Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Diploma Thesis
at the Department of Information Technology
and Electrical Engineering

MPEG-2 Decoder for SHAPES DOL

Simon Mall

Advisors: Wolfgang Haid, Kai Huang
Professor: Prof. Dr. Lothar Thiele

April the 16th, 2007

Abstract

The SHAPES project addresses the challenge of finding a scalable HW/SW design style for future CMOS technologies and proposes an architecture consisting of "small" tiles connected by "short" wires. The Distributed Operation Layer (SHAPES DOL) helps the programmer to find an optimal mapping of an application onto this architecture. In order to exploit the parallel hardware, an application for the SHAPES DOL is specified according to a process network based model of computation.

In this diploma thesis, an MPEG-2 video decoder application for the SHAPES DOL is developed. The decoder is implemented as a reconfigurable Kahn Process Network, meaning that the number of processes can be adjusted without changing the implementation of the processes.

Several mapping–relevant parameters available at the application level are identified and a simulation–based method for obtaining these application parameters is presented.

A prototype of the mapping stage is implemented, which makes use of an evolutionary algorithm to solve the multi–objective mapping optimization problem. The mapping stage is used to map the MPEG-2 video decoder application onto a multi processor architecture.

Contents

1 Introduction **1**

1.1 The SHAPES Project 1

1.2 Distributed Operation Layer (SHAPES DOL) 3

1.3 Motivation and Goals 5

2 MPEG-2 Video Decoder **7**

2.1 The MPEG-2 Video Standard 7

2.2 Selection of a Reference Implementation 21

2.3 Parallelization of the Algorithm 28

2.4 Implementation 38

3 Profiling DOL Applications **47**

3.1 Mapping–Relevant Parameters 47

3.2 Tracing a DOL Application 49

3.3 Extraction of the Parameters 54

3.4 The `DOLProfiler` Package 55

4 The Mapping Stage **59**

4.1 Overview 59

4.2 System Specification 62

4.3 Gene Representation 73

4.4 Performance Data and Objective Values 81

5 Evaluation and Results **85**

5.1 MPEG-2 Decoder Application 85

5.2 Profiling Results 86

5.3 Mapping Optimization 89

5.4 Conclusion and Future Work 91

A Task Description **97**
A.1 Introduction . . . 97
A.2 Tasks . . . 98
A.3 Project Organization . . . 101

B Presentation Slides **103**

C CD–ROM Contents **115**
C.1 Directory Contents . . . 115
C.2 Step–by–Step Instructions . . . 117

1 Introduction

This thesis is part of an international project called SHAPES (scalable software hardware architecture platform for embedded systems). The first section covers the motivation for initiating the SHAPES project and a description of some of its goals and concepts. The second section describes the Distributed Operation Layer, which is ETH Zurich's contribution to the project. The last section of this chapter then presents the motivation of this thesis in the context of the SHAPES project and summarizes its goals.

1.1 The SHAPES Project

1.1.1 Motivation

Embedded systems are getting more and more complex. The number of gates per system is increasing and future systems on chip will integrate designs containing billions of gates. This development also leads to problems, as, for instance, the increasing design complexity which has to be managed. A second main problem is the *wire delay problem*. As CMOS technologies become smaller and smaller, the gate delays decrease. However, as the base fabrication technology scales to smaller dimensions, the delay through a wire of fixed length increases, which limits the overall clock frequency [1]. This alarming development is partly eased by the fact that the wire lengths are not fixed, but rather shrink in length with smaller technologies. There is nevertheless a wire problem because the number of modules per chip grows exponentially and even if a single module does not have increasing wire problems, their accumulation certainly does. In addition, there are global communication wires which do not shrink in length and therefore indeed have increased delays [1]. In summary, the wire delay problem is not unsolvable, but it does require profound changes in hardware/software design style for future technologies.

1.1.2 The SHAPES HW/SW Architecture

The approach taken to address these problems is the use of *tiled architectures*. A tiled architecture consists of predefined processing tiles which are connected to each other. The SHAPES project proposes heterogeneous tiles, which consist for instance of a RISC processor, a very long instruction word (VLIW) DSP, a distributed network processor and on–tile memories and peripherals [2]. A very important issue is *scalability*, which means that an application should be portable to a different SHAPES hardware architecture without much effort. Specifically, it should be possible to map an application onto architectures with largely different amounts of tiles. Table 1.1 shows the SHAPES target for the range of scalability [3].

The tiled architecture approach has many advantages, it is however difficult for an application to fully exploit its potential. There may be long delays between distant tiles, overloaded communication resources or the application may not expose enough parallelism. In all these cases, the architecture's full computing power cannot be exploited. The *system software* therefore has to make sure that the applications are executed efficiently on the SHAPES hardware, while minimizing the effort required for the application programmer. This is the main software challenge. Two key points are considered: first, as the system itself is highly parallel, so should be the application. The application programmer must be able to fully expose the algorithm's parallelism to the SHAPES platform, which makes

4–8 tiles	low–end single modules for mass market applications
2000 tiles	classic digital signal processing systems
	(e.g. radar, medical equipment)
32000 tiles	high–end systems requiring massive numerical computation

Table 1.1: SHAPES target for the range of scalability.

it necessary to break with the conventional way of writing an application. But even if the application is written in a way that exposes the parallelism well, this information about the algorithmic structure must be preserved by the SHAPES system software. Second, the system software must be fully aware of important architectural parameters like bandwidth, computing capabilities and latencies [3].

1.2 Distributed Operation Layer (SHAPES DOL)

The Distributed Operation Layer is a part of the SHAPES system software environment. Its purpose is to help a programmer of a SHAPES platform to find an efficient mapping of the application onto the hardware. This includes the mapping of application tasks onto computation resources, as well as the mapping of communication links onto communication resources. For making the best use of the parallel hardware resources, it is necessary that the application programmer follows a set of rules and uses a set of interfaces, the *programming model*. According to the DOL programming model, which is described in Section 1.2.1, an application consists of several *processes* which are connected to each other with FIFO *channels* to build a *process network*. There is no shared memory, all communication must therefore take place via channels. Consider, for instance, a specific algorithm that has to be implemented on a SHAPES platform. The application programmer first extracts the algorithm's task–level parallelism and implements the processes $\mathcal{P}_1 \dots \mathcal{P}_n$, as well as the network $\mathcal{N}$ which describes the way the n processes are connected to each other. The programmer is therefore *exposing* the algorithm's parallelism to the DOL, which in turn tries to *exploit* it as much as possible during the mapping process. The extraction of parallelism from an application program is left to the programmer, since profound knowledge of the application domain is required.

1.2.1 The Programming Model

The SHAPES programming model is similar to the *YAPI* programming interface [4], which is an extension of the Kahn process network model [5].

The Kahn process network model is a model for parallel computation, where a program consists of a number of processes, which are connected to each other by FIFO channels. Each process is a usual sequential program extended by a communication interface consisting of two functions: the `wait(U)` function, which performs a blocking read access to the specific channel $\mathcal{U}$, and the `send I on W` statement, which writes the variable $\mathcal{I}$ to the channel $\mathcal{W}$. While the `wait` function blocks the process until enough data is available on the channel, nothing can prevent a process from sending data over a channel. The processes therefore communicate via infinitely large FIFO queues. Kahn process networks have two important properties. First, the processes of a KPN are *monotonic*, which means that only partial information about the input is needed to compute a partial output, or, in other words, future input concerns only future output. This is indeed an important property, because it allows parallelism — a process can start computing before all of its input values arrived [5]. As a second important property, Kahn process networks are *determinate*, which means that the same input history always produces exactly the same output, independent of the scheduling.

The SHAPES DOL API contains the basic communication primitives `DOL_write()` and `DOL_read()`, but differs from the Kahn process network model in the following way:

i) As infinitely large FIFO queues are not realizable, every channel is instantiated with a maximum buffer size. This causes the `DOL_write()` function to stall the calling process if the FIFO queue is full.

ii) The Kahn process network model does not allow in any way to test if the channel $\mathcal{U}$ is empty, before invoking `wait(U)`, which was found to be too restrictive in some cases and may cause inefficiency [6]. The SHAPES DOL API therefore provides two additional functions which allow for a read/write test, namely `DOL_rtest()` and `DOL_wtest()`.

iii) For being able to end the simulation process properly, a call to the function `DOL_detach()` allows a process to remove itself from the pool of active processes. The simulation ends as soon as all processes are detached.

A list of all communication primitives is shown in Table 1.2. Before invoking a blocking read function, a call to the read testing function can be used to find out whether the required amount of data has already been produced by the preceding process. This allows — if a process has more than one input ports — to select at runtime from which port to read. It is, however, up to the application programmer to decide whether the testing functions are actually used. By omitting the two testing functions completely, a predetermined order of port accesses results which may simplify the process structure, reduces communication overhead and leads to the Kahn process network properties mentioned before.

DOL_read(port, buffer, length, process) Reads *length* bytes from *port* and stores the obtained data in *buffer*. If less than *length* bytes are available, the calling process is blocked.
DOL_write(port, buffer, length, process) Writes *length* bytes from *buffer* to *port*. If the FIFO connected to *port* has less than *length* bytes of free space, the calling process is blocked.
DOL_rtest(port, length, process) Checks whether *length* bytes can be read from *port*.
DOL_wtest(port, length, process) Checks whether *length* bytes can be written to *port*.
DOL_detach(process) Detaches the process and prevents the scheduler from firing the process again.

Table 1.2: The SHAPES DOL communication interface.

1.2.2 Functional Verification

One of the main advantages of platform–based design is the possibility to start developing applications before the final hardware architecture is specified. The DOL allows for a functional verification of an application from the beginning of the development process. The application — specified, according to the programming model, with a process network and the process definitions — can automatically be transformed into SystemC [7] source code, which can be compiled and run on a single workstation. The process network can thereby be run in a simulated real–time environment.

1.3 Motivation and Goals

Since the programming model is defined, it is important to check whether it is suitable for complex programs. Additionally, it is interesting to gain some experience in developing applications for the SHAPES platform using the functional simulation. Based on this motivation, the first goal of this thesis is to implement an MPEG-2 video decoder application for the SHAPES DOL. This implementation shall expose the parallelism available in the decoding algorithm, and can then be used for further investigations. The implementation of the MPEG-2 video decoder is covered in Chapter 2.

Having a complex example application at hand, the next goal is to implement a first prototype of the mapping stage. For this reason, mapping–relevant parameters need to be extracted from the application and used for optimizing the mapping of the application onto the architecture. The parameters are extracted by profiling the functional simulation (Chapter 3). The prototype of the mapping stage is presented in Chapter 4.

2 MPEG-2 Video Decoder

2.1 The MPEG-2 Video Standard

This section presents the ISO/IEC 13818 standard, which is the second standard developed by the Moving Picture Experts Group (MPEG) and is therefore also known as the MPEG-2 standard. As mentioned in the introduction, the reason for implementing an MPEG-2 decoder is to test the DOL with a program of moderate complexity. Since it is not the goal to develop a commercial video player, the audio part is omitted.

2.1.1 Development of the MPEG Algorithm

The term *Moving Picture Experts Group* has two related meanings. On the one hand, it is the working group of the ISO/IEC (International Organization for Standardization / International Electrotechnical Commission) which is in charge of the development of international standards for video and audio compression. On the other hand, the term MPEG is commonly used to describe their main product, which is a family of standards of compressed digital audio–visual information. The MPEG format is known to achieve a high compression rate for a required picture quality, which is desirable for exploiting both storage and transmission capacity. However, the high compression comes at the cost of complexity: these standards “probably represent one of the most complicated communication standards ever written” (stated in 1997 by [8]).

In 1992, four years after the formation of the MPEG working group, the MPEG’s first international standard MPEG-1 was completed as ISO/IEC 11172. MPEG-1 was successful in the computer entertainment industry. However, the missing support for interlaced scan[1] prevented its use in digital television [9]. MPEG-1 was later used as the standard for the Video-CD, and it defines the popular audio compression format MP3 (MPEG-1 Audio Layer 3).

In 1994, the MPEG-2 standard was approved as ISO/IEC 13818, an extension of MPEG-1 that adds support for interlaced video, more color formats and other advanced coding features. In order to support a smooth transition from MPEG-1 to MPEG-2, it retained full backward compatibility. MPEG-2 is used in the digital versatile disk (DVD) as well as for digital television (as a video compression engine). With the increasing popularity of high definition TV (HDTV), a new standard was planned especially to handle the HDTV signals, but it was soon discovered that the MPEG-2 standard was sufficient for that purpose, and the MPEG-3 standard was canceled.

The next generation MPEG standard MPEG-4 (ISO/IEC 14496) was started in 1993. The focus of MPEG-4 lies on the convergence of the three major application areas, namely *Television and Film* (provides audio–visual data), *Computer* (provides interactivity) and *Telecommunications* (provides the ability of transmission). The traditional boundaries between these industries disappear more and more — computers make use of video, sound and communications, interactivity is being added to television and video is being added to telecommunications. In terms of an audio–visual standard this means support of interactivity, high compression and universal accessibility [8].

MPEG-7 (ISO/IEC TR 15938) differs in that it does not deal with the actual encoding and decoding of audio–visual data, but it rather describes the way this data is stored, and is hence called *multimedia content description standard*. The ambition is to allow the applications an effective and efficient access, i.e. searching, filtering and browsing.

The latest MPEG standard, MPEG-21, is now in the development phase. This so called *multimedia framework* takes into account a new multimedia usage context, namely the accessibility of audio–visual data anywhere and at any time. This usage context, together with the fact that more and more digital media is produced for professional and personal use, leads to several new concerns like content management, protection of rights,

[1]Interlaced scan, in contrast to progressive scan, first displays every second line of the frame (the *odd field)*, returns then to the top and completes the frame (the *even field*). Interlaced scanning is described in detail later.

13818-1:	**Systems**
13818-2:	**Video**
13818-3:	**Audio**
13818-4:	Conformance
13818-5:	Software
13818-6:	Digital Storage Media — Command and Control (DSM–CC)
13818-7:	Non Backward Compatible (NBC) Audio
13818-8:	10-Bit Video (work item, removed)
13818-9:	Real Time Interface
13818-10:	DSM–CC Conformance

Table 2.1: Parts of the MPEG-2 standard.

protection from unauthorized access or modification and protection of privacy. For this reason, a fundamental unit of distribution and interaction is defined (*"digital item"*) and the goal of MPEG-21 is to support users to exchange, access, consume, trade and otherwise manipulate these digital items in an efficient and transparent way [10].

2.1.2 MPEG-2 Overview

The MPEG-2 standard consists of 10 parts, which are referred to as 13818-1 through 13818-10 (Table 2.1). The *Systems* part describes how several audio–, video– and data streams are multiplexed together to form one single bitstream. The systems part of the standard is very flexible and suited for a nearly error–free as well as for an error–prone environment. The *Video* part is the most important part and describes the video bitstream syntax and semantics. It is described in detail in the following sections. The *Audio* part is omitted here, as we will not consider audio decoding. The *Conformance* part specifies how tests can be performed to verify whether a decoder meets the requirements. The *Software* part consists of reference implementations of audio and video encoders and decoders. Parts 6 – 10 are several extensions that are not relevant for our purpose. The interested reader is referred to [8] for a complete description.

The MPEG-2 standard defines the video bitstream syntax, semantics and the general decoding process. The syntax describes the structure of the bitstream, i.e. the rules to write and read the headers and picture data. The semantics then explains the meaning of the header values and how the decoder has to interpret them. The decoding process explains step–by–step how the encoded picture data is decoded in order to get the plain luminance and chrominance values, which can then be passed on to the display process or written to a file. The exact way the single decoding steps are realized is to a large extend left open.

The encoding process — which is much more complex and time–consuming than the decoding process — is not described by the standard. Every application that is able to convert an uncompressed video sequence into an MPEG-2–conform bitstream is a valid MPEG-2 encoder. This enables a wide range of encoder implementations with different qualities in terms of compression rate, encoding speed and image quality. A DVD encoder, for instance, will be highly optimized in terms of image quality and compression rate, while the encoding speed is not important (because the encoding is done only once at production time). A real–time encoder, on the other hand, must sacrifice image quality in order to reach the required encoding speed and compression rate.

2.1.3 Video Representation

Coding means *altering of the characteristics of a signal to make the signal more suitable for an intended application*[2]. According to this definition, the MPEG-2 standard is indeed a coding standard in that the multimedia signal is made more suitable for transmission and storage. However, a multimedia signal must first be made more suitable for being processed by a digital system, which itself can be called a coding process. To prevent confusion, we will call the latter *video representation.* Video representation consists of spatial and temporal sampling, color representation, and quantization [9].

Spatial Sampling (Scanning)

We start with a two–dimensional image consisting of picture elements called *pels*[3]. The process of converting this intensity matrix to a one–dimensional array or a waveform is called *scanning.*

The simplest scanning method is *progressive raster scanning.* The image is scanned one line after the other from left to right and from top to bottom. This rather abstract description is sufficient for transforming an intensity matrix (digital image) into an intensity stream (byte stream). If, on the other hand, an analog device like a cathode ray tube is considered, numerous additional challenges have to be taken into account.

[2]According to the United States Federal Standard 1037C, "Telecommunications: Glossary of Telecommunication Terms"

[3]The term *pixel* in computer terminology has exactly the same meaning. However, as in video processing the term *pel* is commonly used, we will stick to this term here.

The electron beam has then actually to be moved along the lines of the screen, and the movement back to the beginning of the next line introduces a delay (the so–called retracing period), as well as the movement from the bottom line of one image to the top line of the next image.

One way to improve the image quality without the need of extra bandwidth is *interlaced raster scanning*. The idea behind interlaced raster scanning is to take advantage of the characteristics of the human visual system, which is not able to perceive spatial details for fast motion. An image now consists of two *fields*. The first field contains the odd lines, whereas the second field contains the even lines. The beam therefore first scans lines 1, 3, 5, . . . , returns back to the beginning of the same image and scans lines 2, 4, 6, . . . , and continues with the first line of the next image. At the cost of spatial resolution, fast moving objects are now much better resolved in time. As mentioned before, this matches the characteristics of the human eye. Full support of interlaced raster scanning is one of the main improvements from MPEG-1 to MPEG-2.

Temporal Sampling

The visual system is quite insensitive to temporal changes, an illusion of motion is created by showing at least 16 frames per second. In motion picture technology, the temporal sampling is performed at a rate of 24 frames per second. In television 25 and 30 frames per second are commonly used [9].

Color Representation

One remaining question is the representation of a pel's color. The trichromatic theory of color vision implies that nearly every color can be represented as a combination of the three primary colors red, green and blue. However, instead of transmitting the RGB values, most of the color formats transform them into luminance and chrominance values. The luminance is compatible with monochrome devices, which is the main advantage of this color representation. The chrominance values consist of hue (frequency) and saturation (amount of black).

The format used in MPEG-2 is the *ITU-R BT.601 digital video standard*[4]. The color space used is YCrCb, i.e. the format contains one luminance component Y and two chrominance components Cr and Cb. A camera imaging a scene generates for each pel the primary color RGB values, for transmission and storage they are converted into luminance/chrominance values YCrCb, and right before displaying, they are transformed

[4]The ITU-R was formerly called CCIR and the standard was therefore called CCIR-601.

back into the RGB color space (if a color display is used). The MPEG-2 standard supports 3 different chrominance formats, which are called 4:2:0, 4:2:2 and 4:4:4 format. The chrominance format specifies the proportion of luminance values to chrominance values:

4:2:0 The Cb and Cr matrices are one half the size of the Y-matrix in both horizontal and vertical dimensions. In oder words, there is one Cb and one Cr value for every four Y values.

4:2:2 The Cb and Cr matrices are one half the size of the Y-matrix in the horizontal dimension and the same size as the Y-matrix in the vertical dimension. This means one Cb and one Cr value for every two Y values.

4:4:4 The Cb and Cr matrices are the same size as the Y-matrix in both horizontal and vertical dimensions. There is one Cb and one Cr value for each Y value.

With all three chrominance formats, the location of the Y, Cb and Cr values is exactly defined in the standard, for progressive as well as for interlaced scanning.

2.1.4 Video Compression

An uncompressed short video sequence of 4 minutes duration encoded in the NTSC ITU-R BT.601 4:2:2 format requires more than 5 GB of storage capacity, as can be verified quickly by calculating

$$(720 \cdot 486 + 2 \cdot 360 \cdot 486)\,\frac{\mathrm{B}}{\mathrm{frame}} \cdot 30\,\frac{\mathrm{frames}}{\mathrm{s}} \cdot 240\,\mathrm{s} = 5.04 \cdot 10^9\ \mathrm{B}\ ,$$

where $720 \cdot 486$ is the number of luminance pels, and $360 \cdot 486$ is the number of chrominance pels per picture in the NTSC ITU-R BT.601 4:2:2 format. This is exactly what a single–sided DVD can hold. Luckily, coded video data contains a lot of redundancy.

Spatial Redundancy

Spatial redundancy, also known as *intra frame* redundancy, can be reduced by processing every single frame of a video separately. As video is nothing more than a sequence of images, the same techniques as for image compression can be used. A very popular approach for image compression, used in JPEG and adopted by many video coding standards, including MPEG-2, is called *Discrete Cosine Block Transform*[5]. First, assume an image with large fields of the same color (e.g. a black drawing on a white paper). Images

[5]See e.g. [11] for an introduction to the discrete cosine transform.

of this type can be compressed very easily with run length encoding (RLE): instead of saving all consecutive bytes of the same value (e.g. ‘0000000000’), the amount of bytes is saved (‘10’ ‘0’, meaning 10 consecutive zeros). However, this approach does not work in case of high resolution images where such sequences almost never occur. Instead, high resolution images contain smooth transitions which cannot be compressed by the means of RLE. Even worse, the human eye is very sensitive to these low frequency parts and quantization steps are clearly visible. On the other hand, the eye is quite insensitive to high spatial frequencies. The discrete cosine block transform exploits this fact. The procedure is as follows:

i) The image is divided into blocks.

ii) Each block is transformed into the frequency domain by the means of the discrete cosine transform (DCT). Most of the useful information now lies in a few low–frequency coefficients, whereas the high–frequency coefficients are all close to zero.

iii) The coefficients are quantized which sets the high frequency coefficients to exactly zero.

iv) As the low–frequency coefficients are lying in the upper–left and the high–frequency coefficients in the bottom–right corner, zig–zag scan is applied in order to bring similar frequencies close together.

v) The resulting stream of coefficients is very well suited for RLE, as it contains a long sequence of zeros (the high–frequency coefficients).

The fact that the human visual system is more sensitive to low spatial frequencies is exploited by using different quantization steps for different frequencies. This *intra frame quantization matrix* can be chosen freely but the MPEG-2 standard provides a default matrix that is used if no other matrix is specified. With this default quantization matrix, transform coefficients are quantized more coarsely with increasing horizontal and vertical spatial frequencies.

The described procedure is not a simple low–pass filter, as the high frequency coefficients are quantized and not cut off. The small but important difference is the following: If an image contains significant high frequency coefficients, they are not removed, but it rather degrades the compression rate. This is important because the MPEG-2 standard tries to reach a good compression rate while maintaining a high quality image rather than to force a specified compression rate at the cost of image quality.

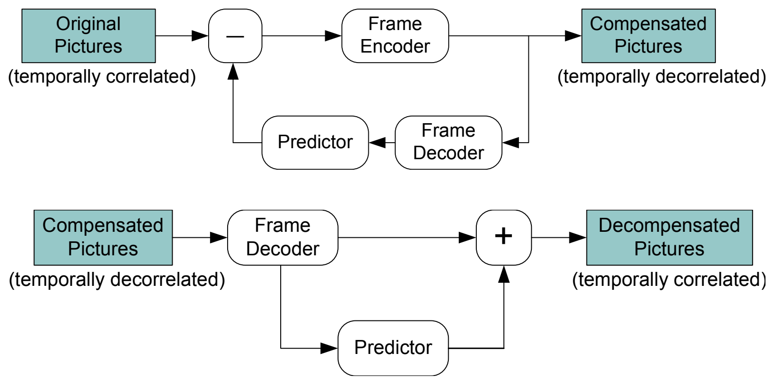


Figure 2.1: General predictive coding and decoding scheme.

Temporal Redundancy

Temporal redundancy or *inter frame*[6] redundancy stems from the fact that consecutive images are often highly correlated. The simplest way to decorrelate consecutive images is *frame differencing*. Instead of coding the image itself, the difference between the actual and the previous image is coded. If there is little motion the difference image is mostly uniform and can be coded very efficiently (as a uniform block has all coefficients = 0 except the DC-coefficient). Frame differencing is a predictive coding technique where the predictor is simply a frame buffer (the prediction is therefore the previously decoded frame). Figure 2.1 shows the general predictive coding and decoding scheme. It can be verified easily that the decoded image is equal to the uncoded (original) image, as long as the predictor is the same in both cases.

Of course, the first frame of a sequence cannot be coded predictively, as there is no previous frame. Such a frame is called *intra coded* frame, or I frame for short, and considers only spatial redundancy. For being able to access frames in the middle of a video sequence, I frames are periodically inserted.

Motion Compensation The more motion a video contains, the less we gain from frame differencing. This calls for *motion compensation*. The idea is to improve the prediction by estimating the motion of objects between frames.

The main motion compensation unit is the macroblock, which consists of 6 to 12 blocks (depending on the chrominance format). Each macroblock is accompanied by one or more

[6]In a progressive video, a *frame* is the same as a picture, whereas in an interlaced video, a picture may refer to a frame or a single field, depending on the context. The terms *picture* and *image* are interchangeable.

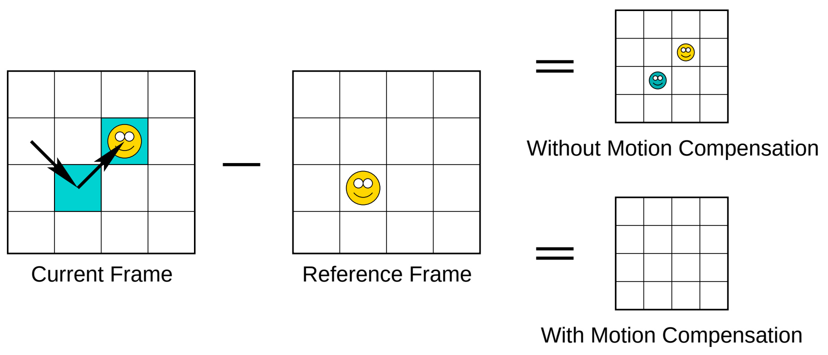


Figure 2.2: Illustration of the use of motion vectors.

motion vectors. The motion vector indicates, which macroblock of the reference frame must be used to reconstruct the macroblock of the current frame. This is illustrated in Figure 2.2. First, assume predictive coding without motion compensation, i.e. frame-differencing. Taking the difference between the current frame and the reference frame, all macroblocks are uniform except two: the one with the smiley in the current frame and the one with the smiley in the reference frame (resulting in a “negative smiley”). By introducing the motion vectors, we can define for each macroblock of the current frame exactly which macroblock of the reference frame should be taken to build the difference. In this way it is possible to refer to the macroblock containing the same object at a different position, and as a consequence, all resulting macroblocks are uniform.

Note that the left motion vector in Figure 2.2 was chosen arbitrarily. The important thing is that the macroblock at the origin of the vector in the reference frame is the same as or at least very similar to the macroblock at the head of the vector in the current frame. It is in fact not necessary at all that the motion vectors represent motion of real existing objects, it is rather the encoders job to find the macroblock of the reference frame that leads to the most uniform differential values. It is mostly this task, which makes an encoder implementation so challenging and it is now obvious that finding the best motion vectors requires trading–of encoding speed for compression rate and image quality.

If the reference frame occurs before the current frame the prediction type is called *forward prediction*. Frames that are coded using only forward prediction are called P frames (*predicted* frames). The reference of a P frame can either be an I frame or another P frame. An extension of the concept leads to *bidirectionally predicted* frames (B frames). B frames use a past and a future reference frame and interpolate the result. This leads to an improved prediction, as can be seen from the following situation: assume a car suddenly appears from behind a building. The block containing the car clearly cannot be predicted using forward prediction. By taking the future frame as a reference, much better prediction is possible. Bidirectionally predictive coded frames are never used as references for prediction, otherwise a deadlock–like situation could occur where the current frame must be fully decoded in order to get its own reference frame. For maintaining causality, i.e. making sure that both reference frames of a B frame are decoded before the B frame itself, the frames are reordered during the encoding process (*coded order*). This makes the decoding process much simpler: the decoder can then process one frame after the other and simply restores the original order (*display order*) before sending them to the display process.

If a macroblock belongs to an intra–coded picture, no prediction is formed. The output of the predictor is then zero and the luminance/chrominance values of the macroblock directly represent the decoded data. As the predictor is not used at all, the compression of such macroblocks is significantly lower than their P and B picture counterparts, but they have the important advantage that they can be decoded without the use of a reference picture. On the other hand, there are blocks that are not coded at all, and even macroblocks that are skipped completely. In that ideal case, the prediction alone represents the decoded data, i.e. the prediction is perfect. In all the other cases, there is a prediction, but it is not perfect. The luminance/chrominance values of the macroblock must then be added to the luminance/chrominance values of the prediction in order to get the decoded data. In summary, the motion compensation procedure is the following:

i) The reference pictures are selected. P pictures have one and B pictures have two reference pictures. The procedure is somewhat complicated by the interlaced scanning compatibility of the MPEG-2 standard: if interlaced scanning is used, the standard allows for both field and frame prediction, and the fields and frames from which predictions are made may themselves have been decoded as either field pictures or frame pictures[7]. The selection rules are clearly defined in the MPEG-2 standard.

ii) The motion vectors are decoded. The motion vectors themselves are coded differentially with respect to previously decoded motion vectors. This reduces the number of bits to represent them.

iii) The prediction is formed by reading luminance/chrominance values from the reference picture, offset by the motion vector.

[7]The distinction between field pictures and frame pictures must be made in most decoding steps. Although this fact complicates a decoder implementation, the underlying concepts are the same in both cases. For this reason the explicit distinction is mostly omitted throughout this MPEG-2 introduction.

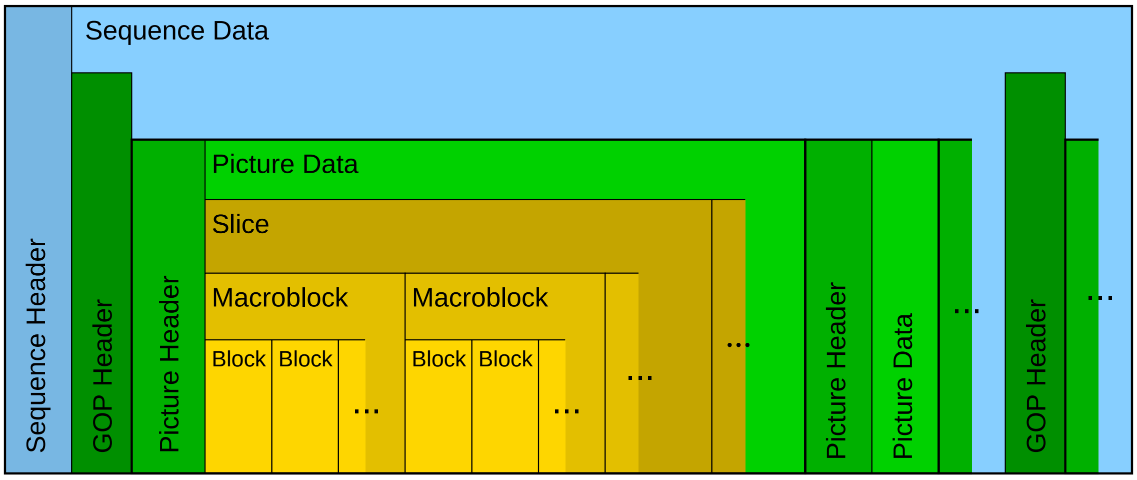


Figure 2.3: Nested syntactic structures of the MPEG-2 bitstream.

iv) The final decoded values are obtained by adding the luminance/chrominance values from the current block to the prediction data.

2.1.5 Bitstream Syntax and Semantics

The video bitstream is organized in a hierarchy in which syntactic structures contain one or more subordinate structures [12]. This is illustrated in Figure 2.3.

