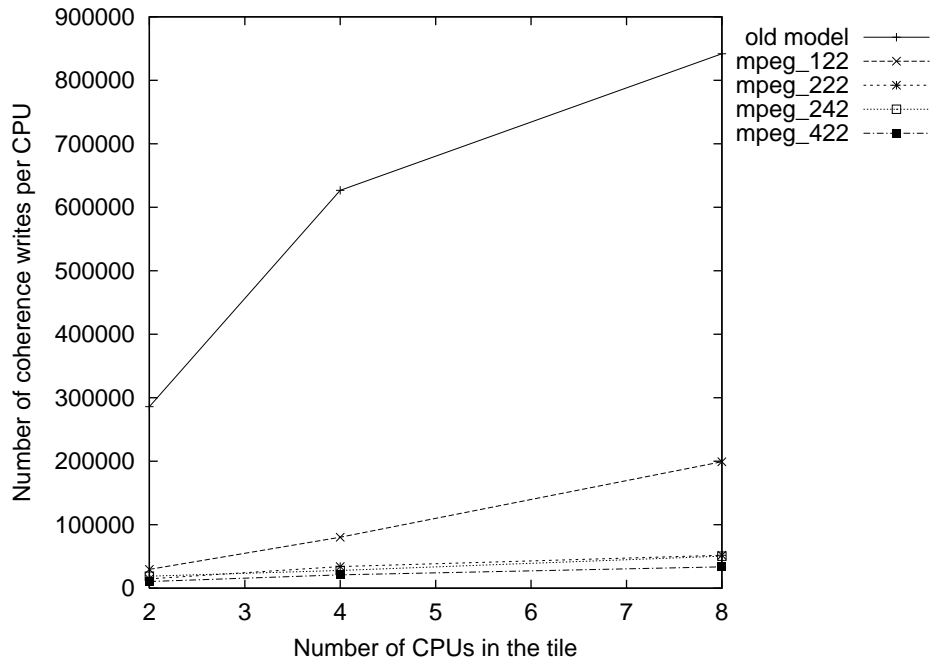


Figure 6.5: Number of coherence writes vs Number of CPUs

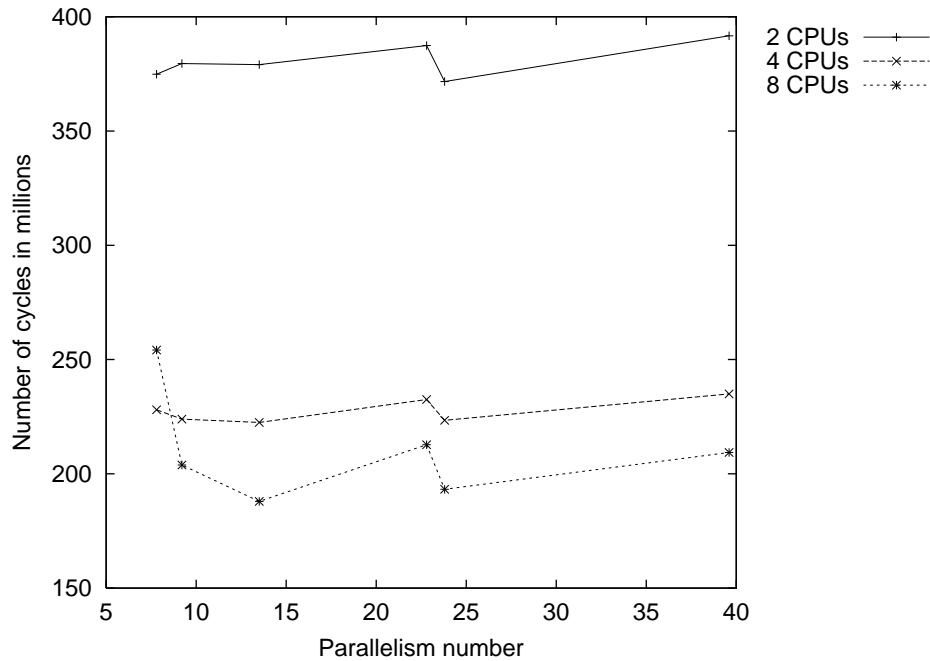


Figure 6.6: Number of cycles vs Parallelism Number

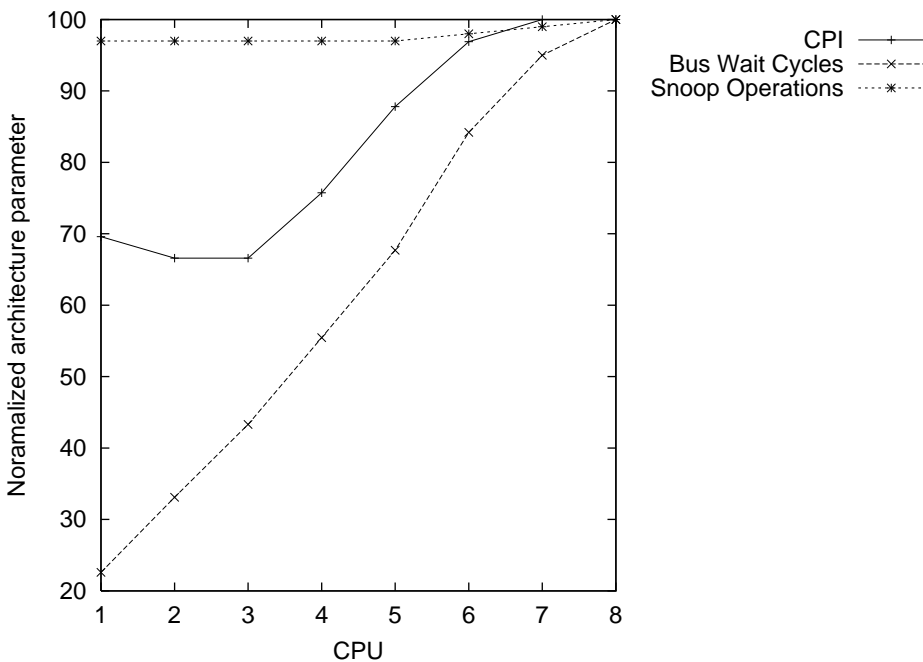


Figure 6.7: Normalized architecture parameters vs CPU id

Summarizing, more than 100% speed up is achieved for models of type *mpeg_ijk* compared to *old_model*, when application was run on a *tile* with 4 and 8 CPUs. Furthermore, as the number of CPUs is increased, conflicts to get access to bus increases and communication resource becomes expensive.

Chapter 7

Conclusions

7.1 Conclusions and Contributions

We modeled an MPEG-2 decoder using YAPI. The model has following features:

- New model is more than 100% faster when run on SpaceCAKE architecture with more than 4 CPUs.
- More than one MPEG-2 decoders can be instantiated, working on different video sequences.
- A number of configurations can be obtained by instantiating more processes or process networks at different levels (at whole application level, slice decoder level or within slice decoder).
- The model is optimized for lesser communication overhead by resizing FIFOs and reducing amount of data transfer over FIFOs.

There are a number of factors which directly affect the performance of an application model. Unidirectional communication, balanced pipelines, granularity of operation are a few typical examples. The model features such as bidirectional communication, unbalanced pipelines, feedback loops etc. could reduce the speed significantly and might lead to deadlocks.

We mapped the application model onto SpaceCAKE architecture to study architectural performance parameters. We observed that an application, which has more parallelism, runs faster (takes lesser number of cycles). Further communication structure of application is equally important. Application model should be optimized for lesser communication overhead.