Video Sequence

The video sequence is the highest syntactic structure of the coded video bitstream. It consists of the *video sequence header*, extension and user data, several *group of pictures* structures (optional) and *picture* structures. The beginning of a video sequence header is marked with the hexadecimal value 0x000001B3. The header itself contains the image size, aspect ratio, frame rate, bit rate, quantizer matrix (if the default matrix is not used), etc. The extension field contains profile and level indication, the chrominance format, progressive/interlaced flag, and all the other MPEG-2 specific information.

Group of Pictures

The group of pictures structure consists of a header only. The GOP start code 0x000001B8 is a marker that separates independent groups of pictures. The picture immediately following the GOP header has to be an I picture, which means that it can be decoded without knowledge of the previous pictures. GOP headers can therefore be used for fast

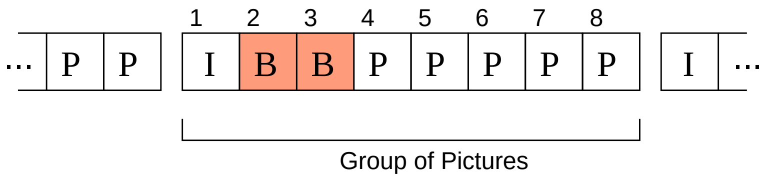


Figure 2.4: Illustration of a GOP in *encoded* picture order.

forward and random access: If a random access in the middle of the video sequence is required, all pictures are skipped until the next GOP header. The GOP header is however optional, the encoder can insert I pictures without being obligated to precede them with GOP headers. This still allows for fast forward and random access, but the decoder then has to read out every picture header in order to find the next I picture.

It is important to understand that the GOP header indicates the begin of an independent group of pictures in *display* order. Although an I picture must follow the GOP header, it does not imply that all the following pictures in coded order can be properly decoded. This is illustrated in Figure 2.4: it shows a complete GOP arranged in coded order. Pictures 2 and 3 are B pictures, which use the last P picture of the previous GOP as well as the first I picture of the current GOP as reference pictures. This means that in display order, pictures 2 and 3 come *before* picture 1, they were reordered during the encoding process for maintaining causality. This also means that they cannot be decoded properly in the case of a random access, where the last P picture of the previous GOP is not available. In terms of the display order, they do not belong to the current GOP and can be skipped in the case of a random access. If both of these B pictures use only backward prediction or no prediction at all (which is, according to the standard, allowed also for B pictures), they can be decoded properly, and the GOP is then called a *closed* GOP. The GOP header contains a bit indicating if the GOP is open or closed, which allows the decoder to act accordingly without examining the picture headers.

Picture

The picture start code is 0x00000100 and marks the beginning of a new picture header. The picture header contains, among other things, the picture coding type (I, P or B). It is followed by picture coding extension information and the actual picture data, which in turn consists of several *slice* structures.

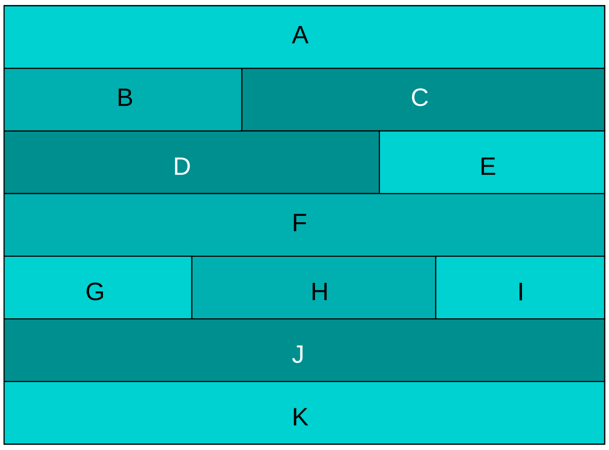


Figure 2.5: Division of a picture into slices.

Slice

A slice is a series of consecutive macroblocks of a single row. Figure 2.5 shows a possible division of a picture into several slices. Note that the slices cover the whole picture. This is called *restricted slice structure* and all profiles defined so far require this structure. The MPEG-2 standard itself allows also a more general slice structure, where only parts of the picture are covered by slices.

The slices occur in the bitstream in raster–scan order from left to right and from top to bottom. The slice start codes are 0x00000101 — 0x000001AF, where the last byte (0x01 — 0xAF) stands for the vertical position of the slice in the picture. The start code is followed by several flags and the macroblock structures.

Slices are the smallest entities which have their own start codes. As the entities become smaller their quantity increases. At a certain point the ease of byte–aligned start codes does not justify the "waste" of 32 bits per header. On the contrary, the goal of a high compression rate makes it necessary to introduce entropy coding at the cost of a more complicated decoding process, as described in the next section.

Macroblock

The macroblock header contains all the information that is needed for motion compensation. After the header, a well–defined number of block structures follow. The number of blocks and their order in the bitstream depends on the chrominance format and is illustrated in Figure 2.6.

In contrast to the previous structures, the macroblock header is not preceded by a start

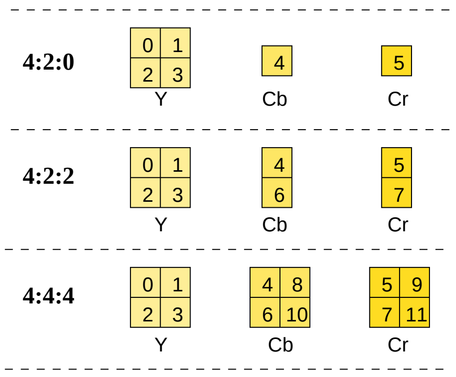


Figure 2.6: Number of blocks contained in a macroblock for a given chrominance format.

code any more, as mentioned before. In addition, *variable length codes* as a form of entropy coding are introduced, which means that the values of the header entries and the motion vectors are coded with different lengths, depending on their probability of occurrence. The decoding process is thereby somewhat complicated, in that the whole macroblock (including all contained blocks) must be variable length decoded in order to find the begin of the next macroblock. Significant computation is necessary to decode the variable length codes.

Block

The lowest syntactic structure is called block. In the encoded bitstream a block consists of the variable length coded DCT coefficients for a 8×8 chrominance/luminance matrix.

2.1.6 The Decoding Process

Assuming that all the header information is extracted and saved in global variables, the decoding process now describes how to transform the block of DCT coefficients into an 8×8 matrix of chrominance/luminance values. A simplified data flow diagram is shown in Figure 2.7.

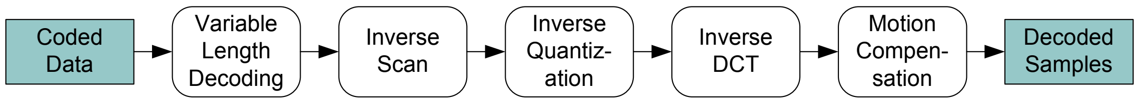


Figure 2.7: Simplified data flow diagram of the decoding process.

The variable length decoding (VLD) process converts the variable length bit strings into the motion vectors and the DCT coefficients according to lookup tables specified in the MPEG-2 standard. The resulting coefficients are one–dimensional. The inverse scan (IS) process then converts the one–dimensional stream into a two–dimensional array of coefficients, using either *zig-zag* or *alternate* scanning order. The inverse quantization (IQ) is essentially a multiplication by the quantizer step size, followed by a saturation to keep the coefficient values inside a specific range. These steps yield the complete spatial frequency domain description of the block, and the actual luminance/chrominance values can be obtained by applying the inverse DCT (IDCT). The final and most complicated step is motion compensation (MC), which is described in detail in Section 2.1.4.

2.2 Selection of a Reference Implementation

The aim of the first part of this thesis is to test the DOL with a complex algorithm. The DOL requires the application programmer to abandon the usual way of writing a program as a single sequence of instructions. Instead, the application programmer must expose the algorithm's inherent parallelism by writing (sequential) processes and a process network. The process network describes the way the processes are connected to each other. One of the key questions is whether the DOL provides enough functionality to allow the application programmer to do so in an efficient way. We do not intend to write an MPEG-2 implementation from scratch, so an existing reference implementation of the MPEG-2 standard is chosen. In this section, the criteria for choosing a reference implementation and five candidates are analyzed. The implementation that fits our purpose best is then presented in more detail.

2.2.1 Selection Criteria

A. Portability and Suitability for Parallelization. Platform–specific optimizations (MMX support and the like) are rather disturbing than helpful. Already included parallelism is advantageous but not necessary. The main building blocks are predetermined by the standard itself (like variable length decoding, inverse quantization, inverse DCT and motion compensation). We expect therefore that the implementations do not differ very much in their suitability for parallelization, especially since we are interested in coarse–grained parallelization.

B. Clearness and Documentation. An extensive documentation facilitates the parallelization considerably. The implementation should be well structured and clear.

C. Comparability. There are several similar platforms presented in literature, and the MPEG-2 algorithm is a common benchmark application. The reference implementation should be well comparable to related works.

D. Efficiency. An efficient implementation would be preferable, however, we expect a trade–off between efficiency and clearness. For our purpose clearness is more important than efficiency, because the efficiency of a sequential implementation is not necessarily portable to the parallel case. We can therefore not assume a priori that an efficient sequential implementation leads also to a better parallel implementation, whereas a clear and well documented implementation does that without much doubt.

E. Completeness. Not every decoder implements all possible kinds of MPEG-2 profiles. We expect an MPEG-2 library used in a commercial media player program and tested by thousands of users to be much more complete than a sample decoder programmed for academic reasons. Completeness is however again not our main concern.

F. License. Last but not least, it is important to have the right to use, change and publish the source code.

2.2.2 The Candidates

libmpeg2

The libmpeg2 library [13] is a SourceForge project for decoding MPEG-1 and MPEG-2 video streams. It contains highly optimized assembly routines for specialized hardware (e.g. MMX and AltiVec), but for the sake of portability, generic C routines are also available. Great effort was put into speed and the authors are very confident in that they wrote one of the fastest MPEG-2 decoder library. One of the main development goals was conformance, the MPEG streams are however restricted to the main profile. A drawback of the libmpeg2 is the absence of documentation. The library comes with a testbed (*mpeg2dec*) which includes a demultiplexer for MPEG-2 program streams. The library is widely used by media players, but, as far as we know, never referenced by scientific papers. The libmpeg2 library is released under the terms of the GNU General Public License (GPL).

libavcodec

The libavcodec library is a collection of video and audio codecs. Besides MPEG-1 and MPEG-2, it contains also MPEG-4 codecs and many different audio codecs. It is a part of the FFmpeg project [14] and is used by popular media players like *MPlayer*, *xine* and *VLC Media Player* and therefore offers compatibility with all possible MPEG bitstreams at the cost of high complexity. It is not very well documented. This library is released under the terms of the GNU Lesser General Public License[8] (LGPL).

MSSG mpeg2dec

The MPEG Software Simulation Group (MSSG) developed an MPEG-2 coder and decoder with emphasis placed on a correct implementation of the standard and a simple structure [15]. Its main purpose is to demonstrate a sample implementation and serve as an educational tool — it is not optimized for speed but based on relatively fast algorithms. It is often used in literature as a reference implementation for parallel decoding, e.g. in [16] and [17]. The package contains several bitstreams and a small verification program and it is freely available on an “as is” basis.

Jahshaka and Helix

The Jahshaka [18] and Helix [19] projects both are open source media players which contain MPEG-2 libraries. Both provide only end–user documentation, but a lot of information is available through developer forums. As they are embedded in very large projects they are not suited very well for our purpose, because of rather complicated compilation and configuration steps.

2.2.3 Presentation of the MSSG’s mpeg2dec

According to the criteria, the MSSG’s mpeg2dec is suited very well for our purpose. The mpeg2dec application is written in ANSI C and compiled and tested on several platforms (gcc on SunOS and Linux, djgpp on MS-DOS, Visual C++ on MS Windows). It contains 16 source files, 4 header files and a makefile. Table 2.2 shows the most important files together with a short description. The source code is well–structured and several functions can be used almost unchanged for our purpose, like VLD–, header decoding–, bit level– and DCT–routines. The remaining files follow the syntactic description of the

[8]The LGPL, in contrast to the GPL, allows to link a program to a non-LGPL program which can be free or proprietary software. In the case of the GPL, such a linked program must adopt the GPL.

MPEG-2 standard. This is where the major adaptations are necessary, because global data structures are heavily used.

Listing 2.1 shows the outline of mpeg2dec in pseudo–code. The most interesting part is the macroblock loop. The blocks are looped twice — once for decoding the DCT coefficients and a second time for performing the inverse DCT and for adding the prediction and coefficient data. This has to be reorganized for our purpose, as will be seen later. The implementation makes heavy use of global variables. During the decoding process, a lot of tables defined in the standard document are needed. Most of them belong to the VLD, and are therefore defined in `getvlc.h`. The more general tables like scanning– and quantizer matrices are defined in `global.h`. All the information extracted from the headers is stored in global variables, which are declared also in `global.h`. Listing 2.2 shows the most important global variables. Note that this list is by far not complete; a total of more than 100 variables are declared. Most of the variable names are self–explanatory and the complete semantics can be looked up in the appropriate section of the ISO/IEC 13818-2 document [12].

The MPEG-2 scalability feature allows for several independently coded layers. If a videosequence contains more than one layer, all the layer specific variables need to be saved separately. In addition to the global variables, the `layer_data` structure is therefore declared, which contains all these layer specific variables (see Listing 2.3).

mpeg2dec.c	main() function, initialization, command-line option processing
getpic.c	picture decoding routines
motion.c	motion vector decoding routines
recon.c	motion compensation routines
gethdr.c	header decoding routines
getblk.c	DCT coefficient decoding routines
getbits.c	bit level routines
getvlc.c	variable length decoding routines
idct.c	fast inverse discrete cosine transform
idctref.c	double precision inverse discrete cosine transform
global.h	declaration of global variables

Table 2.2: The most important files of the mpeg2dec application.

```
main() {
  <open input file>
  initialize_decoder();
  decode_bitstream() {
    for (all videosequences) {
      for (all pictures) {
        picture_data() {
          for (all slices) {
            for (all macroblocks) {
              <get macroblock mode>
              <decode motion vectors>
              <get macroblock pattern>
              for (all blocks) {
                <decode DCT coefficients>
              }
              motion_compensation() {
                <form predictions>
                for (all blocks) {
                  <perform inverse DCT >
                  <add prediction and coefficient data>
                }
              } /* end of motion_compensation() */
            }
          }
        } /* end of picture_data() */
        frame_reorder();
      }
    }
  } /* end of decode_bitstream() */
  <close input file>
}
```

Listing 2.1: Outline of mpeg2dec.

```
/* pointers to generic picture buffers */
unsigned char *backward_reference_frame[3];
unsigned char *forward_reference_frame[3];
unsigned char *auxframe[3];
unsigned char *current_frame[3];
unsigned char *substitute_frame[3];

/* non-normative variables derived from normative elements */
int Coded_Picture_Width;
int Coded_Picture_Height;
int block_count;
int Second_Field;
int profile;
int level;

/* normative derived variables (as per ISO/IEC 13818-2) */
int horizontal_size;
int vertical_size;
int mb_width;
int mb_height;
double bit_rate;
double frame_rate;

/* ISO/IEC 13818-2 sec. 6.2.2.1: sequence_header() */
int aspect_ratio_information;
int frame_rate_code;
int bit_rate_value;

/* ISO/IEC 13818-2 sec. 6.2.2.3: sequence_extension() */
int profile_and_level_indication;
int progressive_sequence;
int chroma_format;

/* ISO/IEC 13818-2 sec. 6.2.2.4: sequence_display_extension() */
int video_format;
int color_description;
int color_primaries;
int matrix_coefficients;
int display_horizontal_size;
int display_vertical_size;

/* ISO/IEC 13818-2 sec. 6.2.3: picture_header() */
int picture_coding_type;

/* ISO/IEC 13818-2 sec. 6.2.2.6: group_of_pictures_header()  */
int frame;
int closed_gop;
int broken_link;
```

Listing 2.2: Most important global variables.

```
/* layer specific variables */
struct layer_data {
  /* bit input */
  int Infile;
  unsigned char Rdbfr[2048];
  unsigned char *Rdptr;
  unsigned char Inbfr[16];

  /* from mpeg2play */
  unsigned int Bfr;
  unsigned char *Rdmax;
  int Incnt;
  int Bitcnt;

  /* sequence header and quant_matrix_extension() */
  int intra_quantizer_matrix[64];
  int non_intra_quantizer_matrix[64];
  int chroma_intra_quantizer_matrix[64];
  int chroma_non_intra_quantizer_matrix[64];

  int load_intra_quantizer_matrix;
  int load_non_intra_quantizer_matrix;
  int load_chroma_intra_quantizer_matrix;
  int load_chroma_non_intra_quantizer_matrix;

  int MPEG2_Flag;
  /* sequence scalable extension */
  int scalable_mode;

  /* picture coding extension */
  int q_scale_type;
  int alternate_scan;

  /* picture spatial scalable extension */
  int pict_scal;

  /* slice/macroblock */
  int priority_breakpoint;
  int quantizer_scale;
  int intra_slice;
  short block[12][64];
};
```

Listing 2.3: Layer specific variables.

2.3 Parallelization of the Algorithm

As described in Section 1.2, the application programmer has to expose the available parallelism to the DOL, which in turn tries to exploit it as much as possible. Depending on the structure and the complexity of the algorithm, parallelization may become a challenging task. This chapter deals with the parallelization of the MPEG-2 algorithm. The first section presents the methodology, which is then used in the following sections to split up the algorithm into a number of concurrent processes.

2.3.1 Methodology

One possible approach for parallel algorithm design consists of four stages: *Partitioning*, *Communication Analysis*, *Agglomeration* (also referred to as *Granularity Control*) and *Mapping*. The computation and the data operated on by this computation is first decomposed into smaller tasks. This can be done in one of two ways. Either the data is first decomposed and then computation is assigned to it (*domain decomposition*), or the computation is first decomposed and then the data is assigned to it (*functional decomposition*). The former way leads to data parallelism, and the latter to pipeline parallelism. After partitioning, the communication between the tasks has to be analyzed and appropriate data structures have to be defined. During the agglomeration phase the design has to be evaluated and tasks are combined wherever this improves performance or reduces cost. Finally, each task is mapped to a processor with the ambition of maximizing processor utilization and minimizing communication costs [20].

This general methodology is a good starting point, but several modifications must be made in order to use it for our purpose. The mapping stage can be omitted completely, as it is the main purpose of the DOL to help a programmer to find an efficient mapping. This is therefore done in a later stage. The second important difference stems from the complexity and the nested structure of the MPEG-2 algorithm. In order to fully explore the parallelism, a refined approach is needed.

In Section 2.3.2, the MPEG-2 decoding algorithm is therefore divided into *abstraction levels*. Each level consists of a *data unit*, a *computation* that is performed on it, and a *decoded data unit* as a result of the computation. For every abstraction level, both the possibilities of domain decomposition and functional decomposition are investigated. All decompositions are evaluated by the means of a communication analysis, i.e. the less inter–process communication is necessary, the better. It is important to note that in the analysis, all decompositions are treated completely independent from each other. Due to dependencies between the abstraction levels, however, not all of these decompositions can actually be implemented. This issue is addressed in the last stage of parallelization, the *partitioning* stage (Section 2.3.3), where several feasible decompositions are selected and combined.

The evaluation of a decomposition is an important step and may exhibit a trade–off between degrees of freedom for the optimization procedure and communication overhead. The finer the granularity, the more processes we have and therefore the more degrees of freedom are available to the mapping optimization procedure. However, a fine granularity normally leads to more communication overhead, which in turn may reduce performance (especially in data-intensive video processing applications). But as we will see later, there are also cases where a refinement of the granularity leads to *less* communication overhead.

2.3.2 The Abstraction Levels

As described in Section 2.1, the MPEG-2 video bitstream is organized in a syntactic hierarchy, which means that it consists of nested syntactic structures (see Figure 2.3 in Section 2.1.5). This organization naturally leads to six different data abstraction levels (Table 2.3) for which the following observations can be made.

i) Levels 1 – 4 provide no actual picture data but rather contain local and global header information and multiple instances of the subordinate structures. The picture level, for example, consists of header information (like I, P or B picture) and multiple instances of the slice structure. Therefore the computation associated with levels 1 – 4 are mainly header information extraction and it cannot be expected that they provide much potential for a functional decomposition. Wherever there is no computation associated with the data unit, the functional decomposition is therefore omitted.

ii) The potential for domain decomposition of levels 1– 5 will differ from level to level and must be examined carefully. Generally speaking, the more the data units are independent from each other, the less communication will be necessary. The general scheme of data parallelism obtained by a domain decomposition is shown in Figure 2.8. What remains is to define the output data type of the dispatch process (*dispatched data type*) and the input data type of the collect process (*collected data type*), as well as to implement the processes themselves.

1	**System Level**
	Data unit: MPEG-2 bitstream
	Demultiplex and decode video, audio and data stream
	Decoded data unit: stream of decoded pictures, audio samples and data
2	**Videosequence Level**
	Data unit: Coded video sequence bitstream
	Extract header information, decode GOPs (if available) and pictures
	Decoded data unit: Decoded videosequence (a number of decoded pictures)
3	**Picture Level**
	Data unit: Coded picture bitstream
	Extract header information and decode slices
	Decoded data unit: Decoded picture (raw 8-bit RGB or YUV data)
4	**Slice Level**
	Data unit: Coded slice bitstream
	Extract header information and decode macroblocks
	Decoded data unit: Decoded slice (one 16 pels wide row of a picture)
5	**Macroblock Level**
	Data unit: Coded macroblock bitstream
	Variable length decoding, block decoding and motion compensation
	Decoded data unit: Decoded MB (a 16×16 pels large section of a picture)
6	**Block Level**
	Data unit: Coded block bitstream (decoded DCT coefficients)
	Inverse Scan, Inverse Quantization, Inverse DCT
	Decoded data unit: Decoded block (8×8 chrominance or luminance values)

Table 2.3: Data abstraction levels.

iii) The computation at level 6 is quite complex, and a thorough investigation of possible functional decompositions is necessary. It is the only level where significant improvements due to pipeline parallelism are expected. There are two techniques of exploiting pipeline parallelism: first, a single process can simply be split up into several smaller processes, connected in series. Second, parts of the computation can be transferred to the upper adjacent level, where it is performed for all blocks of the current macroblock.

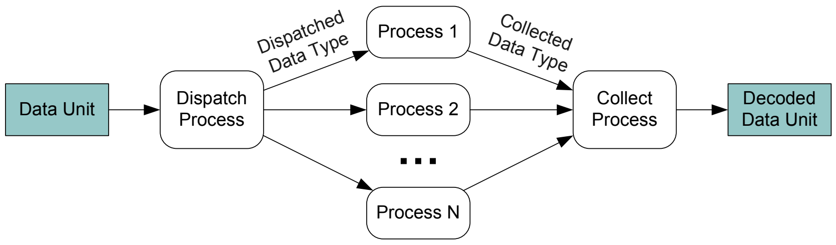


Figure 2.8: General domain decomposition scheme.

System Level

The top level data unit is an MPEG-2 bitstream. The stream is demultiplexed and the video, audio and data streams are decoded.

Domain Decomposition Decomposing the MPEG-2 bitstream into a video, audio and data stream is simple, and the corresponding decoding computation can easily be performed in parallel, as there is almost no interdependence. This decomposition does not follow the scheme of Figure 2.8 in that there is not an arbitrary number of N parallel processes, but instead one for video, one for audio and one for data. It would also be possible to decode several video streams in parallel, but both decompositions are not of our interest, as we are dealing with video only and assume just a single video stream at a time.

Videosequence Level

The data unit is a coded video bitstream, and the decoded data unit is a sequence of decoded pictures together with global information like picture size, aspect ratio information and frame rate. The videosequence header information is extracted (contains the aforementioned global information) and the GOPs (if available) and the pictures are decoded.

Domain Decomposition 1 Several GOPs can be processed in parallel and there is almost no interdependence. The main reason for introducing GOP headers is the independence — every GOP starts with an I picture and can therefore be decoded without knowledge of previous pictures. One difficulty is the fact that open GOPs are not completely independent from each other. Although this is no problem in the case of a random access (cf. Section 2.1.5), where the dependent B pictures can simply be skipped, this is not possible any more in the case of a decomposition,

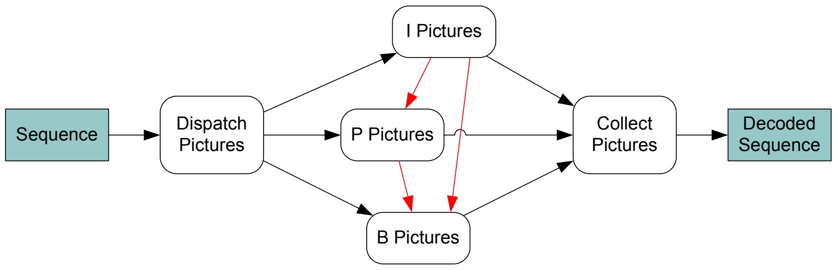


Figure 2.9: Picture level domain decomposition.

because the skipped pictures would never be decoded. This problem is addressed in Section 2.3.3. The main problem with this decomposition arises when there are no GOP headers in the bitstream. Assume that there are N processes decoding N GOPs in parallel. If a bitstream without GOP headers is applied to the decoder, only one out of N processes is busy, which is not tolerable. Either the decoder is clearly declared to require GOP headers (which theoretically makes it non–compliant with the MPEG-2 standard), or a mechanism is introduced which reads out every single picture header and autonomously forms groups of independent pictures.

Domain Decomposition 2 Decode N pictures in parallel, regardless of whether they are I, P or B pictures. However, it is obvious that predictive coding is inherently sequential on the picture level. The whole concept of compression by the means of predictive coding is based on the availability of the previously decoded picture. A huge amount of communication would be necessary in order to provide the processes with all the needed information, and every process must be connected to all other processes with two channels (one for each direction), which is clearly impractical.

Domain Decomposition 3 The third way of decomposing is a fixed amount of three parallel processes, one for each coding type. This concept is shown in Figure 2.9. The number of channels is significantly reduced, as it is clearly defined which process provides which neighbor with reference pictures. But the amount of communication is still huge, and the load will be poorly balanced, as the amount of I–, P– and B–type pictures and the resulting decoding effort is far from being even.

In summary, the only feasible decomposition is to process GOPs in parallel and, with some effort, the single GOPs can be made completely independent.

Picture Level

This level's data unit is a coded picture bitstream. The picture header information is extracted (I, P or B picture type, etc.) and the slices are decoded and put together to a decoded picture.

Domain Decomposition The slices of a picture can be processed in parallel. Slices are separated by a slice start code and are completely independent of each other. The only problem that must be solved at this level is the dependency on previously decoded pictures in the context of predictive coding. Since we avoid processing pictures in parallel, the preceding pictures can clearly be expected to be already decoded. The problem is theoretically reduced to a frame storage process that delivers parts of the stored pictures upon request. As will be shown later, the actual implementation can be done much more efficiently.

Slice Level

At the slice level the header information of the coded slice bitstream is extracted and the macroblocks are decoded.

Domain Decomposition The macroblocks can also be processed in parallel. The dispatch process can decode the incoming slices, form macroblock packets and send them to the macroblock processing units. All macroblocks of a picture are independent. As mentioned before, variable length codes are introduced at this level and the macroblocks are not preceded by a start code any more. As a consequence, for building a macroblock packet, the whole macroblock (including all its blocks, down to every single DCT coefficient) must be variable length decoded. The macroblocks need access to the previously decoded pictures during the motion compensation step. This problem is the same as for the decomposition into slices, and is addressed in Section 2.3.3.

Macroblock Level

The macroblock is the main motion compensation unit. This means that the macroblock header contains all the information needed for motion compensation, including the motion vectors. The actual prediction forming and combination of prediction and coefficient data is carried out on the block level. The computation on the macroblock level consists therefore only of extracting header information, decoding the variable length coded motion vectors and decoding the blocks.

Domain Decomposition In contrast to the slice level, where the number of macroblocks was unknown, it is now well–known how many blocks are contained in the macroblock. The number of blocks depends on the chrominance format, which is extracted from the sequence header[9]: There are 6 blocks for the 4:2:0 format, 8 blocks for the 4:2:2 format and 12 blocks for the 4:4:4 format.

Functional Decomposition The computation consists of three stages: variable length decoding, block decoding, and motion compensation (Figure 2.10). Although only the motion vectors are needed at this level, it is inevitable to perform the variable length decoding not only on the motion vectors, but also on all the block data. Since macroblocks are not preceded by a start code, this is the only way to find out where the current macroblock ends and the new one begins.

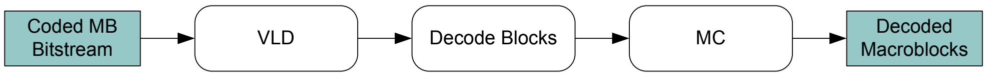


Figure 2.10: Macroblock level pipeline.

Block Level

The data unit of this level is a coded block bitstream which consists of DCT coefficients. These coefficients are decoded and transformed into an 8×8 block of luminance/ chrominance values. A further domain decomposition into single DCT coefficients makes no sense, as the inverse discrete cosine transform is a block operation that can only be applied to all of its coefficients. The computation however is now quite complex, and a functional decomposition is possible.

Functional Decomposition As described in Section 2.1.6, the video decoding process is naturally divided into the five functional units variable length decoding, inverse scan, inverse quantization, inverse DCT and motion compensation. As we are interested in coarse–grained parallelism only, we restrict ourselves to these five building blocks and do not consider splitting them up further. Motion compensation and variable length decoding is done on the macroblock level, therefore only inverse scanning,

[9]To be more precise, the chrominance format is contained in the sequence extension. For the sake of clarity, *header* stands here for all the information that is coded before the actual picture data and includes header, extension and user data.

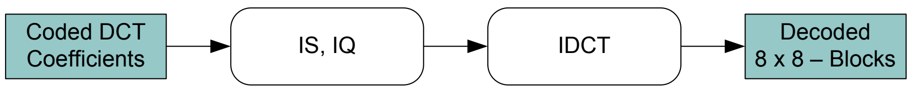


Figure 2.11: Block level pipeline.

inverse quantization and inverse discrete cosine transform remain. Figure 2.11 shows the resulting block level pipeline. According to execution time profiling results of [16], the three parts VLD, inverse DCT and motion compensation need roughly the same execution times[10]. Inverse scanning and inverse quantization consume much less time, and can therefore be integrated into a single process. It is even possible to perform the inverse scanning and inverse quantization together with the variable length decoding on the macroblock level. This reduces communication overhead, as will be seen later.

Summary

The algorithm is divided into six data abstraction levels, and four applicable data domain decompositions are identified. All of them follow the abstract general domain decomposition scheme of Figure 2.8 and can therefore be combined. Additionally, two possible functional decompositions are identified, one on the macroblock level and the other one on the block level. In the next section, some of the decompositions are selected and combined to build the process network.

2.3.3 Partitioning

Having a repertory of possible decompositions, partitioning the algorithm consists now merely of selecting one or several of them, and elaborating with the goal of balancing the pipeline and avoiding stalled processes as much as possible.

Three of the available domain decompositions are chosen to be implemented. For making it possible to compare different decompositions without having several completely different implementations, all three decompositions are combined, with each of them having a parameter N standing for the number of parallel processed entities. The selected domain decompositions are shown in Table 2.4.

[10]The basis of the measurement is the same sequential MPEG-2 decoding algorithm that is used here (MSSG mpeg2dec), running on a workstation.

1	**GOP**	N_1 groups of pictures are processed in parallel
2	**Macroblock**	N_2 macroblocks are processed in parallel
3	**Block**	N_3 blocks are processed in parallel

Table 2.4: Selected domain decompositions.

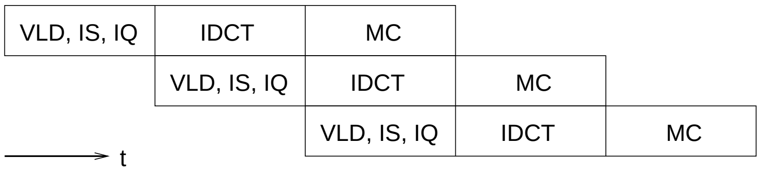


Figure 2.12: Selected Pipeline.

The reason for omitting the decomposition of pictures into slices is the following: according to the general domain decomposition scheme (see Figure 2.8), each decomposition is accompanied by a dispatch process. A process that dispatches slices could do nothing more than search the GOP buffer for the next slice start code and send the slice to the following processes. The slice dispatch process would therefore be idle most of the time, and the pipeline would be badly balanced. If, however, macroblocks are dispatched instead of slices, the variable length decoding from the motion vectors down to every single DCT coefficient has to be carried out, before sending it to the following processes. In terms of the pipeline, this is an advantage: while the current macroblock is processed, the variable length decoding of the next macroblock already takes place.

In addition to the three domain decompositions, both identified functional decompositions are chosen to be implemented, which leads to the pipeline shown in Figure 2.12. Recall that inverse scanning and inverse quantization are computationally inexpensive and do not require separate stages.

The resulting process network structure for the case $N_1 = N_2 = N_3 = 1$ (i.e. only pipeline parallelism, no data parallelism) is shown in Figure 2.13. The grayed out arrows indicate the possible data parallelism that can be obtained by increasing N_i. Note that there are four processes which do not perform actual decoding: On the one hand the processes *dispatch_gops* and *collect_gops*, which, additionally to dispatching and collecting groups of pictures, handle file input and output. *dispatch_blocks* and *collect_blocks*, on the other hand, are only here to allow to have several instances of the *transform_block* process. Since the *transform_block* process is computationally expensive, this decomposition is

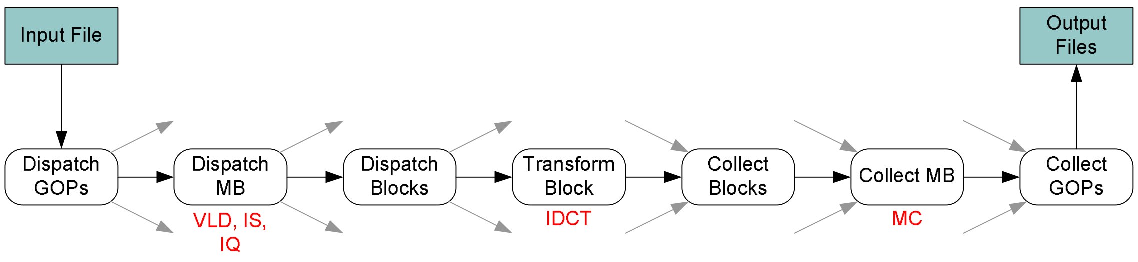


Figure 2.13: MPEG-2 process network structure.

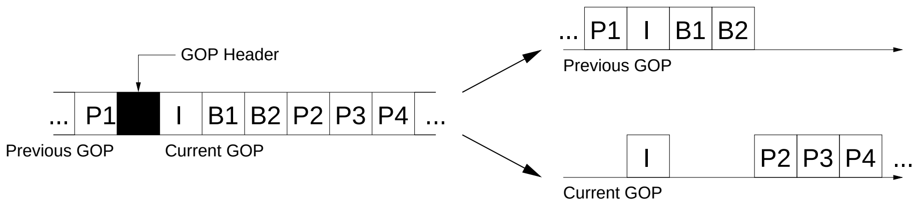


Figure 2.14: Illustration of how to make the groups of pictures independent by attaching B pictures to the previous GOP.

justified. Although it is possible to move the inverse scanning and inverse quantization to the *dispatch_blocks* process, this would significantly increase the communication overhead due to the necessary quantization matrices[11].

The remaining issue is how to treat open GOPs. As described in detail in Section 2.1.5, open GOPs contain pictures that depend on the previously decoded GOP. If an open GOP is to be processed in parallel with its preceding GOP, these dependent pictures cannot be decoded correctly. This problem can however easily be solved by "attaching" the first consecutive B pictures, which immediately follow the first I picture of the current GOP, to the previous GOP. This principle is illustrated in Figure 2.14. The two B pictures B1 and B2 are attached to the end of the previous GOP, since they depend on P1. Note that the I picture is needed as a reference picture by both the previous and the current GOP. When the decoded pictures are collected, one of the I pictures can simply be discarded.

Note that this partitioning does not need any frame storage process for motion compensation. Since there is no decomposition into single pictures, the *collect_mb* processes can simply keep the frame storage in their local memory. This is especially advantageous, because such a frame storage process can easily be overstrained and become the system's bottleneck. Additionally, frame storage processes are not suited very well

[11]The MPEG-2 standard allows that quantization matrices change, which means that they cannot simply be transmitted once and stored in the local memory of the process.

for the Kahn process network model. If the requests to the frame storage process arrive unevenly from the different processes, a lot of them will be stalled, waiting for the requested data to be delivered, while the storage process itself is waiting for a request from another channel.

2.4 Implementation

This section presents the final MPEG-2 decoder application for the SHAPES DOL. First, an overview is given on how to write an application according to the DOL programming model, taking the MPEG-2 decoder implementation as an illustrating example. Then, the configuration of the process network and the processes is described, followed by a brief user's guide. The last section then shows how the processes can be profiled in order to improve the implementation.

2.4.1 Structure of DOL Applications

An application written for the DOL consists of an XML file, which describes the process network, and a `src` subdirectory containing the source code of the processes, together with a predefined header file `process.h` and a global header file named `global.h`. The global header file contains only type definitions and preprocessor directives, as the programming model does not allow any global variables. Each process consists of at least one source– and one header file. Additional source files can be introduced, which then have to be included directly into the source code of a specific process using an `#include` directive. It is not possible to include the same source file in more than one process, as the functions would be defined twice. If a function has to be called from more than one process (a global error handling function, for instance), it must be defined with the `extern` keyword in all but one process. Table 2.5 shows the directory structure and files of the MPEG-2 decoder implementation. Note that the process network definition is called `example9.xml`, because the DOL package comes with 8 examples, and the MPEG-2 decoder can be considered as the 9th.

2.4.2 Configuration of the Process Network

Recall the general structure of the process network shown in Figure 2.13. By adjusting the parameters N_1, N_2 and N_3, diverse implementations can be realized without changing the process definitions. The parameterized process network is specified using *iterators*[12],

[12]An element of the XML specification which generates repeating structures.

example9.xml	Process network definition
src/process.h	Predefined process header file
src/global.h	Global type definitions
src/dispatch_gops.[ch]	Dispatch GOP process definition
src/dispatch_mb.[ch]	Dispatch macroblock process definition
src/vlc_mb.[ch]	VLC functions, included by dispatch_mb.c
src/motion_mb.c	MV decoding, included by dispatch_mb.c
src/dispatch_blocks.[ch]	Dispatch block process definition
src/transform_block.[ch]	Transform block process definition
src/idct_block.c	Fast Inverse DCT, included by transform_block.c
src/idctref_block.c	Reference IDCT, included by transform_block.c
src/collect_blocks.[ch]	Collect block process definition
src/collect_mb.[ch]	Collect macroblock process definition
src/predict_mb.c	Motion comp. functions, included by collect_mb.c
src/collect_gops.[ch]	Collect GOP process definition
src/store.c	File store functions, included by collect_gops.c

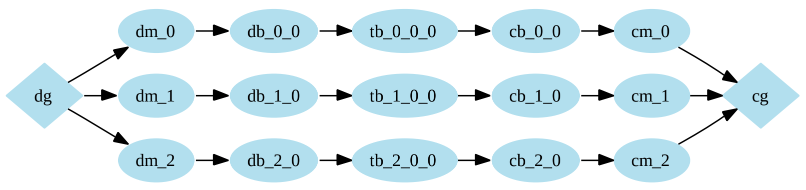
Table 2.5: MPEG-2 decoder files.

and all processes are written in a way that allows for an arbitrary number of connected channels. In order to set up an implementation, the process network *and* the process definitions must be configured with the desired parameters. This section explains how to configure the process network by adjusting the `example9.xml` file, and Section 2.4.3 shows how to configure the processes by adjusting the `global.h` file.

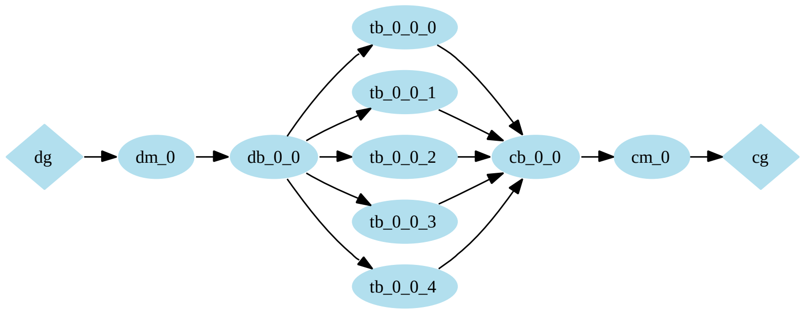
Listing 2.4 shows an excerpt from the process network definition file `example9.xml`. The three variables called `N1`, `N2` and `N3` can be set to the desired value. The listing additionally shows the first process instantiation to illustrate how these variables are used to set the range of iterators. Figure 2.15 shows three example networks. Note the abbreviations used for the process names, where *d* stands for dispatch, *c* for collect, *t* for transform, and *g*, *m*, *b* for GOP, macroblock, and block, respectively.

2.4.3 Configuration of the Processes

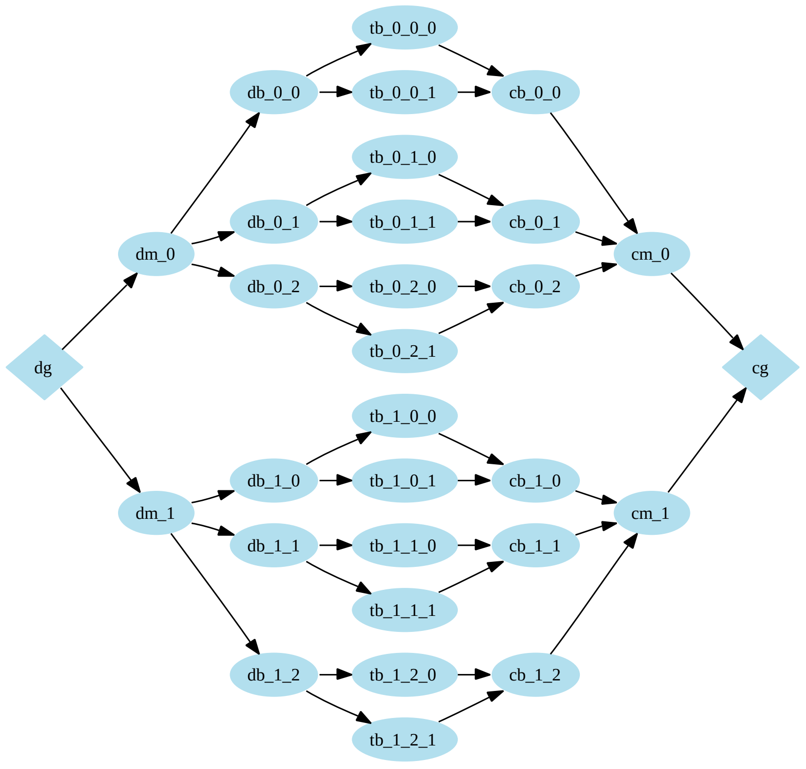
Listing 2.5 shows a small excerpt of the `global.h` file, where the parameters must be changed. The three constants should always be set to the same values as the variables of the process network definition. If the constants are smaller, processes are instantiated



a) $N_1 = 3,\ N_2 = 1,\ N_3 = 1$



b) $N_1 = 1,\ N_2 = 1,\ N_3 = 5$



c) $N_1 = 2,\ N_2 = 3,\ N_3 = 2$

Figure 2.15: Diverse implementations realized by varying the parameters N_i.

```
<!-- N1 is the number of GOPs processed in parallel -->
<variable name="N1" value="2"/>

<!-- N2 is the number of macroblocks processed in parallel -->
<variable name="N2" value="2"/>

<!-- N3 is the number of blocks processed in parallel -->
<variable name="N3" value="2"/>

<!-- instantiate processes -->
<process name="dispatch_gops">
  <iterator variable="i" range="N1">
    <port type="output" name="out">
      <append function="i"/>
    </port>
  </iterator>
  <source type="c" location="dispatch_gops.c"/>
</process>
(...)
```

Listing 2.4: Excerpt from the process network definition file.

```
  (...)
/* define the number of parallel processed gops, macroblocks, and blocks
   (used for looping all ports).
   IMPORTANT: this number must be less or equal to the number of defined
   ports in the process network specification.
   If there are more processes defined in than actually used,
   the program cannot quit correctly, as processes are started that are
   never detached. */

#define NUM_OF_GOP_PORTS          2
#define NUM_OF_MB_PORTS           2
#define NUM_OF_BLOCK_PORTS        2
  (...)
```

Listing 2.5: Excerpt from the global.h file.

which will never be used, and since these processes will also not be detached, the program cannot quit correctly. If the constants are larger than the process network definition variables, the processes try to write data to non–existing ports, which leads to runtime errors.

2.4.4 User's Guide

The following steps are required to build and run an implementation with a specific configuration:

1. Install and configure the DOL according to the DOL documentation and the SystemC library.
2. Edit the `example9.xml` file and set the variables `N1`, `N2` and `N3` to the desired values.
3. Edit the `src/global.h` file and set the constants `NUM_OF_GOP_PORTS`, `NUM_OF_MB_PORTS` and `NUM_OF_BLOCK_PORTS` to the same values.
4. Use the DOL to flatten the XML file and generate the SystemC source code (see DOL documentation for details).
5. Compile the generated source code.
6. Run the `sc_application` executable.
7. Specify the mpeg2 filename when prompted.

The MPEG-2 video file is then decoded and each frame is written to an uncompressed `TGA` file.

2.4.5 Profiling the Process Performance

Although the execution time of the functional simulation is not directly related to the execution time on a real SHAPES hardware, it is interesting to profile the simulation for two reasons:

1. For each process, the time spent in each function can be obtained. This is important for optimizing each process separately.
2. The times spent in the fire function of each process can be used to balance the pipeline.

At this point, a clear distinction between a process *definition* and a process *instance* must be made. Assume, as an example, a simple process network, where a producer process and a consumer process are connected by three identical processes for data processing. Since these three processes have all the same functionality, there is one process definition which is instantiated three times. We have therefore a total of 3 process definitions and 5 process instances.

The SystemC simulation can be profiled with the GNU `gprof` tool. For doing so, the application must be compiled with the `-pg` option. Each time the application is run, a profile data file results. These data files can then be processed by `gprof` in order to obtain a list of all functions together with statistical information. It is important to note, that the resulting list contains one entry for the `fire` function of each process *definition*, meaning that the execution time has yet to be divided by the number of process instances of the corresponding process definition. Figure 2.16 shows an excerpt from the profiling results of the MPEG-2 decoder. The columns of interest are *total ms per call*, which is the time spent in the fire function, including all sub–functions, and the *calls*, which is the total number of calls. By multiplying the total ms/call with the number of calls, the total time spent in the corresponding fire function can be obtained.

```
cumulative    self              self    total
  seconds   seconds    calls  ms/call  ms/call  name
  (...)
  2455.78     0.48  1910212     0.00     0.17  transform_block_fire__FP8_process
  2457.83     0.28   359487     0.00     0.42  collect_mb_fire__FP8_process
  2458.98     0.21   359487     0.00     0.65  collect_blocks_fire__FP8_process
  2461.92     0.10   359487     0.00     0.51  dispatch_blocks_fire__FP8_process
  2464.10     0.00       30     0.00    41.58  dispatch_gops_fire__FP8_process
  2464.10     0.00       30     0.00  2500.27  dispatch_mb_fire__FP8_process
  2464.10     0.00       27     0.00  7269.91  collect_gops_fire__FP8_process
  (...)
```

Figure 2.16: Excerpt from the result of profiling a SystemC simulation with gprof.

Figure 2.17 shows the process execution times for three different configurations (calculated and plotted with the Matlab program of Listing 2.6 using the profiling data of Figure 2.16). The dark green bars are the execution times obtained by profiling the configuration $N_1 = N_2 = N_3 = 1$. The light green and yellow bars are calculated by dividing the execution times by the number of process instances. It can be seen that the pipeline is balanced much better in the case of $N_1 = N_2 = N_3 = 2$. It is important to note that these execution times do not show the total speedup of a certain configuration, as no architecture is yet defined. In order to calculate the speedup, it must be known exactly how many processes are mapped to a single computation resource.

A detailed look at the profiling results additionally reveals why the *collect_gops* process is so time–consuming. Figure 2.18 shows that over 90 % of the time is spent in the `Write_Frame` function, which stores a frame on the harddisk. This function needs therefore to be optimized for improving the *collect_gops* process.

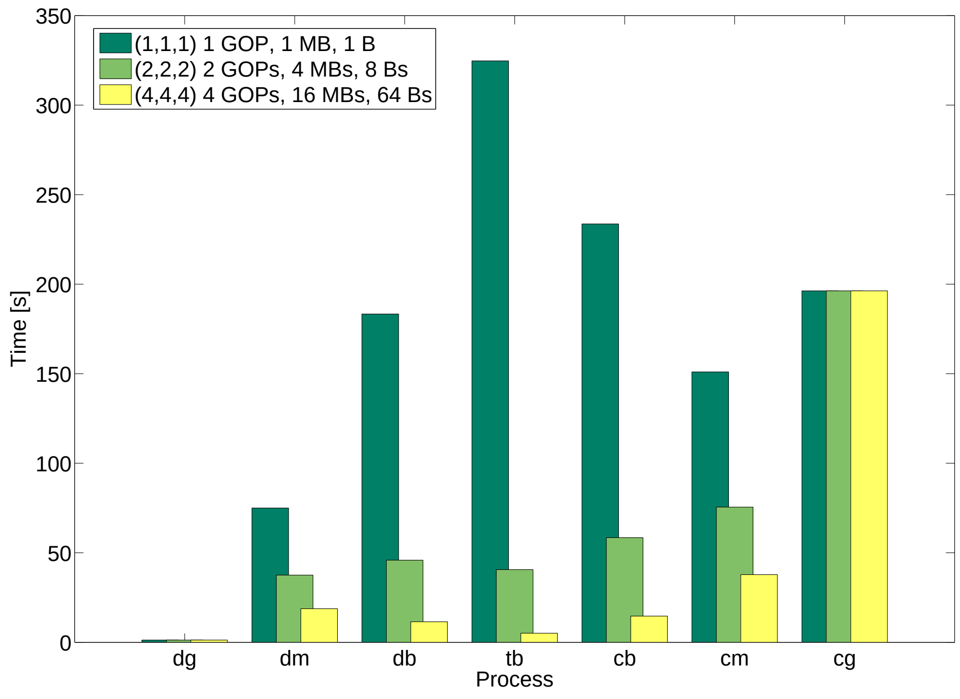


Figure 2.17: Process execution times for a short video containing 137 frames with a resolution of 640×480 pels on a Sun Blade 1500 with a 1 GHz UltraSPARC microprocessor.

```
index  %time    self descendents  name
  (...)
[14]     8.2    0.00      196.29  collect_gops_fire__FP8_process
                0.00      180.14  Write_Frame__FP8_processPPUci
                4.31       11.83  DOL_read__FPvT0iP8_process
                0.00        0.01  malloc
                0.00        0.00  _fprintf
                0.00        0.00  collect_gops_finish__FP8_process
                0.00        0.00  createPort
                0.00        0.00  free [307]
  (...)
```

Figure 2.18: Detailed view of the profiling results for the *collect_gops* process.

```
% exec_times.m

% Note the following convention of the indices for each vector
% [1 2 3 4 5 6 7]':
%       1 dispatch_gops
%       2 dispatch_mb
%       3 dispatch_blocks
%       4 transform_block
%       5 collect_blocks
%       6 collect_mb
%       7 collect_gops

num_of_instances111 = [1 1 1 1 1 1 1]';
num_of_instances222 = [1 2 4 8 4 2 1]';
num_of_instances444 = [1 4 16 64 16 4 1]';

% number of calls. IMPORT PROFILING DATA HERE.
num_of_calls = [30,30,359487,1910212,359487,359487,27]';

% time spent in function per call [ms]. IMPORT PROFILING DATA HERE.
ms_per_call = [41.58, 2500.27, 0.51, 0.17, 0.65, 0.42, 7269.91]';

% calculate the total time spent in process [s]
total_s = (num_of_calls.*ms_per_call)/1000;

bar([total_s./num_of_instances111 ...
     total_s./num_of_instances222 ...
     total_s./num_of_instances444], 1.5)

% Set plot properties here...
```

Listing 2.6: Matlab program to plot the execution times of Figure 2.17.

3 Profiling DOL Applications

A block diagram of the basic DOL software development flow is depicted in Figure 3.1. The application and architecture specification is required by the mapping stage, which tries to find an optimal mapping of the application onto the architecture. For evaluation of a mapping solution, the performance evaluation module can be used, which consists of a simulation framework and an analytical framework. However, as the performance evaluation by the means of a simulation is a complex process, it is desirable to acquire as much information as possible at the application level and to use the simulation framework only where it is necessary.

In this chapter, first the mapping–relevant parameters which are available at the application level are identified. The second section then presents the concept of tracing the application's communication, from which the parameters are then extracted. The last section presents the Java package `DOLProfiler` which implements these parameter extraction routines.

3.1 Mapping–Relevant Parameters

In order to find the mapping–relevant parameters which are available at the application level, the limitations of the application level must be identified. First, the hardware on

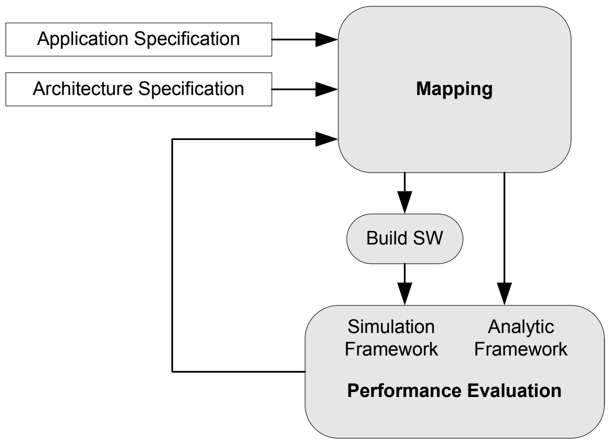


Figure 3.1: Simplified DOL software development flow.

which the application is to be executed, is not available. This means that the acquisition of parameters must fully rely on the SystemC simulation which simulates the multiprocessor platform on a single workstation and which is used for the functional verification. As a consequence, the measure of *time* cannot be used, because the execution times on the workstation do not reflect in any way the execution times of the processes on the real multiprocessor platform. Second, the acquisition of the parameters must not affect the application programmer in any way, i.e. the programming model has to remain untouched, which means that for obtaining the parameters, only the API function calls can be used. Recall that the four most important API functions are `read`, `write`, `init` and `fire`. The `init` and `fire` functions must be implemented by the process source, and are both called by the SystemC scheduler. These calls can be caught, but the only information obtained thereby is how many times the corresponding process is fired. The time spent in the `fire` function is insignificant due to the reasons mentioned before. The `read` and `write` functions are called by the application with the port name and the amount of bytes as an argument. If these calls are caught and stored during the simulation, significant information concerning communication can be obtained. However, it is important to note that the communication is nevertheless not necessarily accurately described by the `read` and `write` calls, and all parameters obtained based on it are still approximations, because of two reasons:

1. Since some processes may be executed much faster on the real multiprocessor platform, while others may not, the order of read and write accesses may change. As an example, assume a generator process connected via a channel to a consumer

process. Assume that in the simulation both processes are equally fast and therefore a `read` immediately follows each `write` call. If these processes are now executed on a multiprocessor platform and the generator is running much faster than the consumer, there may be several writes for each read. As a consequence, the FIFO fill level will constantly grow until the buffer size limit is reached and the generator is forced to slow down.

2. The SystemC scheduler is non–preemptive, i.e. a process is executed until it either performs a blocking read/write or finishes its `fire` function. The more the scheduling on the multiprocessor platform differs from this principle, the less significant the parameters will be.

Based on these observations, for each channel, the parameters of Table 3.1 can be derived. It is however very important, that all these parameters can only be obtained for a given set of input data (in the MPEG-2 decoder case: for a given video stream). For having realistic parameters, realistic input data is necessary, and the significance of the parameters is low if the workload caused by the input data heavily varies[1]. The TATD and the MFL are identified as the most important ones, as the TATD can be used to prevent bottlenecks by balancing the communication load amongst all buses, and the MFL can be used to prevent overflows by balancing the buffer size needs amongst the memory cells.

3.2 Tracing a DOL Application

For calculating the parameters presented before, all calls to the API functions `read` and `write` must be traced during a simulation. This is done by extending the SystemC source code such that a single line is written into a text file at each API function call. Obviously, this file may become very large. In the case of the MPEG-2 decoder, experience showed that a small video sequence of about 20 low resolution images already results in a profiling file of 50MB and more. However, it will often be sufficient to profile such a short simulation, especially if the communication is expected to happen in repeated patterns (as for example, in the case of the MPEG-2 decoder, where the communication pattern will not change very much from one group of pictures to the other).

[1]It is of course possible to write an application in a way that prevents these variations, but this is not required by the programming model and can therefore not be expected.

TATD	Total amount of transferred data
MFL	Maximum buffer fill level
$N_{\mathrm{w,tot}}$	Total number of write accesses
$N_{\mathrm{r,tot}}$	Total number of read accesses
N_{blk}	Number of blocking read accesses
N_{ovf}	Number of write accesses that lead to an overflow
P_{blk}	Percentage of reads that are blocking
P_{ovf}	Percentage of writes that lead to an overflow
$S_{\mathrm{w,avg}}$	Average write chunk size
$S_{\mathrm{r,avg}}$	Average read chunk size
$S_{\mathrm{w,max}}$	Maximum write chunk size
$S_{\mathrm{r,max}}$	Maximum read chunk size

Table 3.1: Parameters that can be derived for each channel, based on the API function calls.

3.2.1 Extension of the Source Code for Tracing the API Calls

As described in Section 1.2.2, the DOL automatically generates SystemC source code from the application specification, which can be run on a single workstation. The structure of this SystemC application, which is used for a functional verification and is now to be extended for generating the trace file, is presented in this section.

The DOL generates several files: a makefile, a file containing the main SystemC module, and a wrapper class for each process, which is instantiated in the main SystemC module. Let us first have a look at the main SystemC module. The file is called `sc_application.cpp` and consists of a single class `sc_application`, which is derived from the class `sc_module` (see SystemC documentation for a description of the `sc_module` class, available e.g. from [7]). For each process, the class contains an instance of the corresponding wrapper class. Additionally, the class contains all necessary FIFOs and the connections of processes to channels. The important part for profiling, are the `thread_` functions, one for each process. Listing 3.1 shows, as an example, the `thread_` function of the `dispatch_gops` process of the MPEG-2 decoder example. It can be seen that, if `INCLUDE_PROFILER` is defined, a line is written to the trace file before and after each time the scheduler calls the processes `fire` function. Listing 3.2 shows an excerpt of the wrapper class of the `dispatch_gops` process. It shows that each time, the process

```
    /*
       This function is called by the SystemC scheduler
       and repeatedly calls the fire function until
       the process has detached itself
    */
    void thread_dispatch_gops()
    {
        while (!dispatch_gops_ins.isDetached())
        {

#ifdef INCLUDE_PROFILER
            if (profiler_output_file != NULL)
                fprintf(profiler_output_file,
                        "%u dispatch_gops started.\n",
                        profiler_event_counter++);
#endif

            dispatch_gops_ins.fire();

#ifdef INCLUDE_PROFILER
            if (profiler_output_file != NULL)
                fprintf(profiler_output_file,
                        "%u dispatch_gops stopped.\n",
                        profiler_event_counter++);
#endif

            eventList.push_back(&dispatch_gops_event);
            sched_event.notify();
            wait(dispatch_gops_event);
        }
    }
```

Listing 3.1: Sample thread function with profiling extension.

calls the API function `write`, a line is written to the trace file, containing the name of the process, the event (write, in this case), the port, and the length (number of written bytes). The port is needed to determine the channel onto which the data is written. Since profiling slows down the simulation time, it needs to be turned on explicitly by defining `INCLUDE_PROFILER`[2].

3.2.2 Structure of the Trace File

The structure of the resulting trace file is shown in Figure 3.2. Note that the lines beginning with a `c` describe the connection of channels to the ports of processes. Each port is represented by a unique identifier, which makes it possible later to relate the read and write events directly to the channels. After the description of the connections, the actual trace is listed. Each line begins with an event number, followed by the name of the process,

[2]In the case of the GNU compiler, this can be done by calling `g++` with the `-DINCLUDE_PROFILER` option.

```
/*
   DOL_write is called by the fire function of the process, for
   writing data to a specific port (which is connected to a
   specific channel).
*/
static inline int DOL_write(void *port,
                            void *buf,
                            int len,
                            DOLProcess *process)
{
    sc_port<write_if> *write_port
        = static_cast<sc_port<write_if> *>(port);
    char *str = static_cast<char*>(buf);

#ifdef INCLUDE_PROFILER
    (static_cast<dispatch_gops_wrapper *>
        (process->wptr))->addToProfile("w", port, len);
#endif

    while (len-- > 0) {
        (*write_port)->write(*str++);
    }
}

(...)

#ifdef INCLUDE_PROFILER
/*
   addToProfile writes the profiling event to the trace file. The address
   of the port is used as a unique identifier, and the relation between
   the ports and the channels is stored at the begin of the trace file.
*/
void dispatch_gops_wrapper::addToProfile(const char *event,
                                         void *port,
                                         int length)
{
    if (profiler_output_file != NULL) {
        fprintf(profiler_output_file,
                "%u %s %s %p %d\n",
                profiler_event_counter++,
                _uniqueName, event, port, length);
    }
}
#endif
```

Listing 3.2: The `write` function of the `dispatch_gops` process with the profiling extension.

the event type (`r` for "read" or `w` for "write"), the address of the port, and the amount of bytes read/written. The begin and end of each `fire` is marked with the keywords `started` and `stopped`.

```
c gm_channel_0 307200 o dispatch_gops ffbee5dc \
                      i dispatch_mb_0 ffbee9d4
c mg_channel_0 1269760 o collect_mb_0 ffbeebe4 \
                       i collect_gops ffbee7c4
c mb_channel_0_0 307200 o dispatch_mb_0 ffbee9ac \
                        i dispatch_blocks_0_0 ffbeedf4
c bm_channel_0_0 307200 o collect_blocks_0_0 ffbef004 \
                        i collect_mb_0 ffbeebbc
c bt_channel_0_0_0 307200 o dispatch_blocks_0_0 ffbeedcc \
                          i transform_block_0_0_0 ffbef214
c tb_channel_0_0_0 307200 o transform_block_0_0_0 ffbef1ec \
                          i collect_blocks_0_0 ffbeefdc
0 dispatch_gops started.
1 dispatch_gops stopped.
2 collect_gops started.
3 collect_gops r ffbee7c4 1
4 dispatch_mb_0 started.
5 dispatch_mb_0 r ffbee9d4 1
6 collect_mb_0 started.
7 collect_mb_0 r ffbeebbc 1
8 dispatch_blocks_0_0 started.
9 dispatch_blocks_0_0 r ffbeedf4 1
10 collect_blocks_0_0 started.
11 collect_blocks_0_0 r ffbeefdc 1
12 transform_block_0_0_0 started.
13 transform_block_0_0_0 r ffbef214 1
14 dispatch_gops started.
17 dispatch_gops w ffbee5dc 1
18 dispatch_gops w ffbee5dc 64
19 dispatch_gops w ffbee5dc 256
20 dispatch_gops w ffbee5dc 4
21 dispatch_gops w ffbee5dc 3398
22 dispatch_gops w ffbee5dc 4
23 dispatch_gops w ffbee5dc 2446
24 dispatch_gops w ffbee5dc 4
25 dispatch_gops w ffbee5dc 873
26 dispatch_gops w ffbee5dc 4
27 dispatch_gops stopped.
28 dispatch_mb_0 r ffbee9d4 64
29 dispatch_mb_0 r ffbee9d4 256
30 dispatch_mb_0 r ffbee9d4 4
31 dispatch_mb_0 r ffbee9d4 3398
32 dispatch_mb_0 w ffbee9ac 1
33 dispatch_mb_0 w ffbee9ac 64
34 dispatch_mb_0 w ffbee9ac 172
35 dispatch_mb_0 w ffbee9ac 1536
 (...)
```

Figure 3.2: Excerpt from a resulting trace file.