We found from experiments that as the number of processors in the architecture increases, conflicts for communication resources increases. So after a certain number of processors, increasing the number of CPUs does not improve performance significantly. We also observed that for our MPEG-2 decoder application, presence of 4 CPUs in the tile gives performance which is good enough. Adding more CPUs must be backed up with more communication bandwidth. We also observed that for fixed number of CPUs in the architecture, there is a limit beyond which increasing the number of processes in the application model does not improve performance much.

7.2 Further Research

Faster MPEG-2 decoder

In the current implementation of MPEG-2 decoder model, we allowed only one process (i.e. Tinput) to get access to video sequence directly. Rest of the processes get access using buffering mechanism. The model can be further improved by reducing buffering and allowing more processes (particularly variable length decoders) to have direct access to video sequence.

More Applications

The application set can be expanded by utilizing our MPEG-2 decoder model. This extension includes applications such as picture scaling, picture in picture, MPEG-2 decoding followed by various picture quality improvement algorithms etc. The bigger application set can be utilized more effectively for design space exploration.

Performance Visualization

At present we do not get a dynamic view of application running over SpaceCAKE architecture. Similar is the case with architecture. The application model, thread scheduler and architecture can be extended to generate traces in some specific format. These traces can be given as input to performance analysis tool [7] and dynamic view of whole simulations can be visualized. This activity may help in identifying problems in both application and architecture.