3.3 Extraction of the Parameters

Having a trace of the API function calls at hand, the parameters can now be extracted by parsing the trace line–by–line and keeping track of all the FIFOs. Let us denote a write access of n bytes to a specific FIFO at time step t as $\mathit{access}\,[t] = n$, a read access as $\mathit{access}\,[t] = -n$, and assume that each time step $t \in [0,\ t_{\mathrm{end}}]$ corresponds to one single read or write access. Additionally, assume that at the end of the trace, the total amount of read data equals the total amount of written data, i.e. all FIFO buffers are empty at the begin and at the end of the simulation. The fill level of the FIFO buffer at a time step t can be calculated with

$$\mathit{fl}\,[t] = \sum_{k=0}^{t} \mathit{access}\,[k] \ . \tag{3.1}$$

Note that a blocking read leads to a negative fill level, which is resolved by the following write accesses. The parameters of Section 3.1 can then be expressed as

$$\begin{aligned}
\mathrm{TATD} &= \frac{1}{2}\sum_{t=0}^{t_{\mathrm{end}}} |\mathit{access}\,[t]| \\
\mathrm{MFL} &= \max_{0\le t\le t_{\mathrm{end}}} \left(\sum_{k=0}^{t} \mathit{access}\,[k] \right) \\
S_{\mathrm{w,avg}} &= \frac{\mathrm{TATD}}{N_{\mathrm{w,tot}}} \\
S_{\mathrm{r,avg}} &= \frac{\mathrm{TATD}}{N_{\mathrm{r,tot}}} \\
S_{\mathrm{w,max}} &= \max_{\forall\, t:\, \mathit{access}[t]>0} \{\mathit{access}\,[t]\} \\
S_{\mathrm{r,max}} &= \max_{\forall\, t:\, \mathit{access}[t]<0} \{|\mathit{access}\,[t]|\} \ .
\end{aligned} \tag{3.2}$$

The total number of read/write accesses ($N_{\mathrm{r,tot}}$ and $N_{\mathrm{w,tot}}$) can be obtained simply by counting the read/write events, the number of blocking read accesses N_{blk} by counting the number of read accesses which lead to a negative fill level, and the number of overflows N_{ovf} by counting the number of write accesses which lead to a fill level that exceeds a predefined maximum buffer size. The percentages can then be expressed as

$$\begin{aligned}
P_{\mathrm{blk}} &= \frac{N_{\mathrm{tot}}}{N_{\mathrm{blk}}} \cdot 100\ \% \\
P_{\mathrm{ovf}} &= \frac{N_{\mathrm{tot}}}{N_{\mathrm{ovf}}} \cdot 100\ \% \ .
\end{aligned}$$

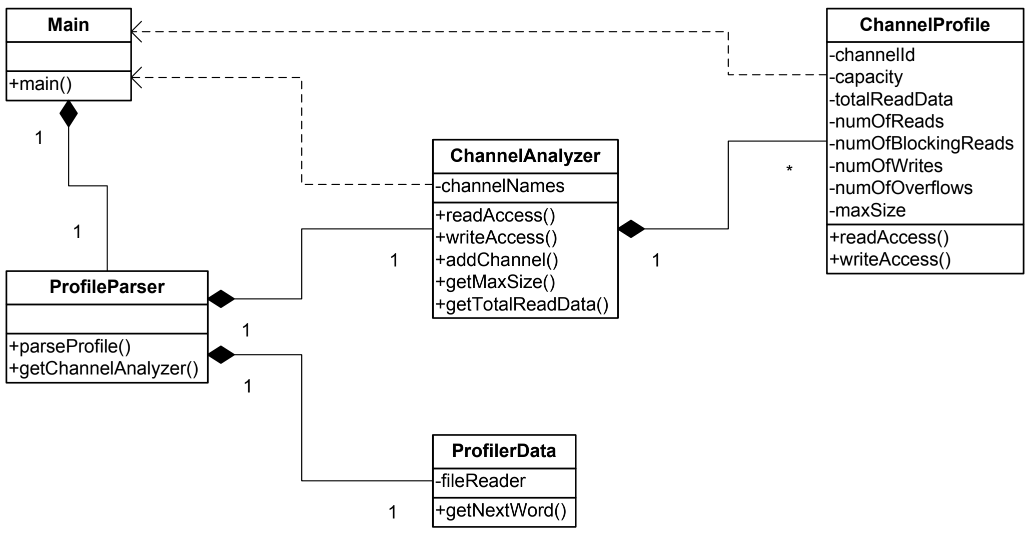


Figure 3.3: Class diagram of the `DOLProfiler` package.

3.4 The `DOLProfiler` Package

The `DOLProfiler` package parses a trace file line–by–line, keeps track of the FIFOs and calculates the parameters according to Section 3.3.

3.4.1 Structure

A class diagram of the `DOLProfiler` package is shown in Figure 3.3. The *ChannelProfile* represents a single channel. It keeps track of the current fill level and can be updated by calling the `readAccess()` and `writeAccess()` functions. Additionally, the parameters of the channel are stored in this class and updated with each read and write access. The *ChannelAnalyzer* contains the ChannelProfiles for all channels of a process network. The read and write accesses are forwarded to the ChannelProfiles. The *ProfilerData* contains the file access routines and delivers the next word upon request. The *ProfileParser* reads the words one–by–one from the ProfilerData and updates its instance of the ChannelAnalyzer. Additionally, a list of process, channel, and port names is stored and the relation between the ports of the processes and the channels is resolved. In the end, the parameters depend only on the channel names, and not on the process names anymore. The *Main* class can be used to parse a trace file and print out a summary of the ChannelAnalyzer. In the mapping stage, the Main class will not be used anymore, but the ProfileParser class will be imported directly.

3.4.2 Usage

Main program

For printing the parameters extracted from a trace file to the standard output, the `main` function of the `Main` class can be called with

```
java dolprofiler.Main <filename>
```

where `<filename>` must be replaced by the name of the trace file. A sample output containing some of the parameters is shown in Figure 3.4.

```
----- Channel Analyzer Report -----
<channel name> <MFL> <TATD> <Ntot> <Pblk>
<gm_channel_0> 71981 89753 45 2
<mg_channel_0> 126956 2158270 103 17
<gm_channel_1> 70919 70921 31 3
<mg_channel_1> 1269807 1650442 79 2
<mb_channel_0_0> 308625 5164436 11816 0
<bm_channel_0_0> 20091 9624979 11815 9
<mb_channel_0_1> 308625 5161363 11813 0
<bm_channel_0_1> 16782 9618835 11813 5
<mb_channel_1_0> 308625 3737038 8538 0
<bm_channel_1_0> 309396 6965710 8538 11
<mb_channel_1_1> 307914 3724750 8530 0
<bm_channel_1_1> 309870 6941133 8529 1
<bt_channel_0_0_0> 307383 4337687 34949 0
<tb_channel_0_0_0> 994 4337686 34948 16
<bt_channel_0_0_1> 307383 4337686 34948 0
<tb_channel_0_0_1> 1491 4337686 34948 0
<bt_channel_0_1_0> 307383 4334941 34927 0
<tb_channel_0_1_0> 994 4334941 34927 16
<bt_channel_0_1_1> 307383 4334941 34927 0
<tb_channel_0_1_1> 1491 4334940 34926 0
<bt_channel_1_0_0> 307383 3139297 25291 0
<tb_channel_1_0_0> 95899 3139296 25290 16
<bt_channel_1_0_1> 307383 3139060 25288 0
<tb_channel_1_0_1> 95662 3139060 25288 0
<bt_channel_1_1_0> 307383 3128317 25207 0
<tb_channel_1_1_0> 91426 3128316 25206 16
<bt_channel_1_1_1> 307383 3128080 25204 0
<tb_channel_1_1_1> 91189 3128080 25204 0
```

Figure 3.4: Sample output of the `dolprofiler.Main.main()` function.

Imported ProfileParser

If the profiler is to be used in the context of another application (as, for instance, in the context of the mapping optimization application), it is more convenient to import the

ProfileParser class and use the resulting parameters directly. In order to do so, first the ProfileParser and the ChannelAnalyzer classes must be imported:

```
import dolprofiler.ProfileParser;
import dolprofiler.ChannelAnalyzer;
```

Inside the body of a function, the ProfileParser object can then be created, with the filename as an argument to the constructor:

```
ProfileParser profileParser = new ProfileParser(filename);
```

The trace file can then be parsed. This can take some time since the trace file can easily be as large as 100 MB or more, so this should be done only once, during the initialization phase:

```
profileParser.parseProfile();
```

After the trace file is parsed, the parameters are ready. A ChannelAnalyzer object can be obtained and the parameters can be read out:

```
ChannelAnalyzer chAnalyzer = profileParser.getChannelAnalyzer();
int TATD = chAnalyzer.getTotalReadData(chId);
int MFL = chAnalyzer.getMaxSize(chId);
```

The channel id *chId* is the number of the channel, specified by the order of appearance at the beginning of the trace file (if, for instance, there are 4 lines in the trace file starting with a `c`, the channels are numbered from 0 to 3).

4 The Mapping Stage

The profiling results derived in the previous chapter can now be used for computing a mapping of the application onto an architecture. As described previously, the final mapping will be found by making use of the simulation framework, but it is desirable to make the best use of the parameters acquired by application level profiling. This chapter therefore deals with the challenge of optimizing the mapping without having detailed simulation results.

The first section gives an overview of the mapping stage and the problem of two–dimensional optimization, and a brief introduction to evolutionary algorithms, the PISA interface and EXPO. From the first section, it will become clear that there are three main issues, which are addressed in sections 2, 3 and 4, namely specification of the problem, representation of the solutions and analysis of the solution based on performance data and objective values.

4.1 Overview

The goal of mapping optimization is to find an optimal mapping of processes onto computation resources and channels onto communication resources with respect to a given set of objective values. The objective values are described in detail in Section 4.4, but by considering two examples it is intuitively understandable that the mapping of processes and the mapping of channels are two competing objectives.

Example 1 *Assume that all processes are mapped to a single processor, even though many processors may be available on the architecture. In that case, the channel mapping is clearly optimal, as the communication can all take place inside the processor and no external communication resource is required. On the other hand, the process mapping is very bad, as a single computation resource is heavily loaded whereas all the other ones are not used at all.*

Example 2 *Consider now the contrary situation. Each process is mapped to its own computation resource. In this case, the process mapping is very well, but there is a lot of external communication necessary.*

The situation becomes even more difficult if we take into account that

- some processes perform more time–consuming calculations, while others are idle most of the time,
- some processes may be executed more efficiently on certain computation resources,
- some channels may transfer much more data than others,
- some communication resources have a larger bandwidth than others,
- some channels may need larger FIFO buffers in order to prevent overflows, while the FIFOs of other channels may be empty most of the time.

Clearly, a systematic method for finding optimal solutions is needed, especially since the problem must be scalable to very complex applications and large architectures.

4.1.1 Solving Multi–Objective Optimization Problems

All multi–objective optimization problems have in common that there is no single optimal solution. Instead, the final solution must be chosen from a set of optimal solutions, which means that the trade–off between the objectives must be considered. To illustrate this, recall Examples 1 and 2. Both of them are optimal, meaning that there is no other solution which is better with respect to one objective value and at least equal with respect to the other. A non–optimal solution can easily be constructed:

Example 3 *Assume that all processes are mapped to a single computation resource, but one of the channels is mapped to an external communication resource. This is obviously not optimal, as the communication can be improved without worsening the computation.*

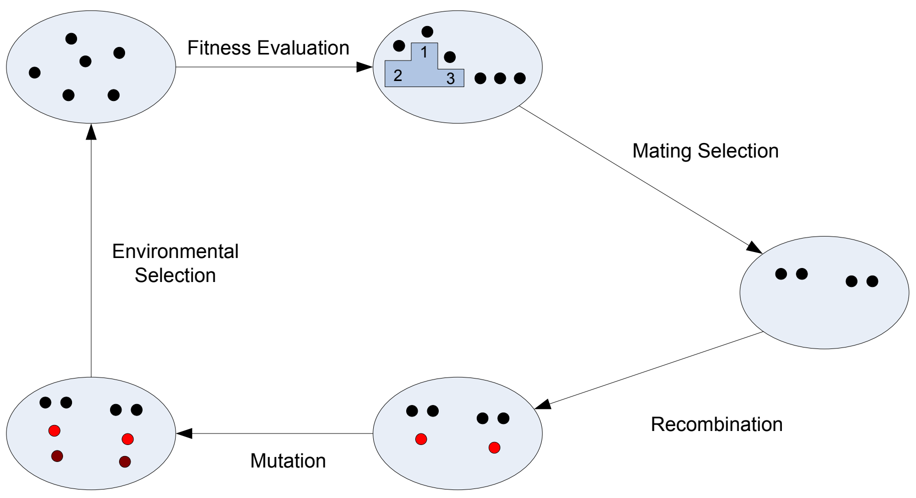


Figure 4.1: Simple evolutionary algorithm.

The solution of Example 3 is *dominated* by the solution of Example 1, meaning that the solution of Example 1 is equal with respect to one objective value (namely the computation), and better with respect to the other (the communication). This concept can be extended to an arbitrary number of objective values, see for example [21]. The main challenge of multi–objective problems is therefore to find all non–dominated solutions (so called *pareto optimal* solutions). Having all pareto optimal solutions at hand, the most appropriate one can be chosen.

4.1.2 Evolutionary Algorithms

Whenever the search space is too big for evaluating all possible solutions, search heuristics must be used. Evolutionary Algorithms are search heuristics inspired by biological evolution. Each solution is represented by a gene. A set of genes, called *population*, is iteratively improved by variation and selection. A possible implementation of an evolutionary algorithm is shown in Figure 4.1. Each gene is evaluated, and the best genes are selected for mating, which means that two of them (the *parent genes*) are recombined and result in a *child gene*. Some of the genes of the resulting population are then mutated, meaning that they are slightly changed, which simulates errors in the genetic material and may lead to better solutions. Since the population now contains more genes than before the mating step, an environmental selection is performed where only the fittest genes survive. This reduces the size of the population to the size of the initial population, and the whole process can be iteratively repeated.

4.1.3 The PISA Interface and EXPO

The *platform and programming language independent interface for search algorithms* (PISA) is a text–based interface that separates the problem–specific part of an optimizer from the problem–independent part [22]. The problem–specific parts are evaluation of solutions and generation of new solutions by variation of some selected solutions. The selection step itself is problem–independent and is therefore separated from the evaluation and variation via the text–based interface. This interface allows to have a library of optimization algorithms (*selectors*) and a library of optimization problems (*variators*), which can be combined arbitrarily. For the mapping optimization implementation, using PISA has two important advantages:

1. The optimization algorithms of the selector library can directly be used which saves implementation time and allows to run the optimization with different algorithms by simply exchanging the selector.
2. An existing optimization problem can be taken from the library and adapted for our purpose.

For a list of available optimization problems and algorithms, refer to the PISA web-page [23]. One of the available variators is EXPO, which is used for the optimization of a network processor architecture [24, 25]. The Java source code of EXPO is available, which makes it possible to use EXPO for our purpose. EXPO defines three Java interfaces which are implemented by problem–specific classes, and in order to use EXPO for our specific mapping problem, only these three classes need to be implemented from scratch. The use of EXPO has the advantage that the communication with the selector via the PISA interface is already implemented and extensively tested. Additionally, EXPO provides a graphical user interface which simplifies controlling the simulation, displaying single solutions, and plotting the whole population. Table 4.1 shows a list of the classes and interfaces, together with a short description.
Each of the following three sections addresses one of the three classes. In each section, the necessary concepts are elaborated, the interfaces are presented and a short description of the implementation is given.

4.2 System Specification

The first issue to address is the system specification. EXPO allows to read in the specification from a text file during the initialization, which has the advantage that the

1	**ShSpecification** class implements **Specification** interface. Contains a model of the specification, i.e. architecture, application, execution times, performance data, etc. (Section 4.2)
2	**ShGene** class implements **Gene** interface. Contains the representation of a gene incl. the variation operators. (Section 4.3)
3	**ShAnalyzer** class implements **Analyzer** interface. Contains methods that calculate the objective values of a gene. (Section 4.4)

Table 4.1: The three interfaces which are implemented by the three problem–specific classes.

specification can be changed without re–compiling EXPO. The following requirements can be identified:

1. In order to test the mapping stage, an example architecture must be chosen. An example architecture is proposed in Section 4.2.1.
2. Since the execution times of the processes on the different resource types is a very important parameter for the mapping optimization but it is impossible to obtain them via profiling of the SystemC simulation, an estimation of the execution times is necessary. A way of estimating the execution times is proposed in Section 4.2.2.
3. The architecture allows for two computation resources to be connected via a series of buses, which implies that a channel is mapped to a *communication path* rather than to a single bus. This leads to the problem of finding all possible paths of a given architecture. An algorithm that lists all possible paths is proposed in Section 4.2.3.
4. A data structure is needed, which contains a model of the complete specification, i.e. the application, the architecture, execution times and the performance data obtained through profiling. We call this structure the *system model* and its implementation the `sysmodel` package. The system model is described in Section 4.2.4.
5. Since EXPO requires the whole specification to be in a single text file, a method is needed that gathers all the information available from XML specifications, profiling, and textual specifications and writes the EXPO specification file. This application is called the *EXPO specification file generator* (`expogen`) and is presented in Section 4.2.5.

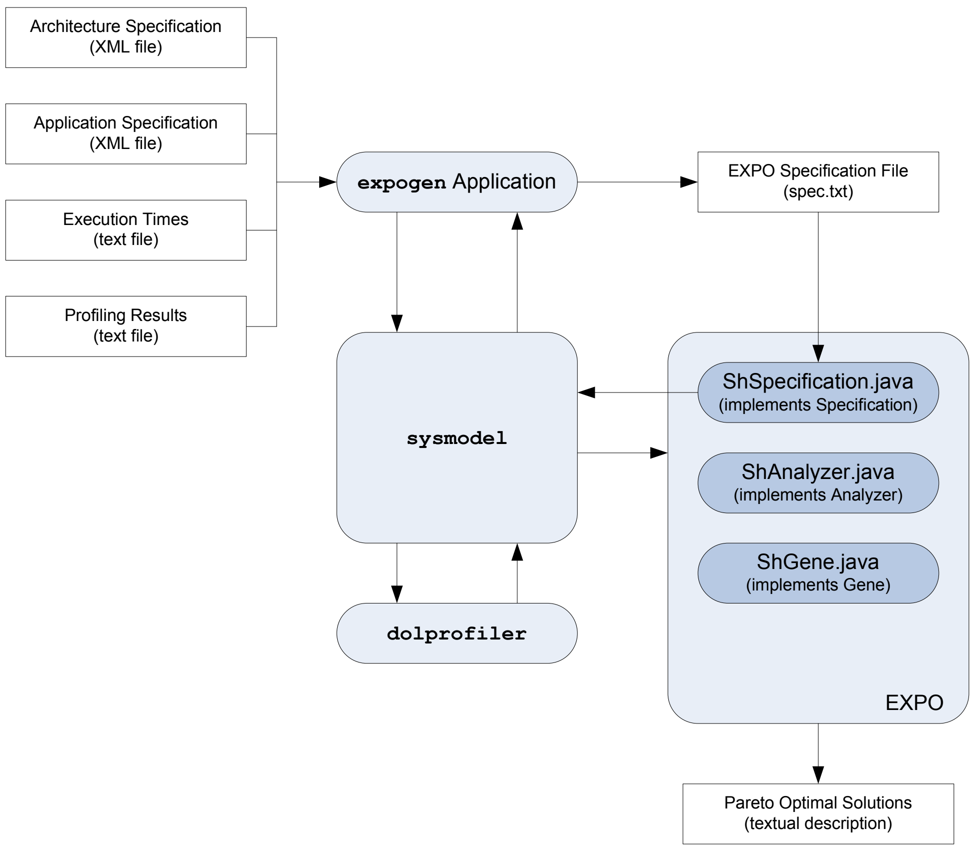
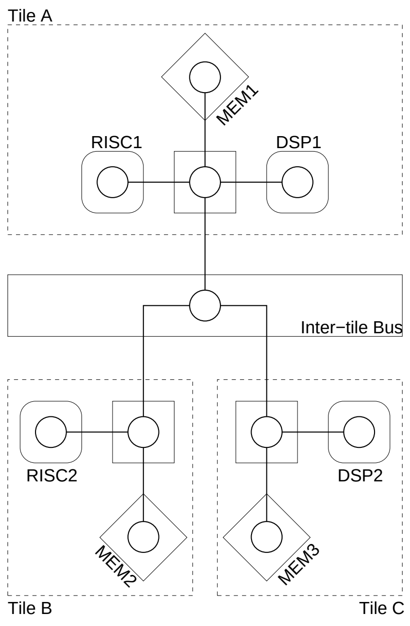

Architecture Specification (XML file)
Application Specification (XML file)
Execution Times (text file)
Profiling Results (text file)
expogen Application
EXPO Specification File (spec.txt)
sysmodel
ShSpecification.java (implements Specification)
ShAnalyzer.java (implements Analyzer)
ShGene.java (implements Gene)
EXPO
dolprofiler
Pareto Optimal Solutions (textual description)

Figure 4.2: Dependencies between the system model and EXPO.

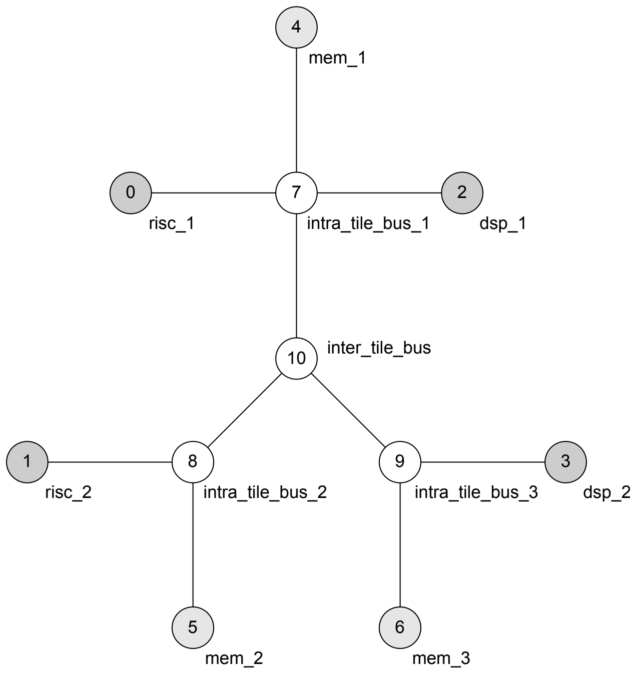
Figure 4.2 illustrates how these modules are put together. Besides the data structure of the model, the `sysmodel` package contains all routines that are necessary to read the XML files containing the application and the architecture, to write the EXPO specification file and to read in the EXPO specification file. This means that the `sysmodel` package is used for generating the EXPO specification file (as a part of `expogen`), as well as for reading the EXPO specification file (as a part of the `ShSpecification` class). Additionally, the `sysmodel` package makes use of the `dolprofiler` in order to obtain the performance data from the SystemC profiling results.

4.2.1 Example Architecture

In order to test the prototype of the mapping stage, an example architecture is needed. It is kept as simple as possible, but nevertheless reflects the SHAPES hardware approach in that it consists of several tiles which are connected to each other. The proposed architecture is shown in Figure 4.3a. The architecture is modeled as an undirected graph, where each



(a) Architecture



(b) Graph

Figure 4.3: Example architecture for testing the mapping stage.

node represents a computation, memory, or bus resource. The graph of the example architecture is shown in Figure 4.3b.

4.2.2 Estimation of the Execution Times

Section 2.4.5 showed how to measure the execution times of an application simulated on a single workstation with SystemC. Although these execution times do not represent the execution times on the real multiprocessor platform (as mentioned in Chapter 3), they can be used to derive a meaningful estimation of the *relative* execution times on the multiprocessor platform. It is important to note that these estimations are only used to test the mapping stage and will be replaced later by results of a more accurate simulation. Therefore we are only interested in statements like *"process A on resource type 1 is roughly n times faster than process B on resource type 2"*, based on which the mapping can be evaluated. Even if no estimation is possible or it becomes obvious that the estimated execution times are not significant, a mapping optimization can be performed. In this case, all the execution times can be set to 1. As a consequence of the objective values, the optimization will then try to distribute the processes evenly on all computation resources, so that in the optimal case, the number of processes mapped to each of the computation

```
% ASSUMPTION: total_s_inst is a vector containing the time spent
%  in each process instance of a specific process definition.
%  (calculated beforehand)

% specify the weighting matrix
speedup = [  % RISC DSP
              1,   1;      % dispatch_gops
              1,   0.8;    % dispatch_mb
              1,   1;      % dispatch_blocks
              1,   0.5;    % transform_block
              1,   1;      % collect_blocks
              1,   0.8;    % collect_mb
              1,   1;      % collect_gops
          ];

% calculate the estimation of the execution time of an instance
% of each process for all resource types.
E = [total_s_inst total_s_inst].*speedup;

% normalize the estimation with respect to the fastest process
% in order to eliminate the dependence on the simulation
% workstation
E_norm = E./(min(min(E)));

% RESULT: E_norm contains the estimated relative execution times.
```

Listing 4.1: Execution times estimation in Matlab for the MPEG-2 decoder case

resources will be the same. This will become clear in Section 4.4, where the objective values are described in detail.

Having the execution times of all the process instances at hand (obtained according to Section 2.4.5), they are multiplied by a matrix which describes the speed–up (or slow–down) factor of each (process, resource type) – pair. The resulting matrix is then normalized to the fastest execution time in order to eliminate the dependence on the workstation performance. Listing 4.1 shows a small Matlab program which performs this calculation. The idea behind this method is to use the SystemC simulation to obtain the relative execution times of the processes independent of the resource type, and to introduce this dependency via the speed–up matrix. The estimation on how much faster a specific process is executed on one resource with respect to another has to be done "by hand", since an instruction level simulation would be necessary to obtain this speed–up.

These execution times can then be given to the mapping stage in the form of a text file. The structure of the text file is kept very simple. It begins with a single "−1" on a separate line (the *delimiter*), followed by a list of names of the resource types, followed by a second delimiter, followed by a list of process names (inside single quotation marks) and the execution times (separated by a space). A final delimiter marks the end of the list. The execution times must be listed in the same order as the resource type names in the first part of the file. Each line beginning with a double–slash is ignored. An example is given in Figure 4.4.

```
// Execution times specification
-1
RISC
DSP
-1
'dispatch_gops' 1.0000 1.0000
'collect_gops' 157.3574 157.3574
'dispatch_mb' 30.0658 24.0526
'collect_mb' 60.5197 48.4158
'dispatch_blocks' 36.7441 36.7441
'collect_blocks' 46.8307 46.8307
'transform_block' 32.5413 16.2706
-1
```

Figure 4.4: Example file that specifies the execution times.

4.2.3 The List of Communication Paths

Besides the mapping of processes onto computation resources, also the mapping of channels onto communication resources is required. An architecture consists of processors ("computation resources"), memory cells and buses. But what exactly is a communication resource? Two observations help to answer this question.

1. It is not guaranteed, that every two computation resources are connected with a single bus, it may be necessary to pass several buses. As a consequence, the channels must be mapped to paths, rather than to single buses.
2. Each channel (and therefore also each communication resource) needs a certain amount of memory for implementing the FIFO buffer. The memory can either be a separate memory cell or the internal memory of a computation resource.

Therefore three different types of channel implementations can be distinguished, leading to three different types of paths onto which a channel can be mapped:

I) *Internal.* The buffer and the control logic are implemented on a single computation resource and no bus is needed. This is possible (and desirable) whenever two communicating processes are mapped to the same computation resource. The corresponding path consists only of the corresponding computation resource.

II) *Semi–internal.* The buffer and the control logic are again implemented on a single computation resource, but either the read or written data must be transferred over a bus. The corresponding path consists of two different computation resources (one at the begin, and the other one at the end of the path), and there are only bus resources in–between.

III) *External.* The control logic is implemented on a computation resource, but an external memory is used for the buffer, so the read *and* written data is transferred over one or more buses. The corresponding path consists of two different computation resources (one at the begin, and the other one at the end of the path), and there is exactly one memory cell, and an arbitrary number of buses in–between.

In the following, an algorithm is presented which finds all possible paths belonging to one of the aforementioned types.

List Paths Algorithm

Given is a graph $G = (V, E)$, where E is the set of edges and V is the set of vertices. Each vertex $v \in V$ stands for a resource and is a member of exactly one of three subsets: V_R is the subset of computation resources, V_B the subset of bus resources, and V_M the subset of memory resources. $v \in V_M$ therefore means that the vertex v is a memory resource, for instance. A *sub–path* P_S is an ordered set of vertices which are pairwise connected with an edge $e \in E$, with v_1 standing for the first and v_n for the last vertex, and for which the following conditions hold:

- $v_1 \in V_R,\, V_M$ (The first vertex of P_S is a computation or a memory resource)
- $v_n \in V_R,\, V_M$ (The last vertex of P_S is a computation or a memory resource)
- $(P_S \setminus \{v_1,\, v_n\}) \cap V_M = \{\}\,,\ |\{v_1,\, v_n\} \cap V_M| \leq 1$ (P_S contains no memory resource except for the first or the last vertex)
- $v_i \neq v_j \,\forall\, v_i, v_j \in V_B$ (P_S contains no bus resource twice)

A sub–path is therefore a direct, but not necessarily the shortest, path between two computation resources or one computation and one memory resource. What remains is to implement a method to find all subpaths between two computation or memory resources, but this is simplified in the case of acyclic graphs in that only one path for each resource pair must be found. This path can easily be found by a recursive search for the end node, beginning with the start node. As soon as the end node is found, the sub–path is complete.

Two sub–paths P_1 and P_2 can be *combined* to build a *path* P, if the last vertex of P_1 and the first vertex of P_2 are the same memory resource. The combined path is obtained by concatenating the second path without its first vertex to the last vertex of the first path. If the first *and* the last vertex of a sub–path are computation resources, the sub–path cannot be combined, but instead itself constitutes a path. A path is consequently an ordered set of vertices which are pairwise connected with an edge $e \in E$ for which the following conditions hold:

- $v_1 \in V_R$ (The first vertex of P is a computation resource)
- $|P \cap V_M| \leq 1$ (P contains at most one memory resource)
- $v_n \in V_R$ (The last vertex of P is a computation resource)

Additionally, one bus resource is only allowed to appear twice in a path when a memory resource lies in–between. A global list $\mathcal{G}$ of paths is kept, to which paths can be added during execution of the algorithm.

Algorithm 1 LIST THE PATHS OF GRAPH G

```
1: for all vertices $v_i \in V_R$ do
2:   $v_{\text{start}} \leftarrow v_i$, $v_{\text{end}} \leftarrow v_i$
3:   add path $\{v_{\text{start}}, v_{\text{end}}\}$ to the global list $\mathcal{G}$
4:   for all vertices $v_j \in V_R \setminus \{v_i\}$ do
5:     $v_{\text{end}} \leftarrow v_j$
6:     add all sub–paths with $v_1 = v_{\text{start}}$ and $v_n = v_{\text{end}}$ to the global list $\mathcal{G}$
7:     for all vertices $v_M \in V_M$ do
8:       add all sub–paths with $v_1 = v_{\text{start}}$ and $v_n = v_M$ to a temporary list $\mathcal{T}1$
9:       add all sub–paths with $v_1 = v_M$ and $v_n = v_{\text{end}}$ to a temporary list $\mathcal{T}2$
10:      add all combinations of the sub–paths in $\mathcal{T}1$ with the sub–paths in $\mathcal{T}2$ to the
         global list $\mathcal{G}$
11:    end for
12:  end for
13: end for
```

In the case of an acyclic graph, the algorithm is simplified by the fact that there is always only one sub–path between each pair of vertices[1]. In that case, the number of paths in a graph can be calculated from the algorithm above as

$$\begin{aligned} |\mathcal{G}| &= n_R + n_R\,(n_R - 1) + n_M n_R\,(n_R - 1) \\ &= {n_R}^2 + n_M n_R\,(n_R - 1) \end{aligned} \tag{4.1}$$

[1] This is true only because the architecture graph is non–directed. If directed graphs are used for modeling the architecture, there may be more than one sub–paths even for acyclic graphs.

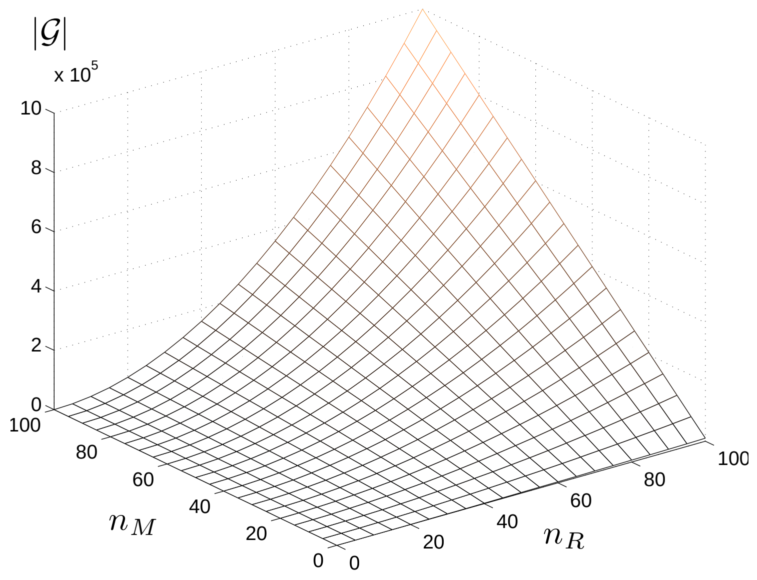


Figure 4.5: Increase of the number of paths $|\mathcal{G}|$ depending on the number of computation resources n_R and the number of memory resources n_M.

where n_R is the number of computation resources and n_M is the number of memory resources. Figure 4.5 shows the number of paths for an acyclic graph depending on the number of computation and memory resources.

4.2.4 The System Model

Figure 4.6 shows a class diagram of the `sysmodel` package. As can be seen, the system model consists of an *architecture model* and an *application model*. The architecture model itself consists of an *architecture graph* and a *path list*, whereas the application model consists of an *application graph* and an *execution times list*.

There are five different types of nodes and each node type has its specific parameters. The parameters of the resource, memory and bus node are given by the XML architecture specification and those of the process and channel node by the profiling results, and are described in detail in Section 4.4. Note that each node has an *id* and a *type id*, the former is unique amongst all nodes of a specific graph, whereas the latter is unique amongst all nodes of a specific graph which are of a specific type.

Example 4 *Consider an application graph with 3 processes and 2 channels. The* id*s of the nodes are numbered from* $0 \ldots 4$*, while the* type id*s of the process nodes are numbered from* $0 \ldots 2$ *and the* type id*s of the channel nodes from* $0 \ldots 1$*.*

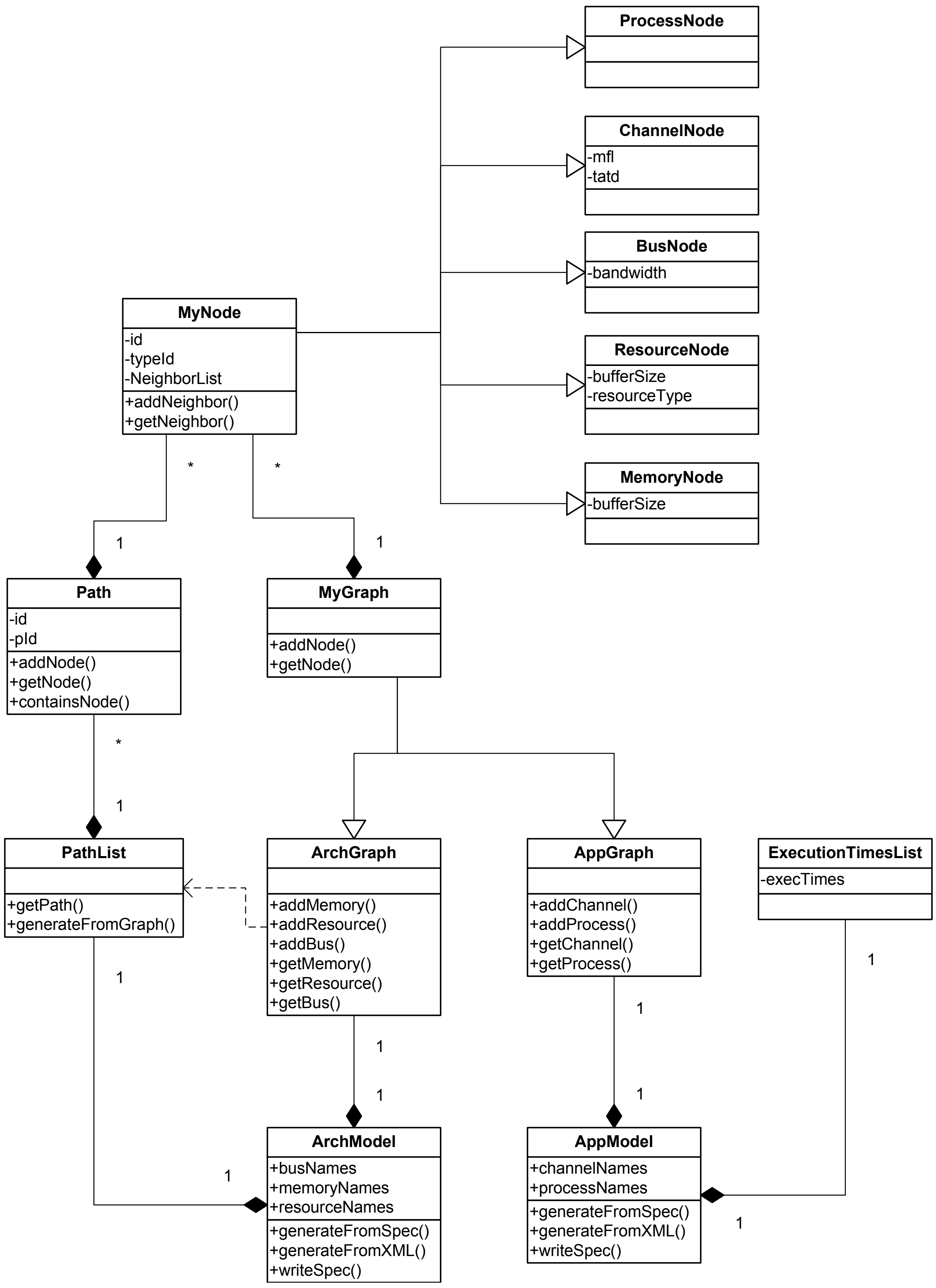


Figure 4.6: Class Diagram of the `sysmodel` package.

The application consists of processes and channels, therefore the application model keeps a list of process names and a list of channel names. These lists build a relation between the type id of the process nodes and channels nodes and their names. The type id is used inside the model only — the input as well as the output of the mapping stage only contain the names. The application model provides methods for building the application graph from the XML specification and from the textual EXPO specification file, as well as for writing the EXPO specification file. The usage of these functions is the following: `expogen` builds the graph from the XML specification and writes the EXPO specification file. When EXPO is run, the `ShSpecification` class then builds the graph from the EXPO specification file (see Figure 4.2 again for an illustration).

The application graph is the main data structure of the application model. It is derived from the generic graph, but only process and channel nodes can be added. The application graph additionally provides some helper functions, like for instance `getChannelInProcess(c)`, which returns the process which is connected to the input of channel `c`. For the sake of clarity, these functions are not shown in the class diagram.

The architecture, on the other hand, consists of computation resources of different types, buses and memory cells. As in the case of the application model, the architecture model keeps a list of their names in order to relate them with the type ids of the nodes. Additionally, again exactly as in the case of the application model, methods are provided to build the graph from the XML specification and from the textual EXPO specification file, as well as for writing the EXPO specification file. Note that a simplified XML scheme is used for the architecture specification, where only few elements are allowed.

The architecture graph is the main data structure of the architecture model. It is also derived from the generic graph, but only resource, memory, and bus nodes can be added.

The architecture model additionally contains the PathList object, which provides a method called `generateFromGraph()`. A call to `generateFromGraph()` computes the list of all paths using the List Paths algorithm presented in Section 4.2.3.

4.2.5 Generating the Specification File with `expogen`

With the small application called `expogen`, the EXPO specification file can be generated. `expogen` takes the application specification, the architecture specification, the execution times and the DOL profiler output as its arguments, and performs the following steps:

- The execution times specification file is read and an instance of the ExecutionTimesList object is initialized.

- The profile result is parsed using the `DOLProfiler` package.
- The application graph is generated from the XML specification.
- The architecture graph is generated from the XML specification.
- The EXPO specification file is written by using the `writeSpec()` functions of the application and the architecture model objects.

The generated EXPO specification file can be used to run EXPO. Minor changes in the specification, as for instance, adjusting bandwidths or buffer sizes, can be made directly on the specification file, without running `expogen`. If the structure of the application or the architecture is affected, the specification file should be re–generated with `expogen` in order to avoid running EXPO with an inconsistent specification file.

4.2.6 Implementation of the `ShSpecification` Class

The implementation of the `ShSpecification` class is very simple, since most of the functionality is implemented in the `sysmodel` package. The most important function of the class is the `simpleFileInput()`, which is defined in the `Specification` interface. `simpleFileInput()` is called by EXPO once during initialization, and reads the whole specification from the text file. This is done by generating a `StreamTokenizer` object from the specification file and calling the `generateFromSpec()` functions of the application and architecture models. After generating these graphs, the list of paths and the $S_{i,j}$–matrix are built, again simply by calling the corresponding methods of the `sysmodel` package. All the remaining functions just ease the use of the system model by providing a direct access to the most frequently used values.

4.3 Gene Representation

The second issue is the representation of a gene, i.e. of a single mapping solution. Additionally, genes must be generated randomly, mutated, and recombined. These operators are described in this section.

4.3.1 Initial Situation

Given is a set of processes P and a set of channels C, which build a process network, implying that each channel $c_k \in C$ is connected to exactly one process at its input, and one at its output[2]. These connections are assumed to be given as two functions called *in(k)* and *out(k)*. $p_{\mathrm{in}(k)} \in P$ is therefore the process connected to the input of channel c_k, and $p_{\mathrm{out}(k)} \in P$ the process connected to the output of channel c_k, accordingly. Furthermore, a set of computation resources (processors) R and a set of communication paths S are given. Each communication path $s_k \in S$ has a computation resource $r_i \in R$ as its start point, a computation resource $r_j \in R$ as its end point, and several bus resources (and possibly a memory resource) in-between. It is possible that several paths have the same start and end resources, the start and end resources are however not interchangeable, as it is always assumed that the FIFO control logic (and possibly also the buffer memory) is incorporated in the start resource. Let $S_{i,j}$ be the set of communication paths with start resource r_i and end resource r_j. The cardinality of S can then be stated as

$$|S| = \sum_{i=0}^{|R-1|} \sum_{j=0}^{|R-1|} s_{i,j} \tag{4.2}$$

where $s_{i,j} \equiv |S_{i,j}|$ is the number of communication paths from start resource r_i to end resource r_j and follows

$$s_{i,j} = \begin{cases} 1 & \text{if } i = j \\ \geq 1 & \text{else} \end{cases} \tag{4.3}$$

because there is always only one communication path if the start and end resources are the same, namely the resource itself (leads to an *internal* FIFO implementation), and each pair of resources is connected through at least one communication path.

Example 5 *The specification shown in Figure 4.7 leads to the following initial situation:*

- $P = \{P_0,\ P_1,\ P_2\},\ |P| = 3.$
- $C = \{C_0,\ C_1\},\ |C| = 2.$
- $P_{\mathrm{in}(0)} = P_0,\ P_{\mathrm{in}(1)} = P_1,\ P_{\mathrm{out}(0)} = P_1,\ P_{\mathrm{out}(1)} = P_2$
- $R = \{R_0,\ R_1\},\ |R| = 2$

In order to get the set of communication paths, all possible paths must be listed using Algorithm 1 of Section 4.2.3, as done in Table 4.2.

Later on, the number of communication paths $s_{i,j}$ for all i, j are needed. Besides that, each communication path must be annotated with a number that uniquely identifies it

[2]The terms “input” and “output” here always refer to the *channel* input and *channel* output. Each channel input is connected to a process output and vice versa.

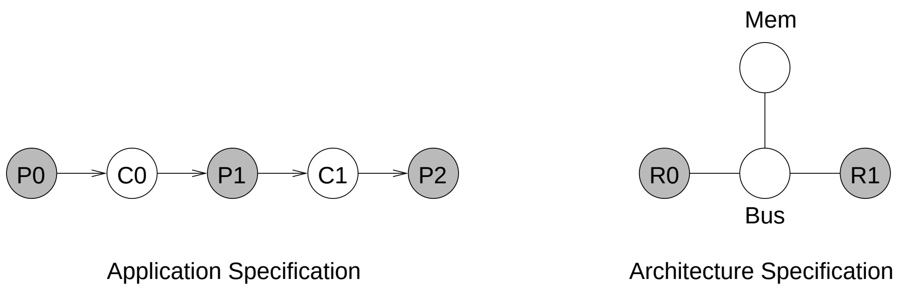


Figure 4.7: Example specification.

No.	Start/End	Path	Type	ID
0	R0, R0	R0	internal	0
1	R0, R1	R0, Bus, R1	semi–internal	0
2		R0, Bus, Mem, Bus, R1	external	1
3	R1, R0	R1, Bus, R0	semi–internal	0
4		R1, Bus, Mem, Bus, R0	external	1
5	R1, R1	R1	internal	0

Table 4.2: List of all possible communication paths. The ID field is calculated during the preparation (Example 6).

among all paths with the same start and end resources. The identifiers must be consecutive numbers starting with 0. This preparation can be done easily by linearly traversing the list of paths, incrementing a separate counter for each start and end resource pair $\{R_i, R_j\}$, and annotate each path with the current content of the counter variable. The final values of all counters can be used to build the $s_{i,j}$–matrix, which is then used later on.

Example 6 *Preparing the specification of Example 5 leads to the ID annotation of Table 4.2, and the following $s_{i,j}$–matrix:*

$$s_{i,j} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

4.3.2 Representation

Each binding is represented by two vectors, $V_P = (p_1\ p_2\ \dots\ p_p)$, which describes the binding of processes to computation resources, and $V_C = (c_1\ c_2\ \dots\ c_c)$, which describes the binding of channels to communication paths. Each element p_k of V_P is an integer number between 0 and $(r-1)$ describing to which of the r computation resources the process P_k is bound. $p_{\mathrm{in}(k)}$ and $p_{\mathrm{out}(k)}$ are therefore the resources, onto which the processes at the input and the output of channel k is bound, respectively. Each element c_k of V_C is an integer number between 0 and $\left(s_{p_{\mathrm{in}(k)},p_{\mathrm{out}(k)}} - 1\right)$, with $s_{p_{\mathrm{in}(k)},p_{\mathrm{out}(k)}}$ being the number of communication paths between the two resources onto which the two processes attached to channel C_k are bound.

Example 7 *Continuing our previous examples, the binding shown in Figure 4.8 has the following representation:*

$$V_P = (0\ 0\ 1) \qquad V_C = (0\ 1) \tag{4.4}$$

4.3.3 Random Generation

Algorithm 2 describes the random generation of an individual. It is important that the random numbers generated are uniformly distributed over the given interval.

4.3.4 Mutation

The most obvious mutation operation is to change one element of one of the vectors. Unfortunately, changing V_P may result in an infeasible solution, namely if one or more elements of the V_C vector represents a path that does not exist. To resolve this problem,

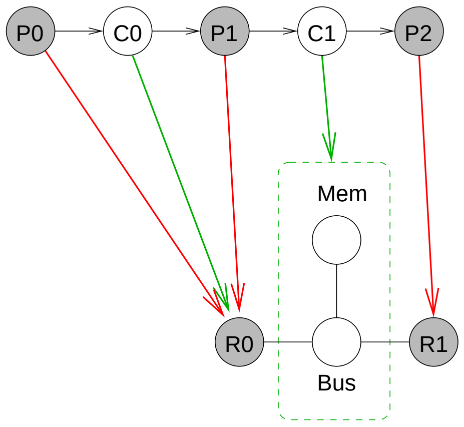


Figure 4.8: Example binding

Algorithm 2 RANDOM GENERATION OF AN INDIVIDUAL

```
1: for all processes $P_k$ do
2:     generate random binding $p_k \in [0, (r-1)]$
3: end for
4: for all channels $C_k$ do
5:     determine $s_{p_{\mathrm{in}(k)}, p_{\mathrm{out}(k)}}$ from the in and out functions, the vector $V_P$ (generated
       before) and the $s_{i,j}$–matrix
6:     generate random binding $c_k \in \left[0, \left(s_{p_{\mathrm{in}(k)}, p_{\mathrm{out}(k)}} - 1\right)\right]$
7: end for
```

the V_C vector can either be re-generated or repaired after a change of V_P. The easiest repair mechanism is to replace all incorrect elements with a random value which lies inside the correct interval. This is preferable over a re-generation as the mutation should perform only a slight variation and not a completely different individual. A change of a single entry of the V_P vector is re-binding of a single process, which means that only the entries of the V_C vector which represent channels that are connected to this process must be repaired. Algorithm 3 describes the mutation process.

Algorithm 3 RANDOM MUTATION OF AN INDIVIDUAL

```
1: if computation binding is to be changed then
2:    randomly choose a process $P_k$
3:    repeat
4:       generate new random binding $p_k \in [0, (r-1)]$
5:    until new binding is different to the old one
6:    for all channels $C_k$ which are connected to $P_k$ do
7:       determine $s_{p_{\text{in}(k)}, p_{\text{out}(k)}}$ from the in and out functions, the vector $V_P$ and the $s_{i,j}$–matrix
8:       generate random binding $c_k \in \left[0, \left(s_{p_{\text{in}(k)}, p_{\text{out}(k)}} - 1\right)\right]$
9:    end for
10: else {communication binding only is to be changed}
11:    randomly choose a channel $C_k$
12:    determine $s_{p_{\text{in}(k)}, p_{\text{out}(k)}}$ from the in and out functions, the vector $V_P$ and the $s_{i,j}$–matrix
13:    repeat
14:       generate new random binding $c_k \in \left[0, \left(s_{p_{\text{in}(k)}, p_{\text{out}(k)}} - 1\right)\right]$
15:    until new binding is different to the old one
16: end if
```

Example 8 *If the binding of Figure 4.8 is to be mutated, the process P_2 can be bound to the computation resource R_0 instead of R_1. The vector V_P then becomes*

$$V_P = (0\ 0\ 0)\ . \tag{4.5}$$

as a consequence, the vector $V_C = (0\ 1)$ is not valid any more, since the channel C_1 now connects resource R_0 with R_0 itself, but there is no path with ID 1 with the start–end pair $\{R_0, R_0\}$ (cf. Table 4.2). The binding of C_1 therefore must be repaired, and the resulting representation is

$$V_P = (0\ 0\ 0) \qquad V_C = (0\ 0) \tag{4.6}$$

4.3.5 Recombination

As the vector V_C strongly depends on V_P, it makes no sense to combine the communication binding (V_C) of two individuals without considering the process binding (V_P). Instead, if two individuals A and B are to be combined, it is desirable to take half of the process bindings from A, and the other half from B. After the process binding is fixed, the communication binding can be derived. Since we want to keep as much information as possible from the parents, all channels that connect two processes which are mapped to the same resources as in one of the parents keep the binding of the corresponding parent. All the other channels must be re-bound randomly. Algorithm 4 describes the recombination process.

Algorithm 4 RECOMBINATION OF TWO PARENTS A AND B TO OBTAIN OFFSPRING C

```
1: generate $V_{P,C}$ by concatenating the first $\lfloor \frac{p}{2} \rfloor$ elements of $V_{P,A}$ and the last $\lceil \frac{p}{2} \rceil$ elements
   of $V_{P,B}$
2: for all channels $C_k$ do
3:   if $V_{P,C}\,(\text{in(k)}) = V_{P,A}\,(\text{in(k)})$ and $V_{P,C}\,(\text{out(k)}) = V_{P,A}\,(\text{out(k)})$ then
4:     take binding of $C_k$ from parent $A$, i.e. $c_{k,C} = c_{k,A}$
5:   else if $V_{P,C}\,(\text{in(k)}) = V_{P,B}\,(\text{in(k)})$ and $V_{P,C}\,(\text{out(k)}) = V_{P,B}\,(\text{out(k)})$ then
6:     take binding of $C_k$ from parent $B$, i.e. $c_{k,C} = c_{k,B}$
7:   else {binding of $C_k$ cannot be taken from one of the parents}
8:     determine $s_{p_{\text{in}(k)},p_{\text{out}(k)}}$ from the in and out functions, the vector $V_{P,C}$ (generated
       before) and the $s_{i,j}$–matrix
9:     generate random binding $c_{k,C} \in \left[0, \left(s_{p_{\text{in}(k)},p_{\text{out}(k)}} - 1\right)\right]$
10:  end if
11: end for
```

The algorithm can easily be extended to generate 2 offsprings C and D from each pair of parents A and B by additionally generating $V_{P,D}$ by concatenating the last $\lceil \frac{p}{2} \rceil$ elements of $V_{P,A}$ and the first $\lfloor \frac{p}{2} \rfloor$ elements of $V_{P,B}$ and generating the channel binding as explained before.

Example 9 *If the two parents A and B shown in Figure 4.9 are chosen to be recombined to build the offspring C, the two vectors*

$$V_{P,A} = (0\ 0\ 0) \qquad V_{P,B} = (1\ 1\ 1) \tag{4.7}$$

are concatenated (according to algorithm 4) and result in the offspring binding vector

$$V_{P,C} = (0\ 1\ 1)\ . \tag{4.8}$$

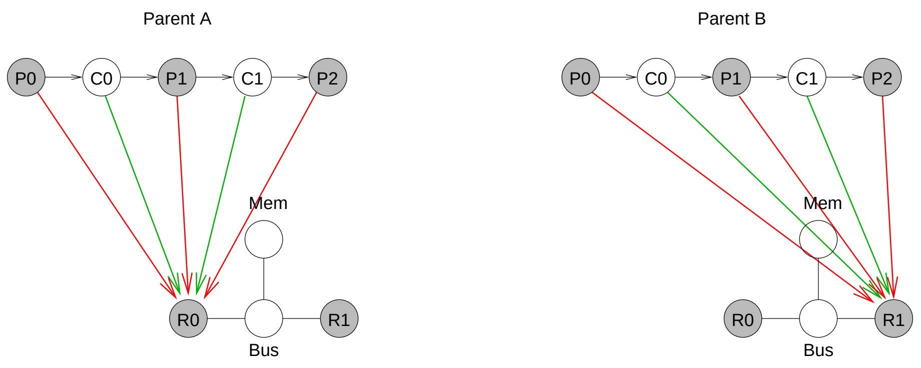


Figure 4.9: Example binding.

Now, each channel has to be considered separately in order to build the vector $V_{C,C}$ from the vectors $V_{C,A}$ and $V_{C,B}$. The first channel (C_0) connects P_0 and P_1. P_0 is bound to R_0 and P_1 to R_1, which is not the case in either parent. This means that the binding of C_0 can not be taken from one of the parents, it must be rebuilt randomly. The second channel (C_1) connects P_1 and P_2, and the binding of both of these processes is equal to the binding of parent B, which means that also the binding of the connecting channel (C_1) can be taken from parent B. The resulting offspring is

$$V_{P,C} = (0\ 1\ 1) \qquad V_{C,C} = (\{0,1\}\ \ 0)\ , \tag{4.9}$$

where the binding of channel C_0 is generated randomly.

4.3.6 Implementation of the `ShGene` Class

The `Gene` interface is the second EXPO interface which must be implemented by a problem–specific class. This class contains the data structures which represent a single gene, and additionally provide methods for generating a random gene, mutating a gene and recombine two genes to obtain a child gene.

The `ShGene` class contains two vectors, `Vp[numOfProcesses]` represents the binding of processes onto computation resources, and `Vc[numOfChannels]` represents the binding of channels onto communication paths. These vectors constitute the vectors V_P and V_C, which were introduced in Section 4.3.2. Additionally, the `ShGene` class contains the functions `generateRandomGene()`, `mutateGene()` and `crossOverGene(parentGene)`, which implement Algorithms 2, 3 and 4 of Sections 4.3.3, 4.3.4 and 4.3.5, respectively.

4.4 Performance Data and Objective Values

As mentioned before, the PISA interface separates the problem–specific part of an optimizer from the problem–independent part. The third and last problem–specific issue is the determination of the *objective values*, i.e. determination of a quality measure for each optimization goal. Selection of genes based on these objective values is then a problem–independent task, for which many different algorithms are available (see the *Selectors* section on the PISA web–page [23]).

4.4.1 Processor Load

The first goal, "minimizing the maximum load of processors" is evaluated by using the following equation for the maximum load of the computation resources $l_{r,\max}$

$$l_{r,\max} = \max_{R_j} \left\{ \sum_{\{P_i : p_i = j\}} t_{i,j} \right\} \tag{4.10}$$

where the maximum is taken over all computation resources, the sum is taken over all processes bound to the corresponding computation resource, and $t_{i,j}$ is the estimated execution time of process P_i on resource R_j. The reason for minimizing the maximum sum of execution times, is the following: if we neglect the blocking read accesses, the total program execution time is given by the maximum execution time over all resources, which is exactly what is minimized.

4.4.2 Communication Load

The measure for the second optimization goal, "minimizing the maximum load of communication links" is

$$l^*_{c,\max} = l_{c,\max} + p_{\text{buf}} \ , \tag{4.11}$$

where $l^*_{c,\max}$ is the sum of the maximum load of communication links $l_{c,\max}$ and a penalty value p_{buf} for an individual with one or more FIFO buffer that is likely to overflow. Let $\mathcal{B} = \{B_1 \dots B_b\}$ be the set of communication resources (buses[3]). Each bus is characterized by a bandwidth *bw(B_i)*. The cumulative amount of transferred data of a bus B_i is denoted with *cumul_data(B_i)*, and stands for the sum of the total transferred data of all channels

[3]A communication resource can also be a dedicated link. For the sake of clarity, the term *bus* includes dedicated links here.

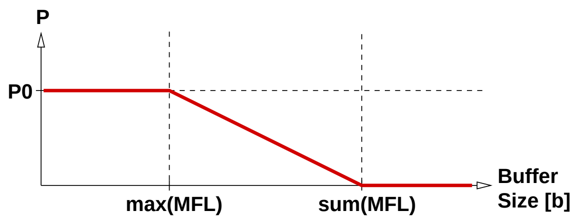


Figure 4.10: Penalty value for buffer overflow depending on the buffer size.

that are bound to a communication path that passes the bus B_i. The maximum load of communication links $l_{c,\max}$ is then

$$l_{c,\max} = \max_{B_i} \left\{ \frac{\text{cumul\_data}(B_i)}{\text{bw}(B_i)} \right\} \tag{4.12}$$

The penalty value p_{buf} is defined according to Figure 4.10. Since more than one channel may be mapped to a single memory resource, buffer overflows are not always predictable. Assume, for instance, that three channels are mapped to a path containing a specific memory resource. Each channel has its maximum fill level, but it is not clear which channel reaches its maximum fill level when. Therefore three cases are distinguished:

- If the buffer size available on the memory resource is larger than the sum of all maximum fill levels, no overflow will occur. The sum of all maximum fill levels stands for the worst case scenario, meaning that all channels reach their maximum fill levels at exactly the same time. In this case, the penalty value is 0.
- If the buffer size available on the memory resource is smaller than the maximum of all maximum fill levels, there will be at least one overflow for sure. The penalty value is then P_0, a constant value specified in the EXPO properties file.
- If the buffer size lies between the aforementioned values, the overflow is more likely, the smaller the buffer size. In this case the penalty value is linearly increasing with an decreasing buffer size.

4.4.3 Implementation of the `ShAnalyzer` Class

The `ShAnalyzer` class contains two important functions. `analyze()` returns an array of `double` values containing the calculated objective values for a given gene. `analyze()` implements the formulas derived in Sections 4.4.1 and 4.4.2. The class additionally contains a function named `getReport()`, which returns a string containing a textual form of a mapping solution for a given gene. `getReport()` therefore converts the numerical representation (i.e. the vectors `Vp` and `Vc`) into a textual form, which is readable and understandable by a human being. This function can be adapted if the mapping is required to be in an XML format for further processing.

5 Evaluation and Results

In this chapter, the capabilities of the mapping stage prototype is demonstrated using the MPEG-2 video decoder application and the example architecture proposed in Section 4.2.1. In the first section, the MPEG-2 video decoder application is configured. The next section then uses the trace file generator and the `dolprofiler` package to obtain the application model's parameters. In the last section, these parameters are used to perform the mapping optimization, using the example architecture and estimated execution times.

5.1 MPEG-2 Decoder Application

We start with the simplest possible configuration, which is $N_1 = N_2 = N_3 = 1$. Since the example architecture contains only 4 computation resources (2 RISCs and 2 DSPs), it is not meaningful to specify an application with hundreds of processes. Figure 5.1 shows this simple process network. If we have a look at the process execution times (profiled according to Section 2.4.5 and shown in Figure 5.2), it becomes obvious that the pipeline is badly balanced. The *dispatch_blocks* process needs roughly twice the time of the *dispatch_mb* process, and the *transform_block* process needs twice the time of the *dispatch_blocks* process. In order to balance the pipeline, a total of 2 *dispatch_blocks* and 4 *transform_block* process instances are necessary. The improved configuration is therefore

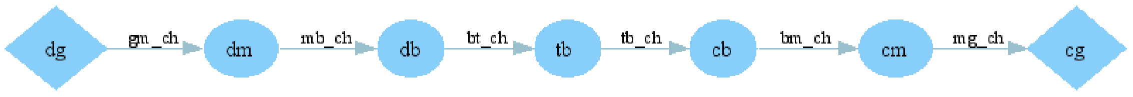


Figure 5.1: Simplest possible process network ($N_1 = N_2 = N_3 = 1$).

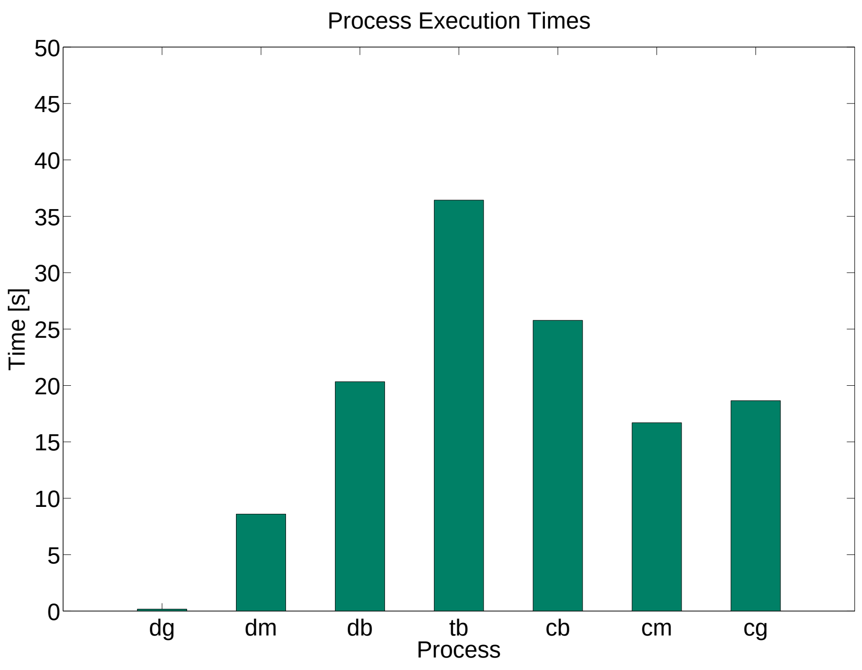


Figure 5.2: Process execution times for $N_1 = N_2 = N_3 = 1$.

$N_1 = 1$, $N_2 = N_3 = 2$, which leads to the process network of Figure 5.3 and the process execution times shown in Figure 5.4.

5.2 Profiling Results

The application can now be simulated with the SystemC application generated by the DOL. As described in Chapter 3, the simulation generates a trace file, based on which the parameters MFL (*maximum fill level*) and TATD (*total amount of transferred data*) can be calculated. The resulting parameters for the MPEG-2 video decoder application are shown in Figure 5.5.

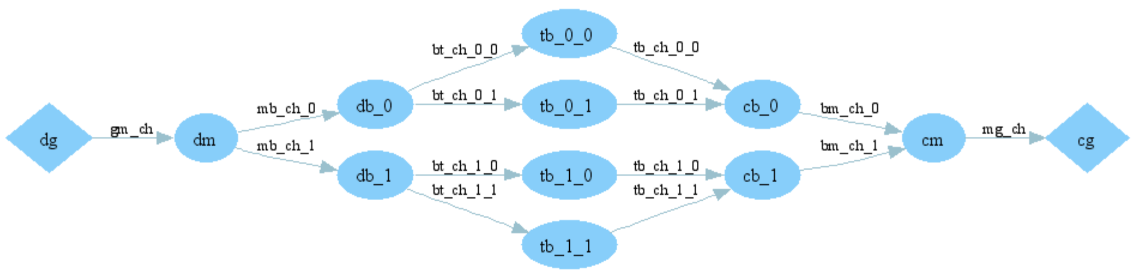


Figure 5.3: Improved process network ($N_1 = 1$, $N_2 = N_3 = 2$).