Bibliography

- [1] Peter Pirsch and H. J. Stolberg. VLSI implementations of image and video multimedia processing systems. *IEEE Trans. on Circuits and Systems for Video Technology*, 8(7):878–891, November 1998.
- [2] E. A. de Kock and G. Essink. *Y-chart Application Programmer's Interface: The YAPI Programmer's and Reference Guide Version 0.4*. Nat.Lab. Technical Note xxx/99, February 2000. Philips company restricted.
- [3] E. A. de Kock et al. YAPI: application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'00)*, pages 402–405, June 2000.
- [4] MAJC Documentation. *MPEG-2 Video Decompression on a Multi-processing VLIW Microprocessor*.
<http://www.sun.com/microelectronics/MAJC/>
- [5] Angelos Bilas et al. Real-time parallel mpeg-2 decoding in software. In *Proc. 11th International Parallel Processing Symposium (IPPS)*, April 1997.
- [6] Ed F. Edwin Rijpkema et al. High level modeling for parallel executions of nested loop algorithm. In *Proc. ASAP'2000*, 2000.
- [7] Anup Gangwar. Eclipse performance analyzer. M.Tech. thesis, Department of Computer Science & Engg., Indian Institute of Technology, Delhi, 2000.

-
- [8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2000.
- [9] E. Iwata and K. Olukotun. Exploiting coarse-grain parallelism in the mpeg-2 algorithm. Technical Report CSL-TR-98-771, Stanford University Computer Systems Laboratory, September 1998.
- [10] Gills Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress 74*. North Holland Publishing Co, 1974.
- [11] Brian C. Smith Ketan Patel and Lawrence A. Rowe. Performance of a software mpeg video decoder. In *Proc. ACM Multimedia Conference*, 1993.
- [12] A. C. J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, 1999.
- [13] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proc. of IEEE*, 83(5):773–801, May 1995.
- [14] *Information technology - Generic coding of moving pictures and associated audio information: Video*. ISO/IEC 13818-2, 1996.
- [15] Pieter van der Wolf et al. An mpeg-2 decoder case study as a driver for a system level design methodology. In *Proc. CODESS'99*, 1999.
- [16] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg and Didier J. LeGall. *MPEG VIDEO COMPRESSION STANDARD*. Chapman & Hall, New York, 1996.
- [17] *PR1910 User Manual*. Philips Semiconductors.
- [18] *TSS User Manual*. Philips Semiconductors.
- [19] *PI-Bus Specification*. Philips Semiconductors.

Index

- Amount of Parallelism, 23
- Application modeling, 17

- B picture, 12
- Balanced Pipelines, 25
- BCU, 43
- block, 13
- Bus, 42
- Bus Wait Cycles, 50

- cache coherence, 42
- Cache Coherence, 51
- Communication, 6, 8, 22
 - primitives, 8
 - Read, 8
 - Select, 8
 - Write, 8
- Communication Overhead, 24, 48
- Computation Overhead, 45
- context switch, 46
- CPI, 50
- CPU, 42

- EOB, 13

- Feedback Loop, 23
- FIFO, 7, 8, 17, 32

- GOP, 12
- Granularity of Operation, 24

- I picture, 12
- IDCT, 20
- In, 8
 - InPort, 8

- Kahn Process Network, 5

- macroblock, 12, 16
- Memory Usage, 27
- MPEG-2, 11
- MPEG-2 Decoder, 17, 28
 - Tadd, 20, 36
 - TdecMV, 20, 34
 - Thdr, 18, 29
 - Tidct, 20, 36
 - Tinput, 18, 29
 - Tiq_idct_add, 35
 - Tisiq, 20, 35
 - Tmc, 34
 - TmemMan, 21, 29
 - Toutput, 21, 31
 - Tpredict, 20, 34
 - TsliceDec, 31, 32
 - Tvld, 18, 32

- TwriteMB, 21, 34
- MPEG-ijk, 36
- multiprocessor, 41
- nested loop, 15
- Out, 8
 - OutPort, 8
- P picture, 12
- Parallelism number, 46, 52
- Pipelines, 22
- PROXY, 43
- Scalability, 27
- sequence_end_code, 12
- slice, 12, 16
- SpaceCAKE Architecture, 41
- sync, 28
- Synchronization Overhead, 25
- Thread Scheduler, 40
- Thread scheduling
 - thread pool, 40
 - thread stealing, 40
- Tile, 41
- Tiq_idct_add, 34
- Tsink, 28
- Tsource, 28
- TSS, 48
- Unidirectional Communication, 24
- video sequence, 12
- workload, 8
- Y-chart, 2, 5
- YAPI, 5, 7, 17
 - Process, 5, 7
 - Blocked state, 7
 - main, 7
 - Ready state, 8
 - Running state, 7
 - Process Network, 7