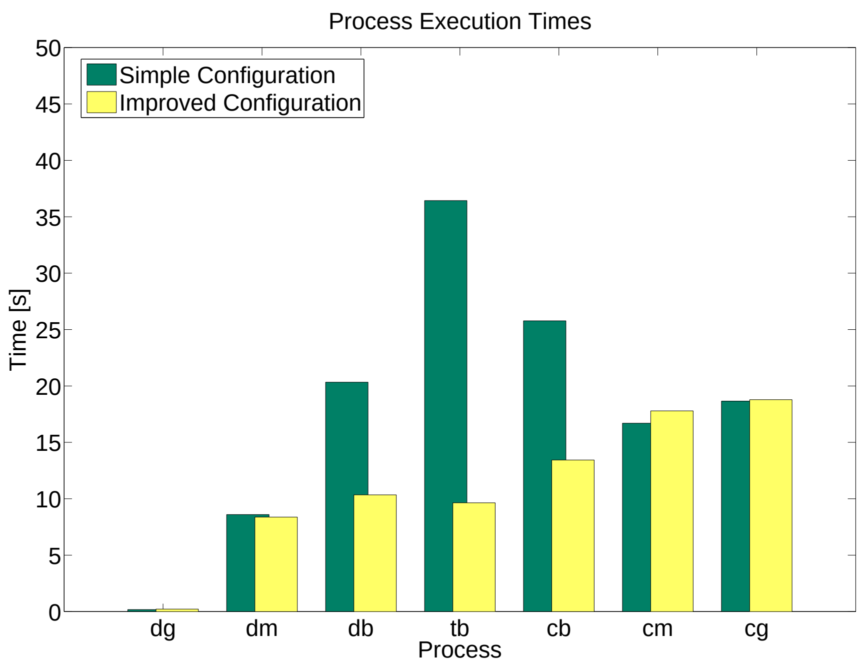


Figure 5.4: Process execution times of the improved process network.

```
----- Channel Analyzer Report ----
<channel>      <MFL>        <TATD>
<gm_ch>      127'243       144'951
<mb_ch_0>    308'625     8'598'176
<mb_ch_1>    308'625     8'604'319
<bt_ch_0_0>  307'384     7'221'242
<bt_ch_0_1>  307'383     7'221'241
<bt_ch_1_0>  307'383     7'226'731
<bt_ch_1_1>  307'383     7'226'731
<tb_ch_0_0>      994     7'221'241
<tb_ch_0_1>    1'491     7'221'241
<tb_ch_1_0>      994     7'226'731
<tb_ch_1_1>      994     7'226'730
<bm_ch_0>     20'091    16'023'199
<bm_ch_1>     19'854    16'035'487
<mg_ch>      126'957     3'808'712
```

Figure 5.5: Resulting parameters for the improved configuration.

5.2.1 TATD

The total amount of transferred data of the `gm_ch` channel is more or less equal to the size of the file containing the video sequence, since the *dispatch_gop* process does not perform any decoding. The amount of data transferred to the *dispatch_blocks* processes (via the `mb_ch` channels) is roughly $2 \cdot 60 = 120$ times larger, mainly due to the variable length decoding. At the block level (`bt_ch` and `tb_ch` channels), the amount of data is increased by a factor of about 1.6 (from $2 \cdot 8.2$ MB to $4 \cdot 7.2$ MB), mainly because the header information is now transmitted together with each block, instead of just once for all blocks of a macroblock. The inverse discrete cosine transform does not change the amount of transferred data, because each 8×8 block of coefficients is transformed into an 8×8 block of chrominance/luminance values. On the `bm_ch` channels there is no significant increase in the amount of data (from $4 \cdot 7.2$ MB to $2 \cdot 16.0$ MB). Finally, after the motion compensation step is performed, most of the remaining header information can be discarded and only the uncompressed raw image data is transferred over the `mg_ch` channel.

5.2.2 MFL

Concerning the maximum fill level, two points are especially noticeable. First, the maximum fill level of the `gm_ch` channels is almost as large as the total amount of transferred data, which reflects the fact that the *dispatch_gop* process is very fast compared to the *dispatch_mb* process. Second, the maximum fill level of the channels after the *transform_block* process are much smaller. The reason for this can be found by having a look at the simple configuration $N_1 = N_2 = N_3 = 1$. The process execution times show, that the *collect_blocks* process is faster than the *transform_block* process, which means that the data does not accumulate on the FIFO channel. In the case of the improved configuration, the execution time of each instance of the *transform_block* process is reduced, but since all the processes are simulated on a single workstation, this reduction does not affect the maximum fill level. This is identified as the main limitation of the parameters at the application level: the maximum fill level of the FIFO buffers depend on the number of process instances mapped to a single resource. The influence of the maximum fill level on the mapping optimization, which can be adjusted by selecting the penalty value P_0, should therefore be chosen small compared to the influence of the total amount of transferred data.

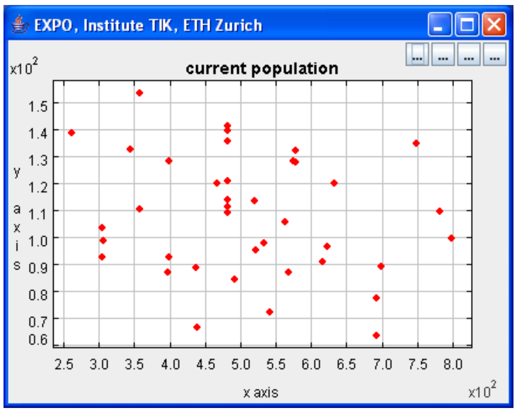
5.3 Mapping Optimization

In order to run the mapping optimization, the EXPO specification file must be generated. This is done with `expogen` according to Section 4.2.5. The execution times are estimated according to Section 4.2.2, with a DSP speedup factor of 10 for the inverse discrete cosine transform process (*transform_block*), and a speedup factor of 2 for the motion compensation process (*collect_mb*). The resulting execution times estimation is shown in Figure 5.6.

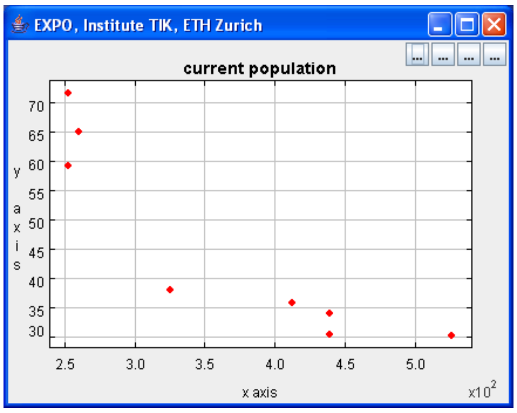
```
// Execution times specification for (122)
-1
RISC
DSP
-1
'dispatch_gops' 1.0000 1.0000
'collect_gops' 86.9733 86.9733
'dispatch_mb' 38.7908 38.7908
'collect_mb' 82.3654 41.1827
'dispatch_blocks' 95.8154 95.8154
'collect_blocks' 124.3919 124.3919
'transform_block' 178.5883 17.8588
-1
```

Figure 5.6: Execution times estimation (normalized to the fastest process).

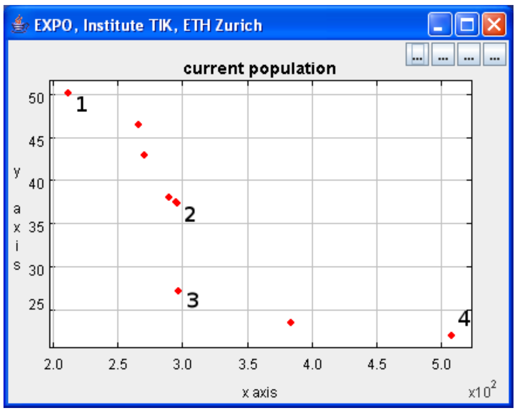
EXPO can then be run with the specification file generated by `expogen`. Figure 5.7 shows the initial population, an intermediate population at generation 20, and the final population, which did not change from generation 80 to 200. Four of the pareto–optimal solutions are labeled with the numbers 1 to 4 for further reference.



(a) Initial population



(b) Generation 20



(c) Generation 200

Figure 5.7: Population at three different generations. The x–axis shows the load of the computation resources, and the y–axis the load of the communication resources.

As can be seen in Figure 5.7c, Solution 1 has a very small maximum execution time, but is bad in terms of the communication. This solution is shown in Figure 5.8. Clearly, this execution time can only be reached by mapping all the *transform_block* processes to DSPs and additionally distributing all the other processes evenly on the other resources. The price to pay is a bad communication performance with many long paths.

Solution 2 trades some of the execution time against a better communication, but if we allow the execution time to be a little bit larger, the communication can be improved again significantly. Solution 3 would therefore be preferable over Solution 2.

Solution 4 finally sacrifices a lot in terms of execution times but reaches by far the best communication performance. This solution is shown in Figure 5.9. Since the *transform_block* processes must be mapped to DSPs, the communication is optimized, by not using the RISC processors at all. Additionally, two connected processes are mapped to the same DSP wherever possible, so that a total of only three external communication paths must be used.

5.4 Conclusion and Future Work

First, an MPEG-2 video decoder application for the SHAPES DOL was successfully implemented. The resulting process network contains no cycles and no *request and answer* structure, but provides a feed–forward data flow from the first to the last process. Additionally, the process network is reconfigurable, which means that the number of parallel processes can be adjusted without changing the implementation of the processes. This is necessary for maintaining scalability and making the best use of the available architecture. The process network does not make use of the read/write testing functions, which means that the application is realized as a Kahn process network (except for the bounded FIFO buffer sizes). It was shown how the process execution times can be profiled in order to choose an appropriate configuration and balance the pipeline. The following conclusions can be drawn:

- The SHAPES DOL model of computation is suited for complex applications, no extensions were found to be necessary.
- The programming effort is, thanks to the functional simulation using SystemC, comparable to the programming effort for standard C programs. For programming, compiling, debugging, testing and profiling the established practices and tools from sequential programming can be used.

```
// ***********************
// GENE INDEX 7914
// ***********************
Gene Vectors:
-------------
Vp = [1 0 1 2 1 2 3 0 3 2 3 3 ]
Vc = [0 1 0 0 0 0 3 0 2 0 0 0 0 3 ]

Implementation Details:
-----------------------
Mapping of Processes onto Computation Resources:
dg ---> risc_2
cg ---> risc_1
dm ---> risc_2
cm ---> dsp_1
db_0 ---> risc_2
cb_0 ---> dsp_1
db_1 ---> dsp_2
cb_1 ---> risc_1
tb_0_0 ---> dsp_2
tb_0_1 ---> dsp_1
tb_1_0 ---> dsp_2
tb_1_1 ---> dsp_2

Mapping of Channels onto Communication Paths:
gm_ch ---> risc_2
mg_ch ---> dsp_1 intra_tile_bus_1 mem_1 intra_tile_bus_1 risc_1
mb_ch_0 ---> risc_2
bm_ch_0 ---> dsp_1
mb_ch_1 ---> risc_2 intra_tile_bus_2 inter_tile_bus intra_tile_bus_3 dsp_2
bm_ch_1 ---> risc_1 intra_tile_bus_1 dsp_1
bt_ch_0_0 ---> risc_2 intra_tile_bus_2 inter_tile_bus intra_tile_bus_3 mem_3
                intra_tile_bus_3 dsp_2
tb_ch_0_0 ---> dsp_2 intra_tile_bus_3 inter_tile_bus intra_tile_bus_1 dsp_1
bt_ch_0_1 ---> risc_2 intra_tile_bus_2 mem_2 intra_tile_bus_2 inter_tile_bus
                intra_tile_bus_1 dsp_1
tb_ch_0_1 ---> dsp_1
bt_ch_1_0 ---> dsp_2
tb_ch_1_0 ---> dsp_2 intra_tile_bus_3 inter_tile_bus intra_tile_bus_1 risc_1
bt_ch_1_1 ---> dsp_2
tb_ch_1_1 ---> dsp_2 intra_tile_bus_3 mem_3 intra_tile_bus_3 inter_tile_bus
                intra_tile_bus_1 risc_1
```

Figure 5.8: Solution 1. The processes are distributed evenly on all available resources. The price we pay for this optimal execution time is a lot of external communication.

```
// ***********************
// GENE INDEX 7963
// ***********************
Gene Vectors:
-------------
Vp = [3 2 3 2 3 2 2 2 3 3 2 2 ]
Vc = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]


Implementation Details:
-----------------------
Mapping of Processes onto Computation Resources:
dg ---> dsp_2
cg ---> dsp_1
dm ---> dsp_2
cm ---> dsp_1
db_0 ---> dsp_2
cb_0 ---> dsp_1
db_1 ---> dsp_1
cb_1 ---> dsp_1
tb_0_0 ---> dsp_2
tb_0_1 ---> dsp_2
tb_1_0 ---> dsp_1
tb_1_1 ---> dsp_1


Mapping of Channels onto Communication Paths:
gm_ch ---> dsp_2
mg_ch ---> dsp_1
mb_ch_0 ---> dsp_2
bm_ch_0 ---> dsp_1
mb_ch_1 ---> dsp_2 intra_tile_bus_3 inter_tile_bus intra_tile_bus_1 dsp_1
bm_ch_1 ---> dsp_1
bt_ch_0_0 ---> dsp_2
tb_ch_0_0 ---> dsp_2 intra_tile_bus_3 inter_tile_bus intra_tile_bus_1 dsp_1
bt_ch_0_1 ---> dsp_2
tb_ch_0_1 ---> dsp_2 intra_tile_bus_3 inter_tile_bus intra_tile_bus_1 dsp_1
bt_ch_1_0 ---> dsp_1
tb_ch_1_0 ---> dsp_1
bt_ch_1_1 ---> dsp_1
tb_ch_1_1 ---> dsp_1
```

Figure 5.9: Solution 4. The external communication is now reduced to to the minimum, at the cost of execution time — two resources are not used at all.

- The main challenge is the parallelization of the algorithm, which requires a good application domain knowledge and allows for lots of different approaches.

Future work may include optimization of the application in terms of both execution time and memory requirements. This was not the main concern in this thesis.

Second, it was shown that the functional simulation can be used for obtaining mapping–relevant parameters at the application level. For this reason, the SystemC source code generator was extended by trace file generating routines. The trace file contains all read and write API calls and can be used to extract the parameters. A Java package `dolprofiler` was implemented, which extracts several parameters for each channel, from which the *total amount of transferred data* and the *maximum fill level* were identified as the most important ones. It was shown, however, that the maximum fill level depends on the architecture, which makes it less significant than the total amount of transferred data.

Third, a prototype of the mapping stage was implemented, using the PISA interface and EXPO. For this reason, an example architecture was chosen, a method of estimating the process execution times was proposed, and a Java application `expogen` was written, which generates a single text–based specification file from the architecture specification, the application specification, the execution times estimation, and the trace file (using the `dolprofiler` package). Additionally, a Java package `sysmodel` was implemented, containing the data structures of the whole system. An algorithm for finding all communication paths of the architecture was proposed and implemented in the system model. The EXPO application was extended by three problem–specific classes, one containing the system specification, a second one containing the objective value calculation routines, and a third one containing a single gene representation, including its random generation, mutation, and recombination routines. The following conclusions can be drawn:

- Application level parameters have been used to find pareto–optimal mappings.
- The main limitation is the absence of accurate execution time measurements, which are necessary in order to optimize the load of the computation resources.
- A second limitation is the fact the number of available resource (and also the scheduling policies) have an influence on the fill levels of the FIFO buffers, which is not considered in the functional simulation.
- Evolutionary Algorithms are suited well for solving the mapping optimization problem. Although for this thesis, the size of the search space hardly justifies the use

of evolutionary algorithms, the architecture is expected to be much more complex in the future, which will make a heuristic optimization approach inevitable.

- The use of PISA and EXPO for solving the multi–objective problem turned out to be very convenient and allowed for a concentration on the problem–specific tasks which significantly reduced the implementation time.

The presented mapping stage is a prototype and a lot of future work is necessary, among other things:

- The consequences of more complex systems are to be investigated, e.g. introduction of hierarchy levels and applicability of the path search algorithm, etc.
- The results of instruction and cycle accurate system simulation, which will be available in the future, are to be integrated in the mapping optimization.
- Mapping constraints imposed by the application programmer are to be taken into account.

A Task Description

A.1 Introduction

The design of future systems on chips requires a scalable hardware-software design approach. In the SHAPES (scalable software hardware architecture platform for embedded systems) project [2], a new hardware-software architecture paradigm is investigated.

Within the SHAPES project, TIK is dealing with the challenge of providing an efficient programming environment. As a first result, a software development framework called distributed operation layer (DOL) has been implemented. The purpose of the DOL is to enable the (semi-) automatic mapping of applications onto the multiprocessor SHAPES platform. In principle, this is achieved by

1. leveraging an appropriate model of computation, and
2. applying appropriate methods for mapping optimization.

The DOL will be used to develop streaming applications from such diverse areas as medical imaging, audio wave field synthesis, and physical modeling of complex systems. One important step towards this goal will be the implementation of a streaming application of moderate complexity using the DOL. Therefore, a primary task of this thesis is the implementation of an MPEG-2 decoder [12] for the DOL.

Based on this implementation, the thesis will then focus

1. on the evaluation of the DOL, suggestions for improvements, and their implementation, as well as
2. on the extraction of mapping-relevant parameters at the application level, the implementation of tools for this extraction, and — based on the extracted parameters — the mapping itself.

A.2 Tasks

The project will be divided into several subtasks, as described below. It is not unlikely that changes of these tasks will occur in the course of the project. If changes occur, they should be discussed with the advisors.

A.2.1 MPEG-2 Implementation for DOL

The first task in context of this thesis will be the implementation of an MPEG-2 decoder for the DOL in C. Preferably, the implementation will not be done from scratch but derived from an available implementation.

Subtasks

- Gaining of an understanding of DOL, the underlying model of computation [5], and the used communication API [4]
- Definition of selection criteria and according selection of a reference MPEG-2 implementation
- Partitioning of the algorithm into processes with the goal of exposing the available parallelism of MPEG-2 to the DOL
- Implementation of MPEG-2 on the DOL using the DOL API for communication between processes
- Functional verification of the implementation

- Comparison of the DOL implementation with MPEG-2 decoder implementations using other models of computation (for instance, SDF [26]) and development frameworks (for instance, Ptolemy [27] or StreamIt [28]).

A.2.2 Profiling for DOL Applications

In mapping optimization, an analytical model is used in which parameters are used to describe the application as well as the underlying architecture. Mapping optimization depends on an efficient acquisition of these parameters to achieve a good performance in terms of both run-time and results. In this thesis, one focus will be the implementation of a profiler to obtain parameter data at the application level.

Subtasks

- Identification of mapping-relevant parameters available at the application level
- Proposal of an appropriate method to profile these parameters
- Prototypical implementation of a profiler and integration into the existing SystemC simulation generator
- Demonstration of profiler capabilities using the MPEG-2 decoder

A.2.3 Mapping of DOL Applications

Based on the parameters extracted by the profiler, the last task is to investigate the mapping of DOL applications onto a heterogeneous multi-processor architecture. The goal is to automatically compute a mapping of the application onto the architecture. The envisioned workflow is depicted in Fig. A.1.
The optimization goal is balancing of resource usage:

- Minimization of the maximum load of processors
- Minimization of the maximum load of communication links

Only the binding part of mapping is considered, scheduling is not considered.

Subtasks

- Definition of an example architecture (two types of computation and communication resources)

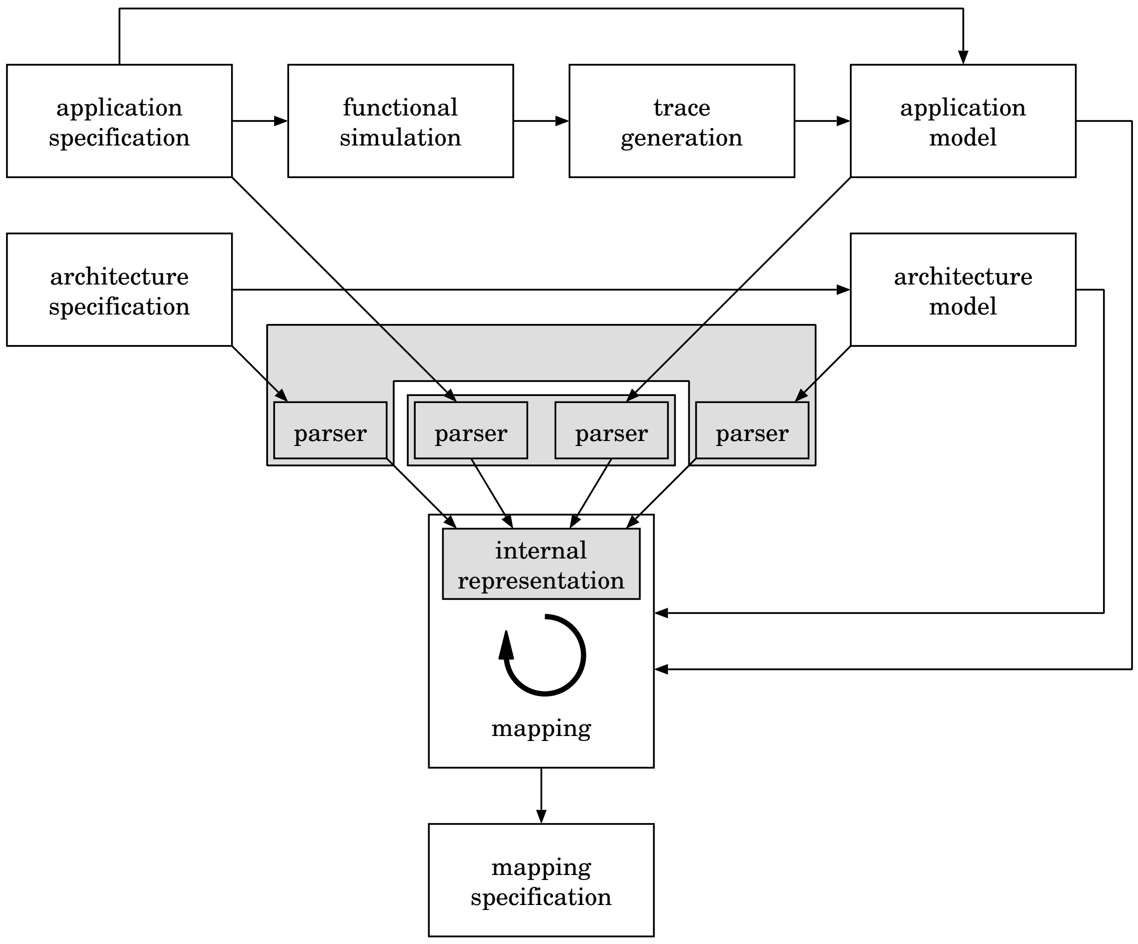


Figure A.1: Workflow for Automated Mapping.

- Definition of the relevant performance data
- Description of how the performance data are gathered, saved, and passed to the mapping stage
- Definition of the mapping stage (for the specific problem at hand): How do the internal representation look like? How is a new mapping generated (in terms of an algorithm)? How is a mapping evaluated (with respect to the optimization goals)?
- Identification of the problem-specific and problem-independent functions required in the mapping
- Specification of a suitable software-architecture for the problem-independent functions

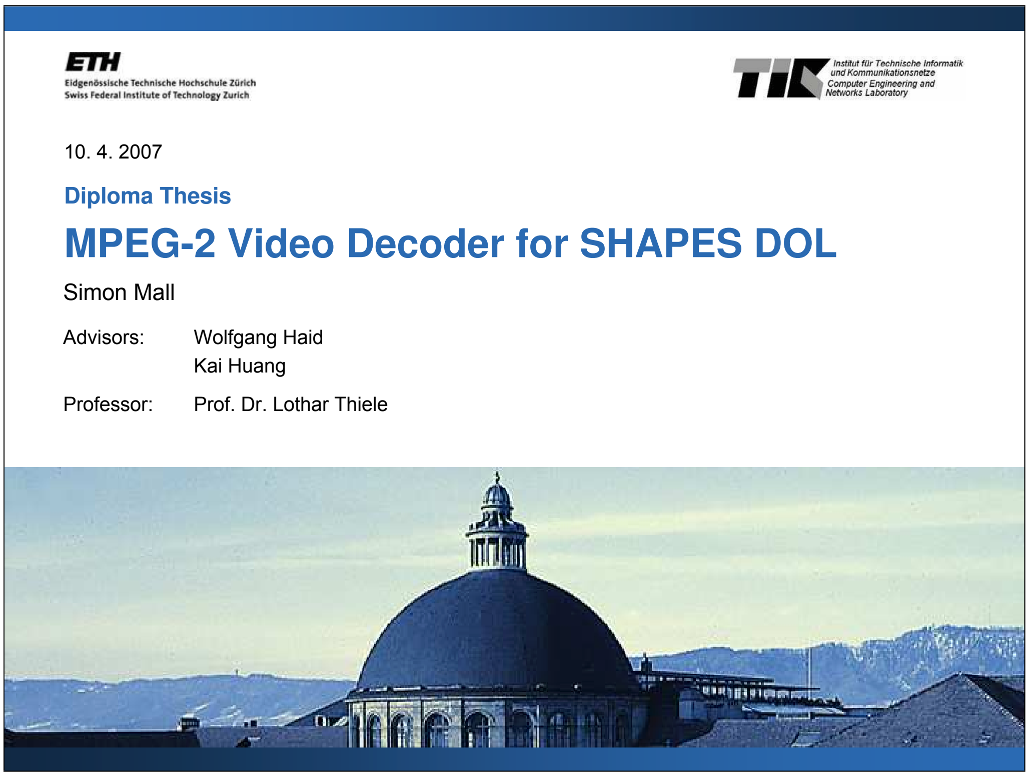
- Implementation of the system (using EXPO [25] for multi-objective optimization)
- Evaluation of the system for MPEG-2

A.3 Project Organization

There will be a weekly meeting to discuss the project's progress. A revision of the working document should be provided the day before.

Four copies of the written report are to be turned in. All copies remain the property of the Computer Engineering and Networks Laboratory.

B
Presentation Slides


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

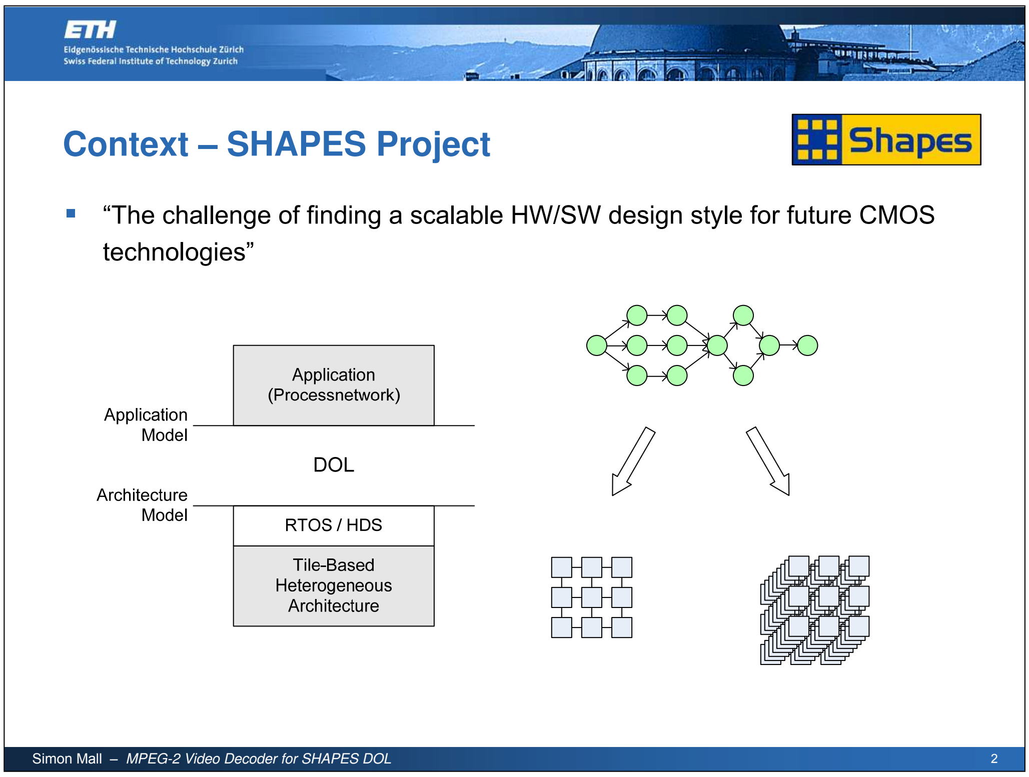
10. 4. 2007

Diploma Thesis

MPEG-2 Video Decoder for SHAPES DOL

Simon Mall

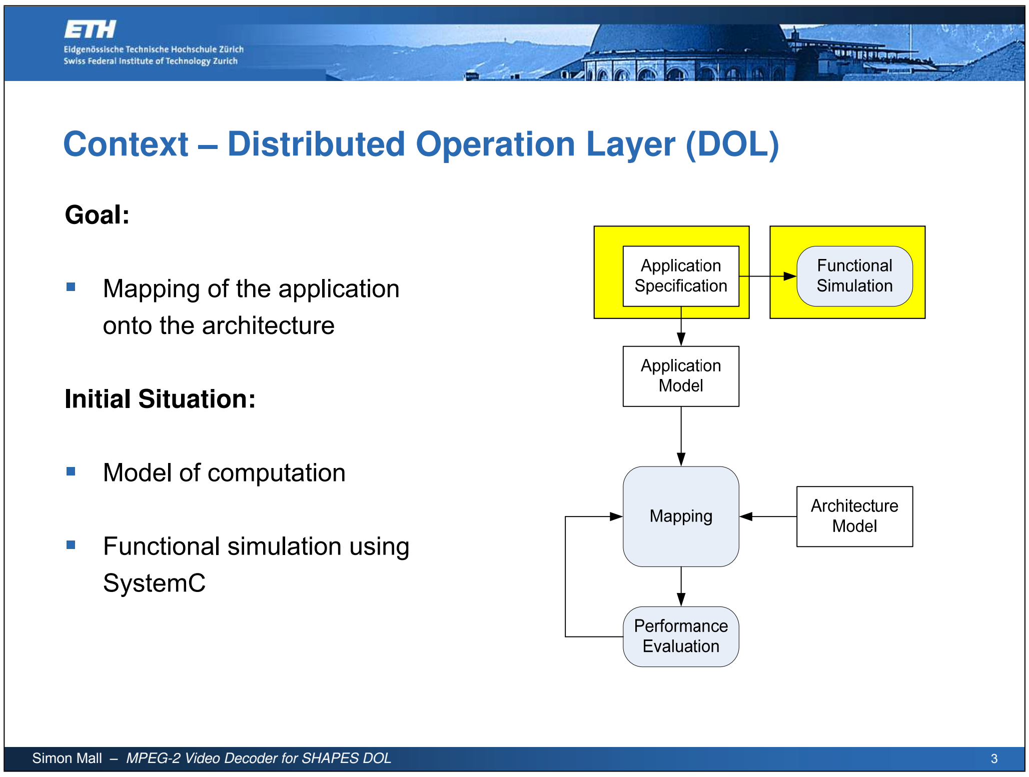
Advisors:	Wolfgang Haid
	Kai Huang
Professor:	Prof. Dr. Lothar Thiele



Context – SHAPES Project

- "The challenge of finding a scalable HW/SW design style for future CMOS technologies"

Shapes
Application (Processnetwork)
Application Model
DOL
Architecture Model
RTOS / HDS
Tile-Based Heterogeneous Architecture



Context – Distributed Operation Layer (DOL)

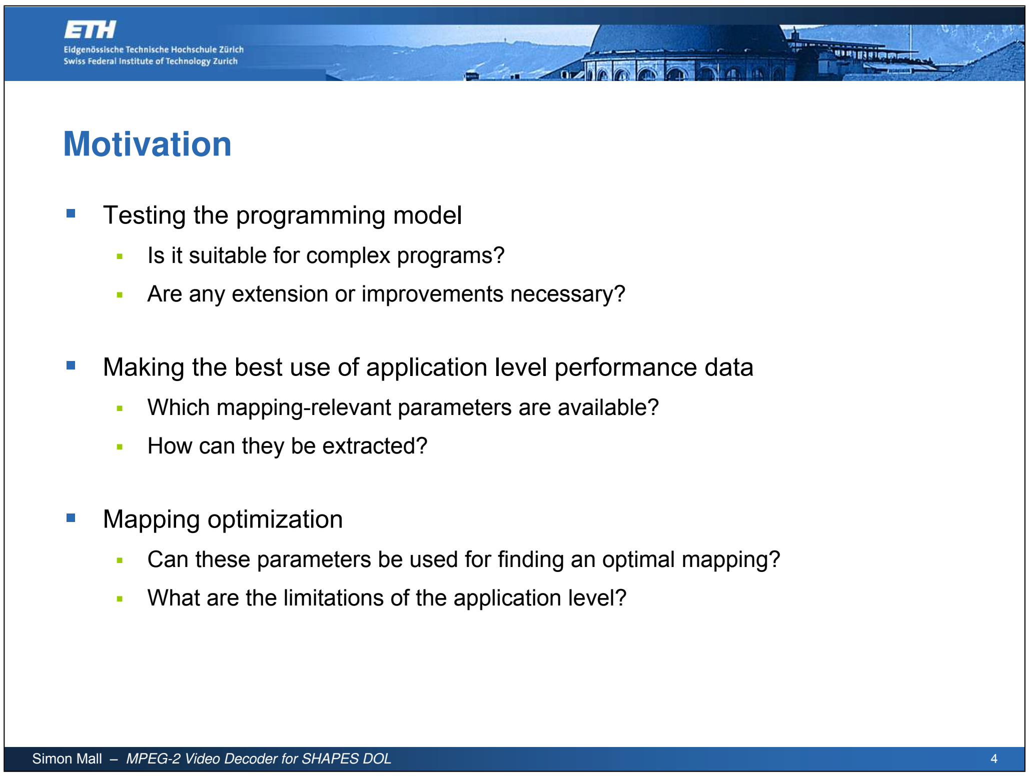
Goal:

- Mapping of the application onto the architecture

Initial Situation:

- Model of computation
- Functional simulation using SystemC

Application Specification
Functional Simulation
Application Model
Mapping
Architecture Model
Performance Evaluation

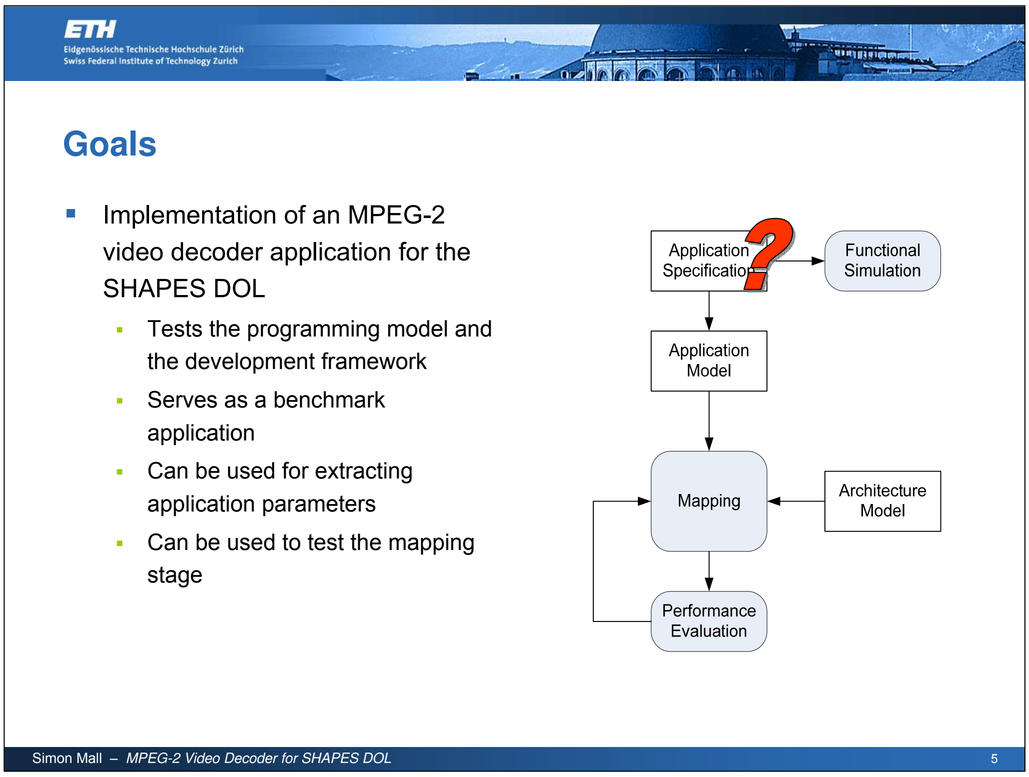


ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Motivation

- Testing the programming model
  - Is it suitable for complex programs?
  - Are any extension or improvements necessary?
- Making the best use of application level performance data
  - Which mapping-relevant parameters are available?
  - How can they be extracted?
- Mapping optimization
  - Can these parameters be used for finding an optimal mapping?
  - What are the limitations of the application level?

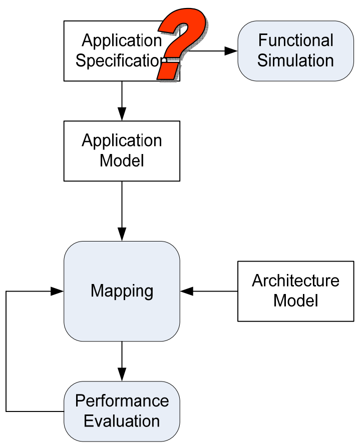
Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 4



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Goals

- Implementation of an MPEG-2 video decoder application for the SHAPES DOL
  - Tests the programming model and the development framework
  - Serves as a benchmark application
  - Can be used for extracting application parameters
  - Can be used to test the mapping stage

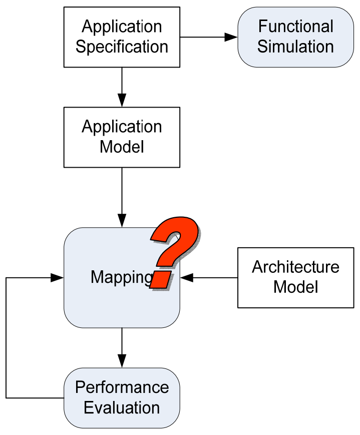


Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 5

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Goals

- Implementation of a mapping stage
  - How to model an application?
  - Limitations of the application level performance data?
  - How to optimize the mapping?
  - Demonstration of capabilities using the MPEG-2 application

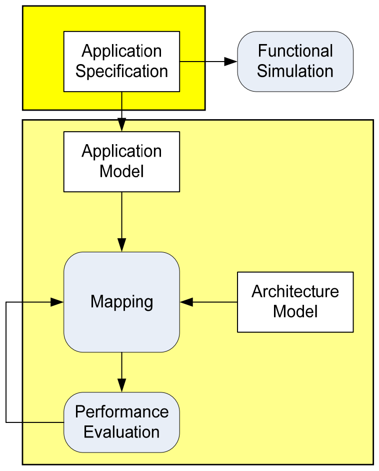


Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 6

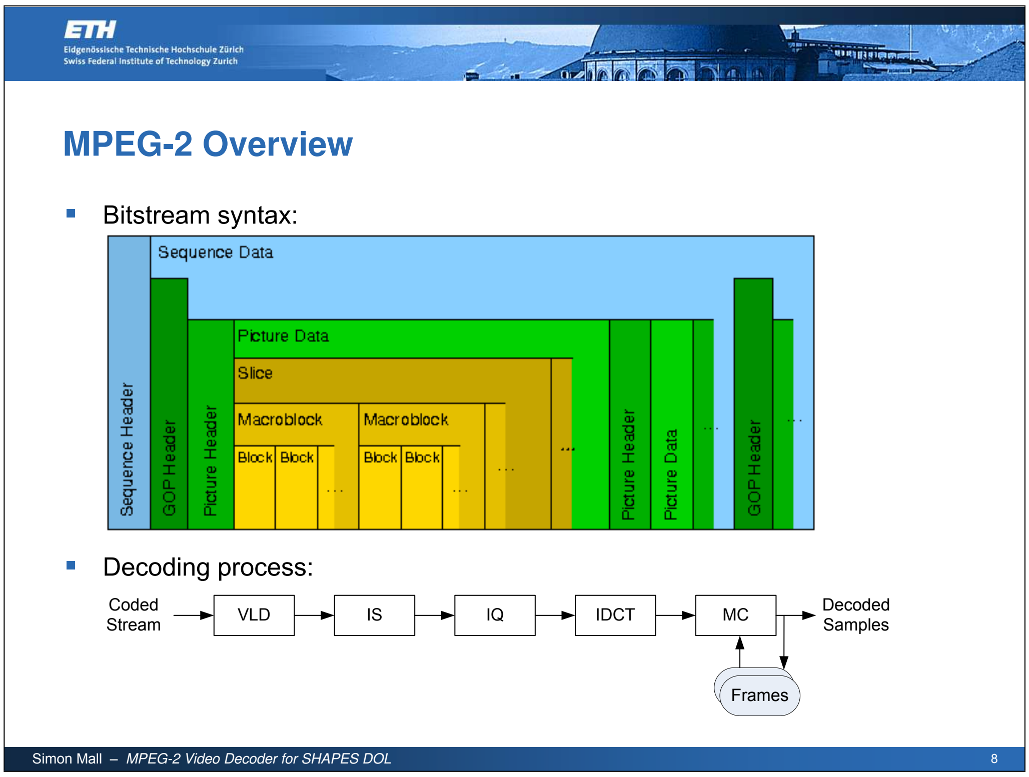
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Outline

- MPEG-2 decoder
  - MPEG-2 overview
  - Parallelization
  - Results
  - Conclusion
- Mapping stage
  - Mapping overview
  - Approach
  - Results
  - Conclusion



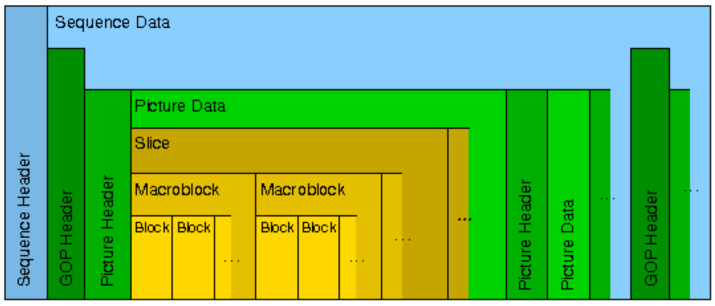
Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 7



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

MPEG-2 Overview

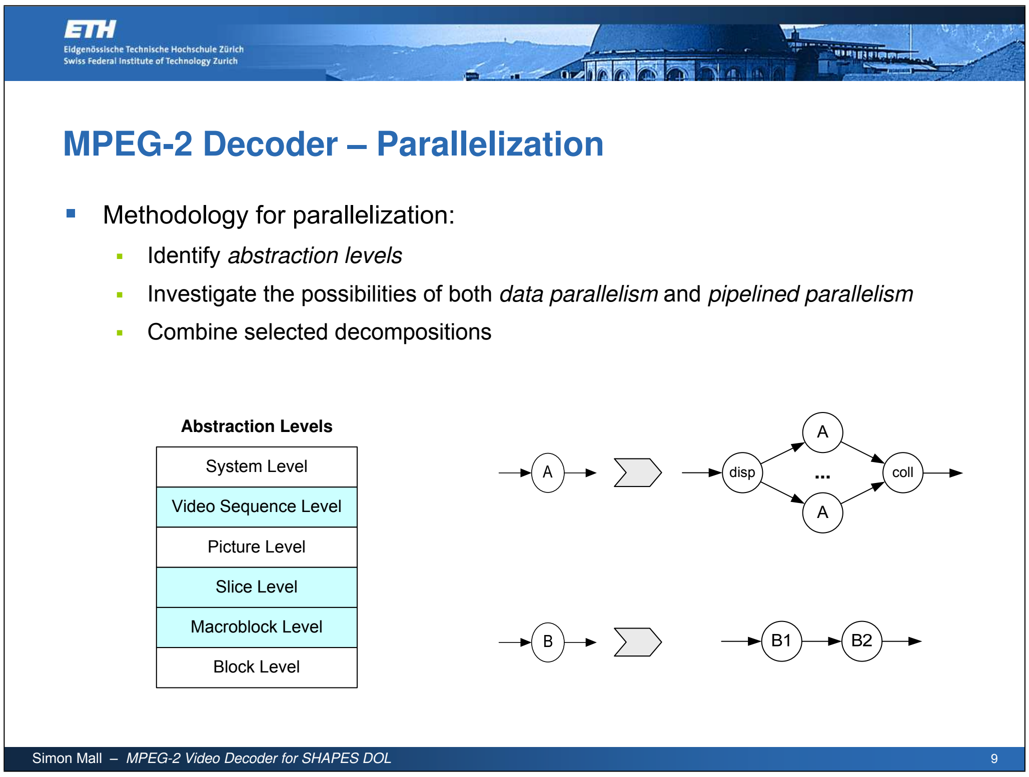
- Bitstream syntax:



- Decoding process:

$$\begin{tikzcd}
\text{Coded Stream} \arrow[r] & \boxed{\text{VLD}} \arrow[r] & \boxed{\text{IS}} \arrow[r] & \boxed{\text{IQ}} \arrow[r] & \boxed{\text{IDCT}} \arrow[r] & \boxed{\text{MC}} \arrow[r] \arrow[d, shift left] & \text{Decoded Samples} \\
 & & & & & \text{Frames} \arrow[u, shift left] &
\end{tikzcd}$$

Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 8



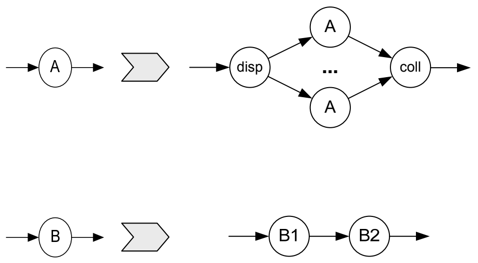
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

MPEG-2 Decoder – Parallelization

- Methodology for parallelization:
  - Identify *abstraction levels*
  - Investigate the possibilities of both *data parallelism* and *pipelined parallelism*
  - Combine selected decompositions

Abstraction Levels

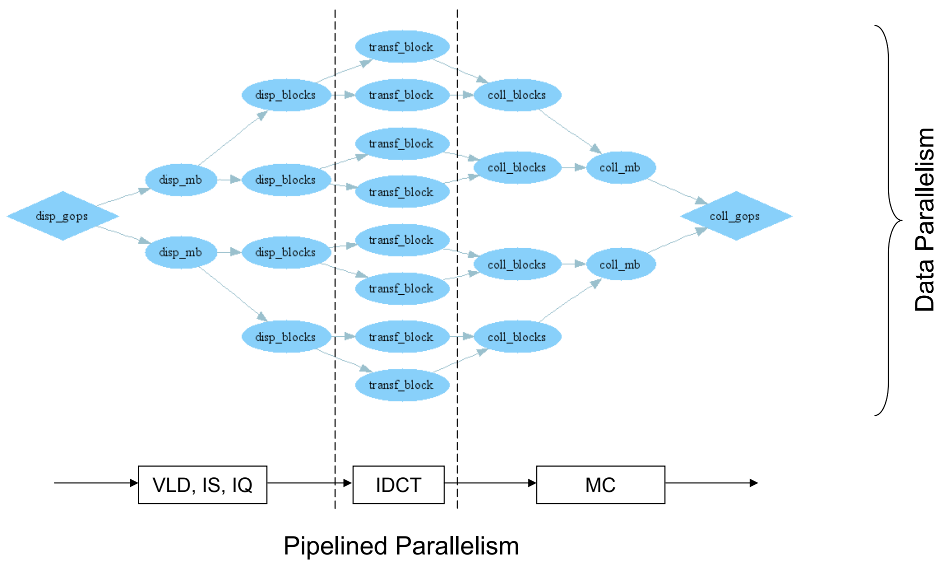
System Level
Video Sequence Level
Picture Level
Slice Level
Macroblock Level
Block Level



Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 9

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

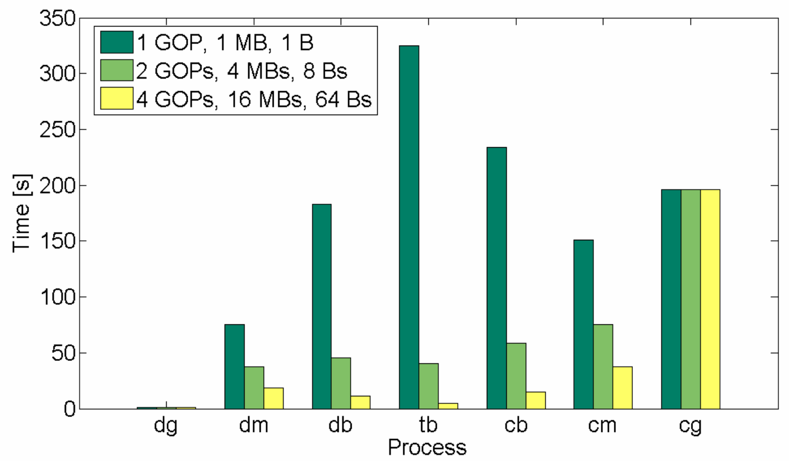
MPEG-2 Decoder – Resulting *Kahn Process Network*



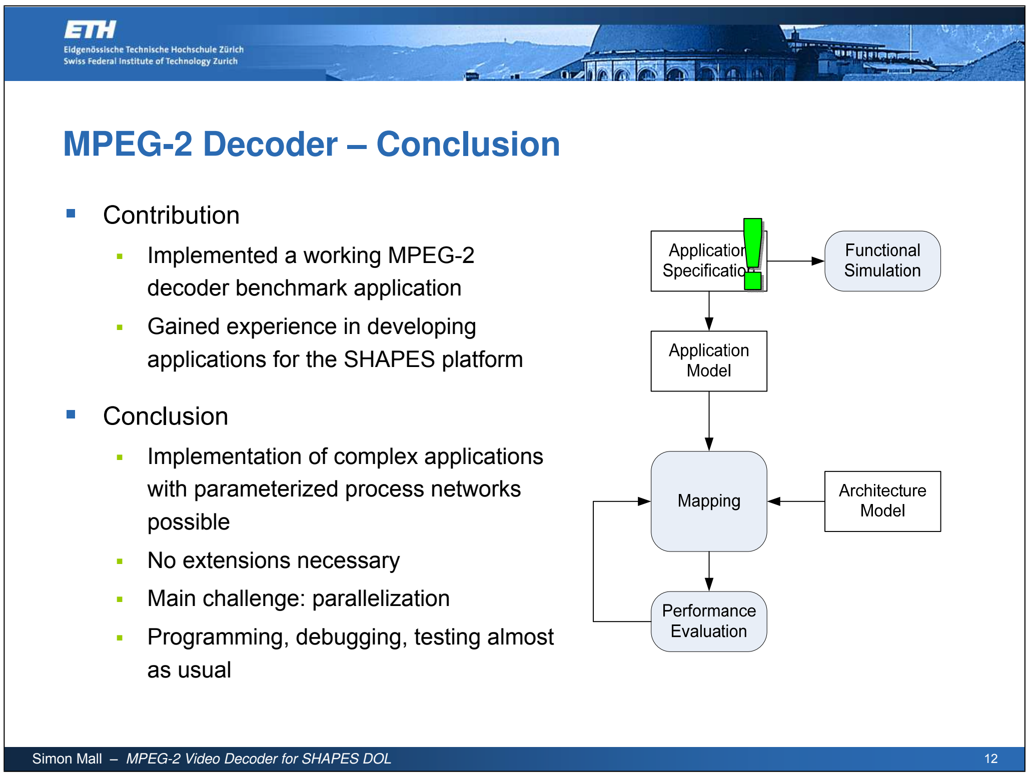
Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 10

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

MPEG-2 Decoder – Resulting Process Execution Times



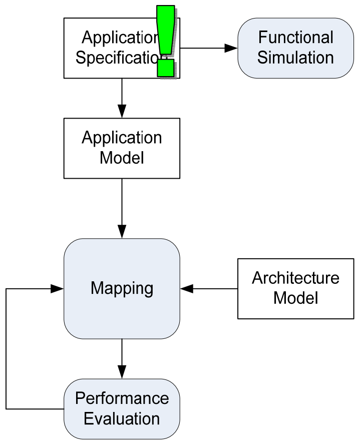
Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 11



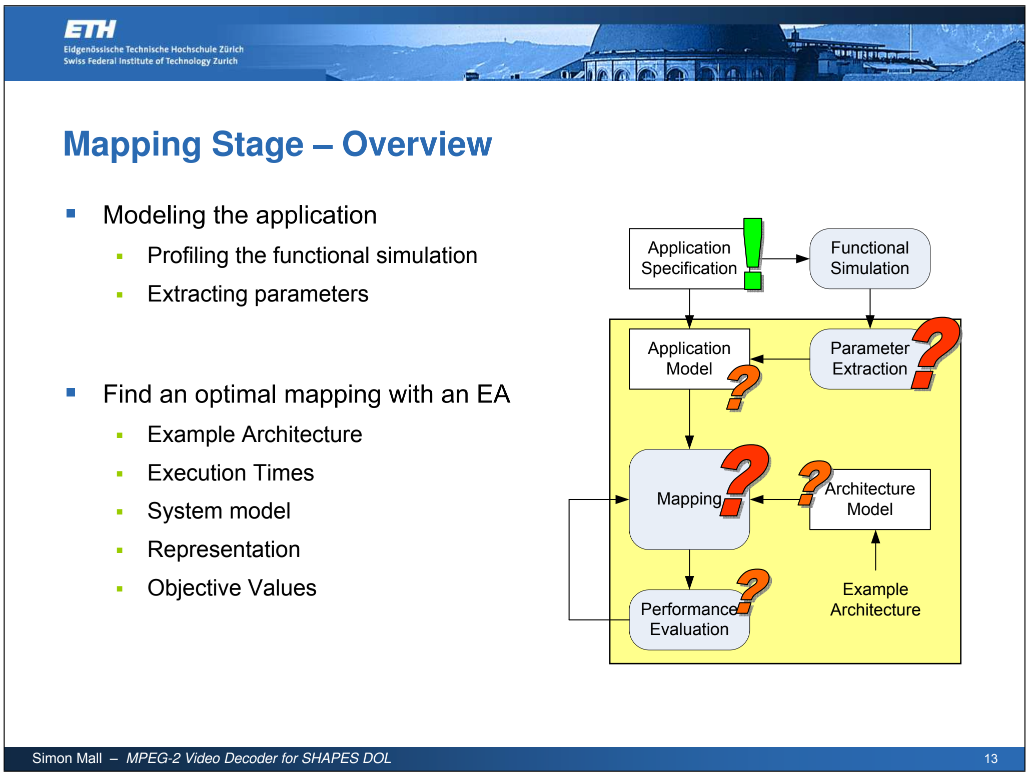
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

MPEG-2 Decoder – Conclusion

- Contribution
  - Implemented a working MPEG-2 decoder benchmark application
  - Gained experience in developing applications for the SHAPES platform
- Conclusion
  - Implementation of complex applications with parameterized process networks possible
  - No extensions necessary
  - Main challenge: parallelization
  - Programming, debugging, testing almost as usual



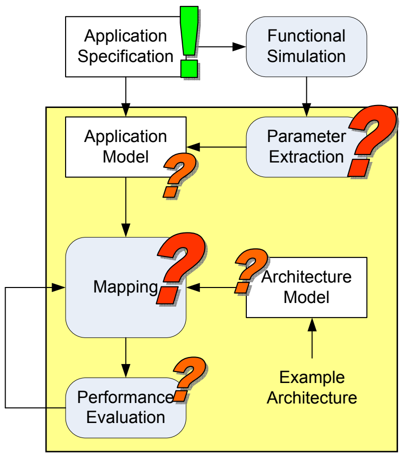
Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 12



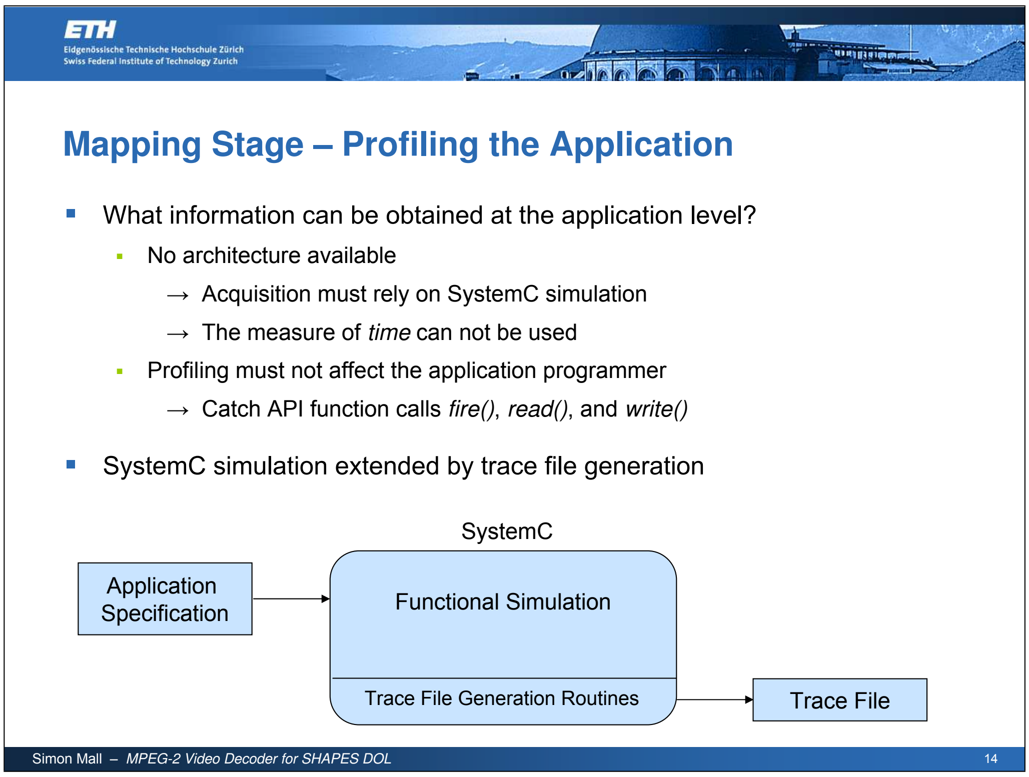
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mapping Stage – Overview

- Modeling the application
  - Profiling the functional simulation
  - Extracting parameters
- Find an optimal mapping with an EA
  - Example Architecture
  - Execution Times
  - System model
  - Representation
  - Objective Values

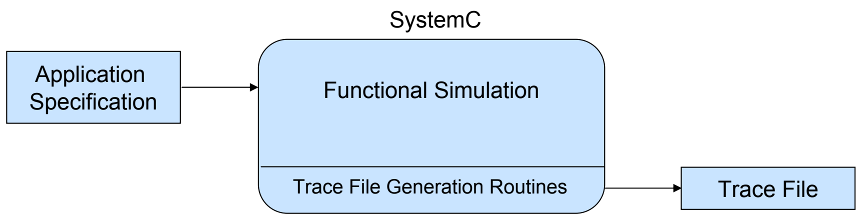


Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 13

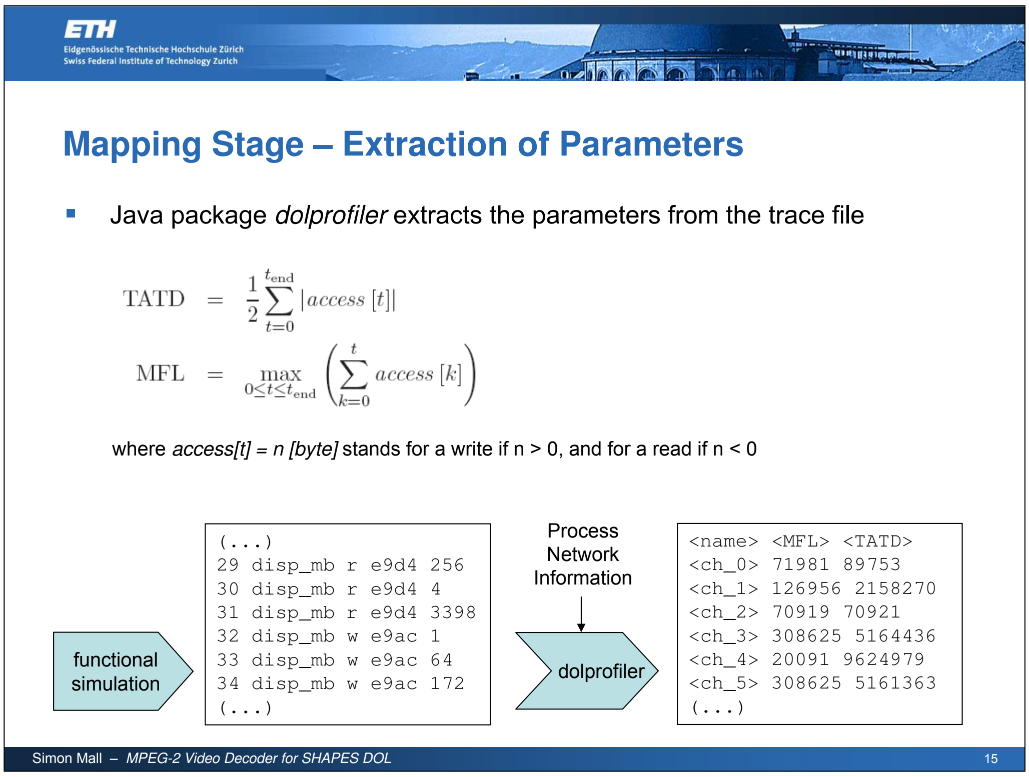


Mapping Stage – Profiling the Application

- What information can be obtained at the application level?
  - No architecture available
    - → Acquisition must rely on SystemC simulation
    - → The measure of *time* can not be used
  - Profiling must not affect the application programmer
    - → Catch API function calls *fire()*, *read()*, and *write()*
- SystemC simulation extended by trace file generation



Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 14



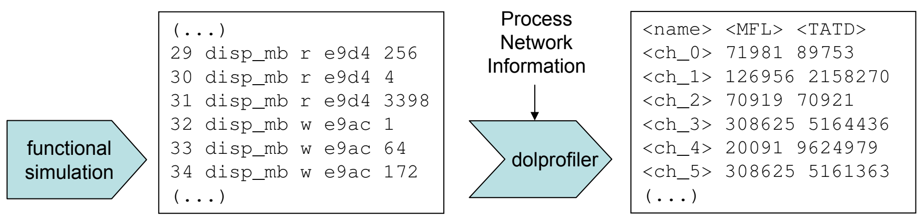
Mapping Stage – Extraction of Parameters

- Java package *dolprofiler* extracts the parameters from the trace file

$$\mathrm{TATD} = \frac{1}{2}\sum_{t=0}^{t_{\mathrm{end}}} |access\,[t]|$$

$$\mathrm{MFL} = \max_{0\le t\le t_{\mathrm{end}}} \left(\sum_{k=0}^{t} access\,[k]\right)$$

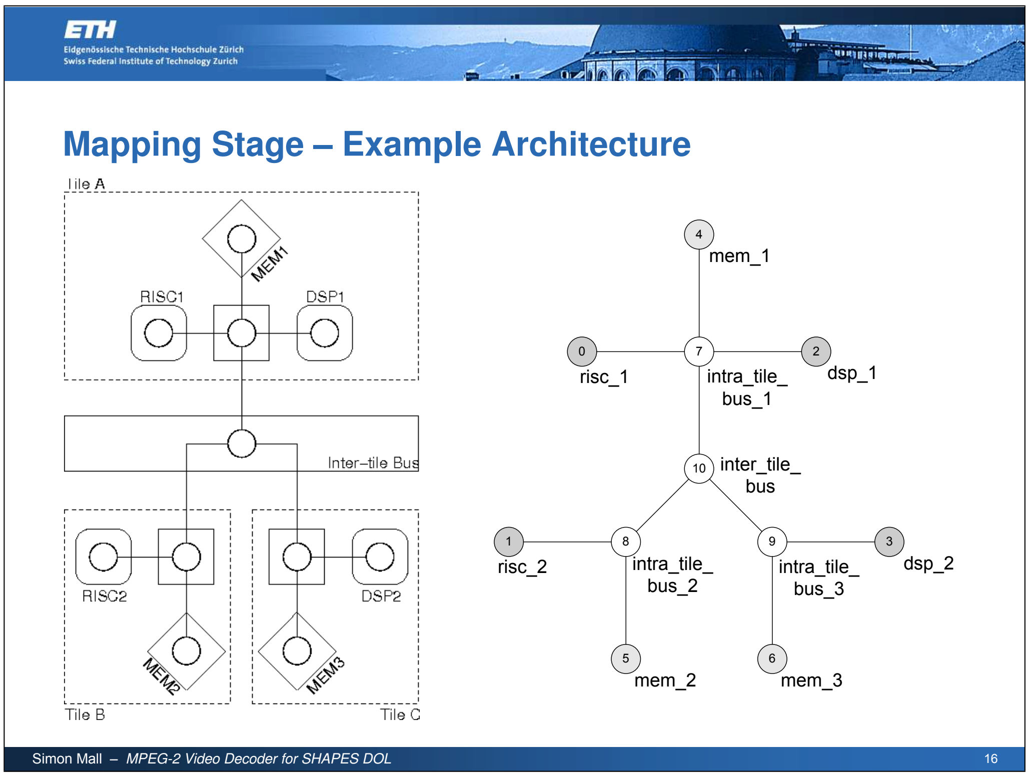
where *access[t] = n [byte]* stands for a write if n > 0, and for a read if n < 0



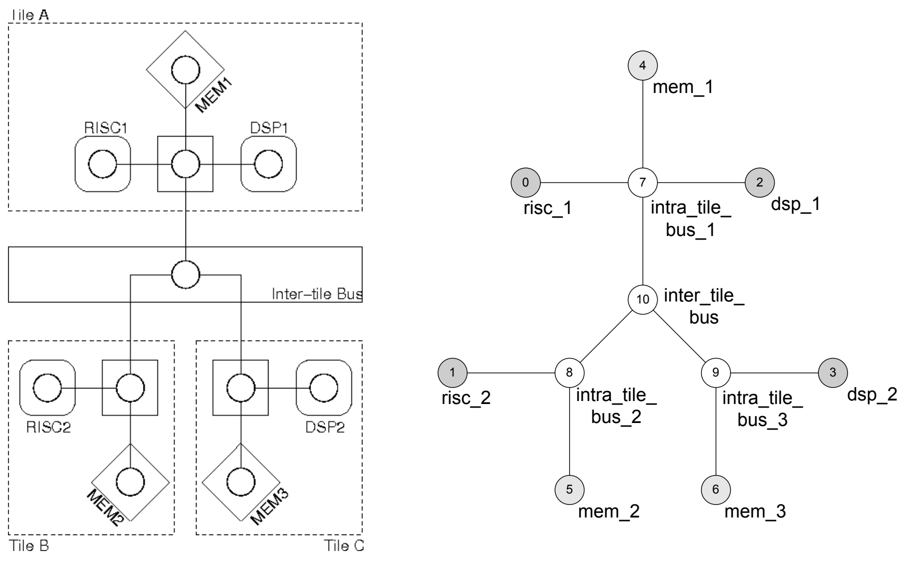
```
(...)
29 disp_mb r e9d4 256
30 disp_mb r e9d4 4
31 disp_mb r e9d4 3398
32 disp_mb w e9ac 1
33 disp_mb w e9ac 64
34 disp_mb w e9ac 172
(...)
```

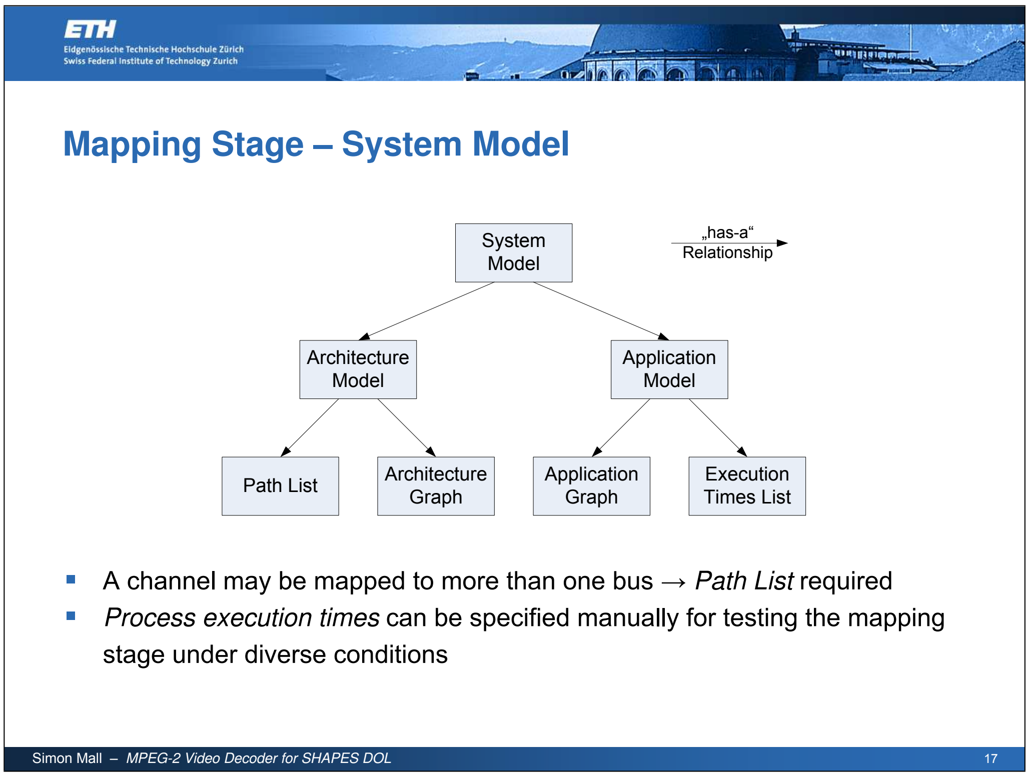
```
<name> <MFL> <TATD>
<ch_0> 71981 89753
<ch_1> 126956 2158270
<ch_2> 70919 70921
<ch_3> 308625 5164436
<ch_4> 20091 9624979
<ch_5> 308625 5161363
(...)
```

Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 15

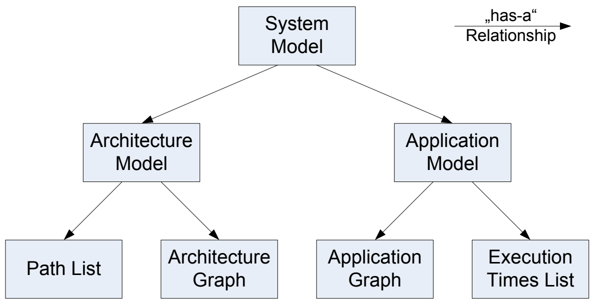


Mapping Stage – Example Architecture

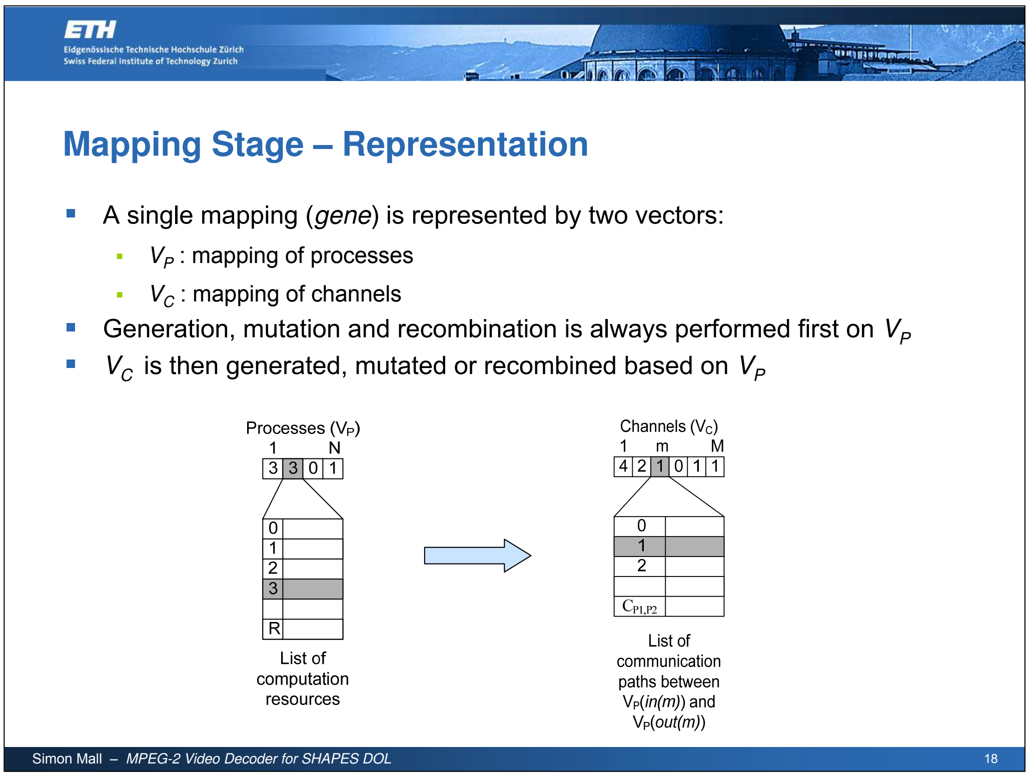




Mapping Stage – System Model



- A channel may be mapped to more than one bus → *Path List* required
- *Process execution times* can be specified manually for testing the mapping stage under diverse conditions



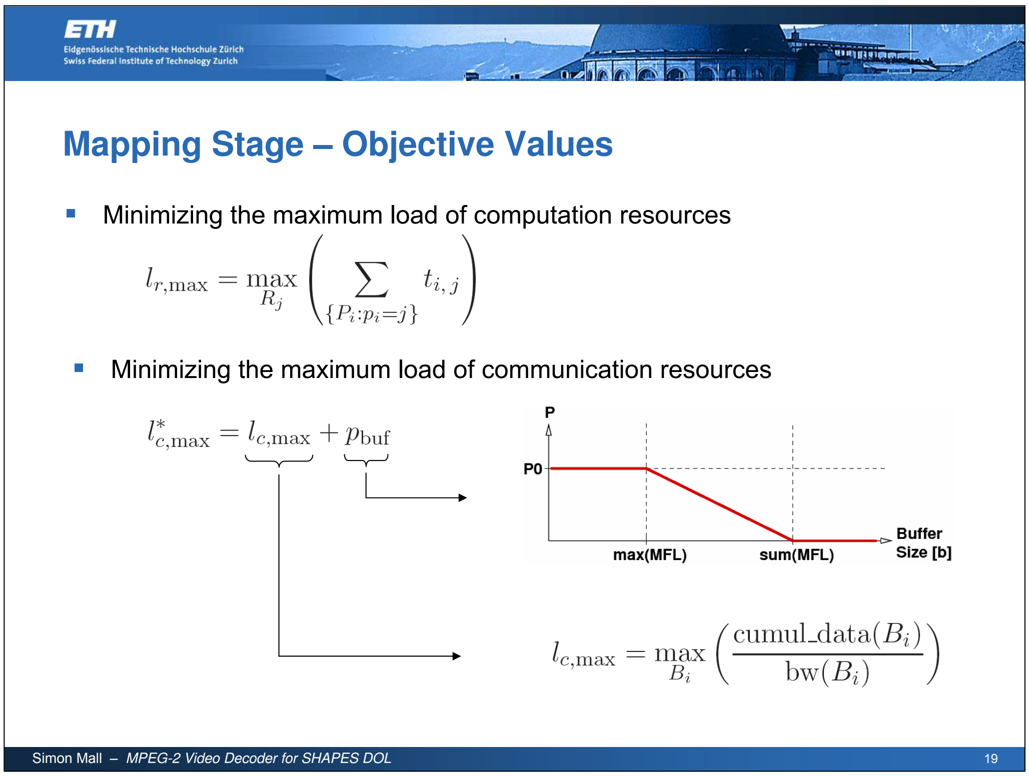
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Mapping Stage – Representation

- A single mapping (*gene*) is represented by two vectors:
  - V_P : mapping of processes
  - V_C : mapping of channels
- Generation, mutation and recombination is always performed first on V_P
- V_C is then generated, mutated or recombined based on V_P

Processes (V_P)
1 N
3 3 0 1
0
1
2
3
R
List of computation resources
Channels (V_C)
1 m M
4 2 1 0 1 1
0
1
2
C_P1,P2
List of communication paths between V_P(in(m)) and V_P(out(m))

Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 18



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Mapping Stage – Objective Values

- Minimizing the maximum load of computation resources

$$l_{r,\max} = \max_{R_j}\left(\sum_{\{P_i:p_i=j\}} t_{i,j}\right)$$

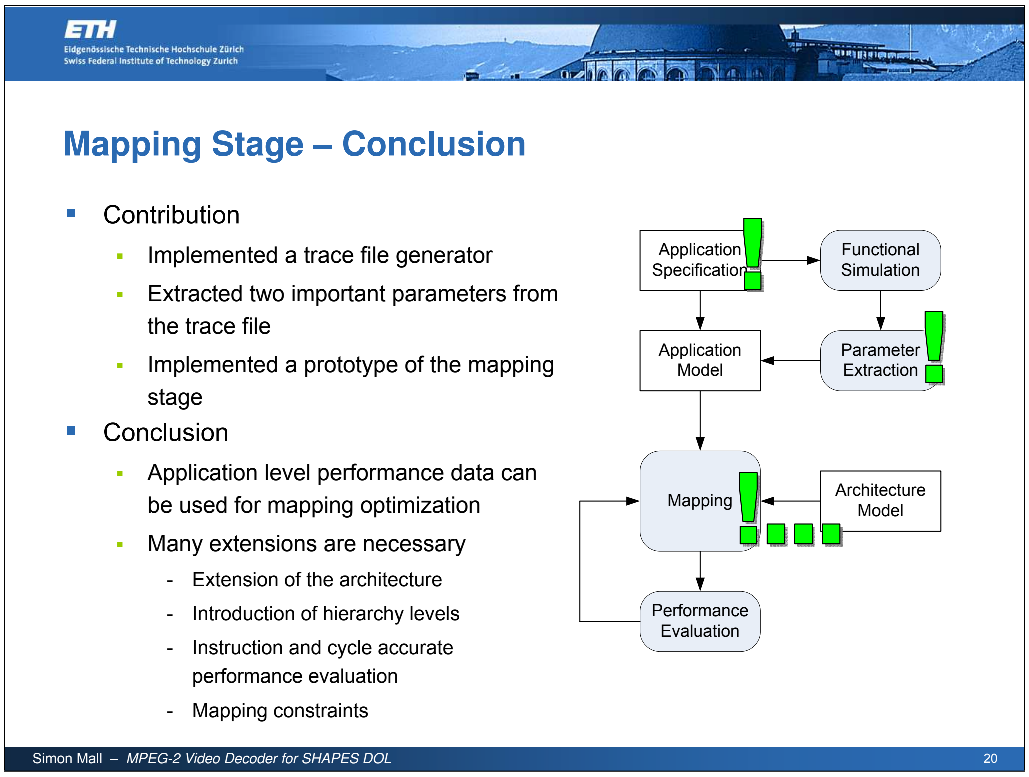
- Minimizing the maximum load of communication resources

$$l^*_{c,\max} = \underbrace{l_{c,\max}} + \underbrace{p_{\text{buf}}}$$

P
P0
max(MFL)
sum(MFL)
Buffer Size [b]

$$l_{c,\max} = \max_{B_i}\left(\frac{\text{cumul\_data}(B_i)}{\text{bw}(B_i)}\right)$$

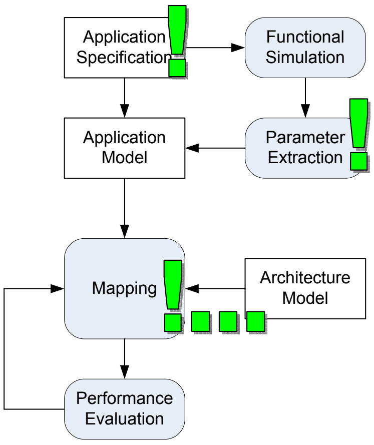
Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 19



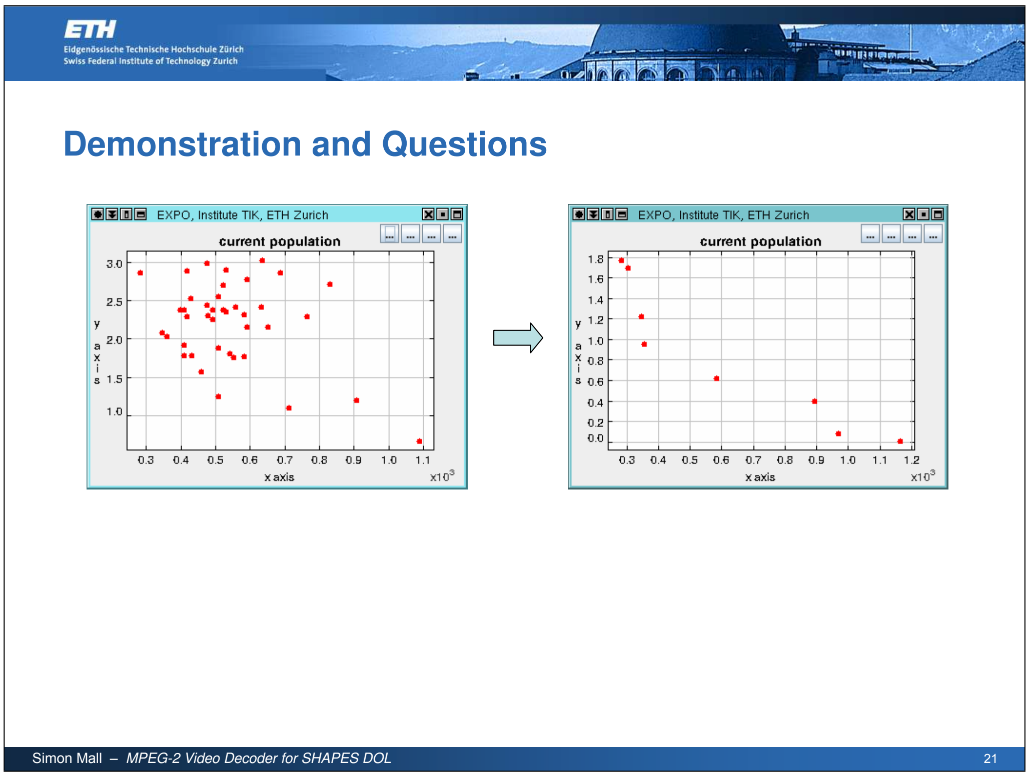
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Mapping Stage – Conclusion

- Contribution
  - Implemented a trace file generator
  - Extracted two important parameters from the trace file
  - Implemented a prototype of the mapping stage
- Conclusion
  - Application level performance data can be used for mapping optimization
  - Many extensions are necessary
    - Extension of the architecture
    - Introduction of hierarchy levels
    - Instruction and cycle accurate performance evaluation
    - Mapping constraints

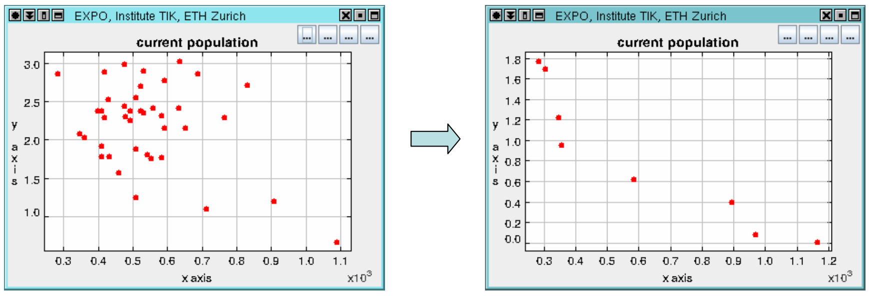


Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 20



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Demonstration and Questions



Simon Mall – *MPEG-2 Video Decoder for SHAPES DOL* 21

C
CD–ROM Contents

The top level directory structure of the CD is shown in Figure C.1. The next section explains the contents of each of these directories separately. Section C.2 contains step–by–step instruction for compiling and running the MPEG-2 decoder and the mapping stage.

C.1 Directory Contents

C.1.1 doc

The doc directory has two sub–directories. diploma_thesis contains the thesis in PDF format and its LaTeX–source. The presentation sub–directory contains the presentation

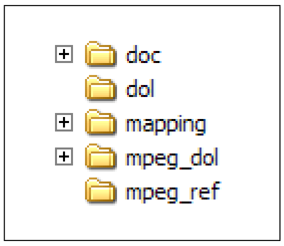


Figure C.1: Top level directory structure.

in the Microsoft Power Point format.

C.1.2 `dol`

The `dol` directory contains a single file `dol_ethz.zip`, which can be copied to a local directory and decompressed, in order to install and configure the version of the SHAPES DOL, on which this thesis is built up (including the trace file generating routines written in the context of this thesis). The archive itself contains a `docs` directory with instructions on how to use the DOL.

C.1.3 `mapping`

The `mapping` directory structure is shown in Figure C.2. The `mapping` directory has to sub–directories. `src` contains all Java source files written in the context of this thesis. `expo_shapes` contains all the files needed for running the mapping optimization with PISA/EXPO. `expo_shapes` itself contains two files and three sub–directories:

`build.xml` The ant script used to compile EXPO.

`do_chmod.sh` A small shell script, which automatically sets all necessary files to be executable. In Linux, this script must be run after each compilation step.

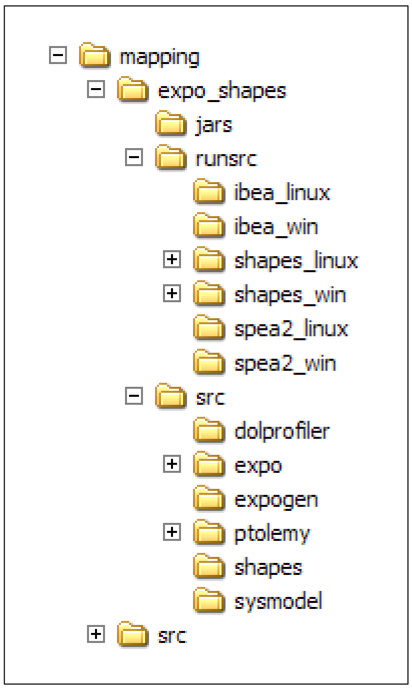
`jars/` This directory contains several libraries used by EXPO.

`src/` This directory contains all the sources of EXPO, the system model, the `dolprofiler` and `expogen`.

`runsrc/` This directory contains the files needed for running EXPO later. No changes should be made here. Instead, after compiling EXPO, a `build/` directory is created, where the specification and the configuration can be changed. See Section C.2 for details.

C.1.4 `mpeg_dol`

The `mpeg_dol` directory contains the source code of the MPEG-2 video decoder implementation for the SHAPES DOL, i.e. the process network definition (`example9.xml`) and a `src` sub–directory containing the process implementations. The DOL can be used to generate the SystemC source code, which can then be compiled and run on a single workstation. The `doc` sub–directory contains the doxygen documentation in



```
⊟ mapping
    ⊟ expo_shapes
        jars
        ⊟ runsrc
            ibea_linux
            ibea_win
            ⊞ shapes_linux
            ⊞ shapes_win
            spea2_linux
            spea2_win
        ⊟ src
            dolprofiler
            ⊞ expo
            expogen
            ⊞ ptolemy
            shapes
            sysmodel
    ⊞ src
```

Figure C.2: Mapping directory structure.

HTML format. Note that doxygen was not really intended to be used with this model of computation, which means that not all of the documentation pages are meaningful.

C.1.5 `mpeg_ref`

The `mpeg_ref` directory contains the MSSG's mpeg2dec, i.e. the reference implementation. All the necessary instructions on how to install and run the reference implementation can be found in the `readme.txt` file in this directory.

C.2 Step–by–Step Instructions

C.2.1 How to Install the DOL

1. Download and compile the latest version of SystemC from the SystemC–webpage [7]. Detailed installation instructions are available on the web–page.
2. Download and install Apache Ant version 1.6.5 or later.
3. Copy and unzip the file `dol/dol_ethz.zip` to a local directory of your choice (we call this directory the *dol base directory*.

4. Edit the `build_zip.xml` file in the *dol base directory*. Change the SystemC paths to the directories where you installed SystemC (note that the name of the directory containing the library depends on the platform, e.g. `lib-linux` or `lib-gccsparcOS5`). Make sure that `java` and `javac` refer to the Java Platform 5.0 executables. Otherwise, specify your Java 5.0 executables in the `build_zip.xml` file.
5. Configure the DOL by calling

   ```
   ant -f build_zip.xml config
   ```

6. Compile the DOL by calling

   ```
   ant -f build_zip.xml
   ```

7. Add the *dol base directory* path, the `<`*dol base directory*`>/bin/jdom.jar`, and the `<`*dol base directory*`>/bin/xercesImpl.jar` to the `CLASSPATH` environment variable.

The DOL is now ready to be used.

C.2.2 How to Compile and Run the MPEG-2 Decoder

This section assumes a working DOL (as described in the previous section).

1. Copy the source files from the `mpeg_dol` directory into a local directory of your choice (the `doc` sub–directory is not necessary).
2. Configure the implementation (the desired number of parallel processed entities) as described in Sections 2.4.2 and 2.4.3.
3. Flatten the process network (resolves the iterator objects):

   ```
   java dol.helper.flattener.XMLFlattener example9.xml Generator
   javac Generator.java
   java Generator > flat_ex9.xml
   ```

4. Generate the SystemC source code:

   ```
   java dol.main.Main -P flat_ex9.xml -c -H hds -D dotty.dot
   ```

5. Change to the directory `hds/src/`. If you want to turn of the DOL profiling (the trace file generation slows down the simulation and needs a large amount of disk space), edit the `Makefile` and remove the `-DINCLUDE_PROFILER` preprocessor argument.

6. Compile the SystemC application with GNU make.
7. Run the application by calling

   ```
   ./sc_application
   ```

8. Enter the filename of the MPEG-2 video clip when prompted.

Note that the decoder accepts `.mpeg` files, but it is better to extract the video sequence and use the `.m2v` video files instead. For measuring the execution times, the filename prompt can be omitted by writing the filename into a text file `run.txt` and calling the application with

```
./sc_application < run.txt
```

(make sure that the `run.txt` file ends with an empty line).

If you turned on the DOL profiler, a `profile.txt` file is generated each time the simulation is finished. This file is needed in the mapping stage (see next section).

C.2.3 How to Compile and Run the Mapping Stage

This section assumes a working DOL. Especially, it is necessary that the path to the dol base directory *is contained in the `CLASSPATH` environment variable.*

1. Copy the whole `mapping/expo_shapes` directory from the CD to a local directory of your choice.
2. Compile EXPO by calling

   ```
   ant linux
   sh do_chmod.sh
   ```

   on a Linux environment, or

   ```
   ant win
   ```

   on a Windows environment.
3. Open a second shell. We call the first one *shell A* and the second one *shell B*.
4. In *shell A*, change to the directory `build/run/expo`.

5. If you just want to try it out, without any configuration, just call `./runEXPO`. If you want to build your own EXPO specification file, the following steps are necessary:

   - Copy the `profile.txt` and a `flat_ex9.xml` files from the MPEG-2 simulation into the `spec` directory.
   - Edit the `spec/exec_times.txt` file and adjust the execution times.
   - Edit the `spec/arch.xml` file, if you want to change the architecture.
   - Generate the specification file by calling (from the `build/run/expo` directory)

     ```
     ./runEXPOGEN spec/flat_ex9.xml spec/arch.xml spec/profile.txt
     spec/exec_times.txt spec.txt
     ```

     (on a single line). This generates the specification file, which is used by EXPO. EXPO does not use the files in the `spec` directory, so any changes must be followed by invoking `runEXPOGEN` again. EXPO can then be run by calling `./runEXPO`.

6. In *shell B*, change to the directory `build/run/spea2_linux`, `build/run/spea2_win`, `build/run/ibea_linux`, or `build/run/ibea_win`, depending on your platform and the desired selector algorithm.
7. Run the selector by calling `./run_spea2` or `./run_ibea`, respectively.

Acknowledgments

I wish to thank the following people:

- **Prof. Dr. Lothar Thiele** for making it possible to write this diploma thesis at the Computer Engineering and Networks Laboratory.
- My thesis advisors, **Wolfgang Haid** and **Kai Huang** for the huge effort they put into this project. Their constant support and assistance was a great help, and the meetings and discussions were always enriching.
- The Services Group of the Computer Engineering and Networks Laboratory for providing a perfect infrastructure and working environment.
- My family and friends for their supportive and motivating attitude, and for understanding my temporary lack of time.

Bibliography

[1] R. Ho, K. Mai, and M. Horowitz, “The Future of Wires,” in *Proc. IEEE*, vol. 89, no. 4, 2001, pp. 490–504.

[2] SHAPES Project Website. [Online]. Available: http://www.shapes-p.org

[3] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini, “SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems,” in *Proc. of the 4th Int. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS’06)*, Seoul, Korea, Oct. 2006, pp. 167 –172.

[4] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers, “YAPI: Application Modeling for Signal Processing Systems,” in *Proc. 37th Design Automation Conference (DAC 2000)*, Los Angeles, CA, USA, June 2000, pp. 402–405.

[5] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” in *Proc. IFIP Congress 74*, North Holland Publishing Co.

[6] “System software requirements for heterogeneous multi-core systems,” SHAPES internal document.

[7] SystemC Community Website. [Online]. Available: http://www.systemc.org/

[8] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An Introduction to MPEG-2*. New York: Chapman and Hall, 1997.

[9] D. T. Hoang and J. S. Vitter, *Efficient Algorithms for MPEG Video Compression*. Chichester: John Wiley & Sons, 2002.

[10] Official Website of the MPEG Committee. [Online]. Available: http://www.chiariglione.org/mpeg

[11] J. Watkinson, *The MPEG Handbook.* Boston: Focal Press, 2001.

[12] *ISO/IEC 13818-2: Information technology — Generic Coding of moving pictures and associated audio information — Part 2: Video*, International Organization for Standarization, 1995.

[13] libmpeg2 Website. [Online]. Available: http://libmpeg2.sourceforge.net

[14] FFmpeg Multimedia System Website. [Online]. Available: http://ffmpeg.mplayerhq.hu

[15] MPEG Software Simulation Group (MSSG). [Online]. Available: http://www.mpeg.org/MSSG

[16] E. Iwata and K. Olukotun, "Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm," in *Technical Report CSL-TR-98-771*, Stanford University Computer Systems Laboratory, Sept. 1998.

[17] A. Bilas, J. Fritts, and J. P. Singh, "Real–Time Parallel MPEG-2 Decoding in Software," in *IEEE Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[18] Jahshaka Website. [Online]. Available: http://www.jahshaka.org

[19] Helix Player Website. [Online]. Available: https://player.helixcommunity.org

[20] I. Foster, *Designing and Building Parallel Programs.* Addison-Wesley, 1995. [Online]. Available: http://www.it.uom.gr/teaching/dbpp

[21] E. Zitzler, "Evolutionary Algorithms for Multiobjective Optimization," in *Evolutionary Methods for Design, Optimisation, and Control.* CIMNE, Barcelona, Spain, 2002, pp. 19–26.

[22] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA — A Platform and Programming Language Independent Interface for Search Algorithms," in *Evolutionary Multi-Criterion Optimization (EMO 2003)*, ser. Lecture Notes in Computer Science, C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, Eds. Berlin: Springer, 2003, pp. 494 – 508.

[23] PISA Website. [Online]. Available: http://www.tik.ee.ethz.ch/pisa/

[24] EXPO Website. [Online]. Available: http://www.tik.ee.ethz.ch/pisa/variators/expo/expo.html

[25] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, "A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures," in *Proc. 39th Design Automation Conference (DAC 2002)*, New Orleans, LA, USA, June 2002, pp. 880–885.

[26] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[27] E. A. Lee. Ptolemy Web Site. [Online]. Available: http://ptolemy.eecs.berkeley.edu

[28] S. Amarasinghe. StreamIt Web Site. [Online]. Available: http://cag.csail.mit.edu/streamit