Design and Implementation of a Simulator for Measuring the Quality of Service for Distributed Multimedia Applications

Rolf Esbjørn Kristensen

Kongens Lyngby 2009
IMM-M.Sc.-2009-XX

Draft

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc.: ISSN XXXX-XXXX

Abstract

Multimedia applications have become widespread today, from streaming over the Internet to portable music and video players. Increasingly multimedia applications are implemented using embedded architectures, which have very tight constraints in terms of cost, performance, power consumption, size, etc. Examples of such systems are smart-phones, an iPod or a TV set-top-box.

Designing embedded systems implementing multimedia applications is difficult because of the inherent variability of functionality execution times (which depend on the video or audio streams processed, their resolution, frame rate, etc.) and stringent Quality-of-Service (QoS) requirements on their performance (e.g., a playback of 25 frames per second for a video device).

Real-time systems theory provides analysis methods that can determine if an application implemented on an embedded architecture meets its timing constraints. There are a lot of results for hard real-time systems, which have to meet their deadlines even in the worst-case, otherwise something catastrophic can happen. In contrast, a multimedia application is a soft real-time system, where certain degradation of performance can be accepted, provided it is not below a given level of QoS.

Multimedia systems are difficult to analyse using existing schedulability analysis theory. Designing an architecture based on the worst-case leads to over design: too much computing power, which is seldom used. Hence, the focus of this thesis is to implement a simulator that can support the designer in evaluating very early in the development process several embedded architecture implementations, and deciding which one meets the QoS requirements for a given multimedia application. This can reduce the time-to-market and development costs by avoiding building a physical prototype, which is costly and time-consuming.

Besides evaluating hardware architectures (CPUs, dedicated hardware, buses), we are also interested in using the simulator to evaluate several scheduling policies, which have a strong impact on the behaviour of the application. Our simulator, which is based on the SystemC library, can take into account Fixed-Priority preemptive scheduling (FP), Earliest Deadline First scheduling (EDF), the Linux 2.6 scheduler and Constant-Bandwidth Server scheduling (CBS). The idea of the CBS is to divide a resource (CPU or bus) into virtual resources, which are given a certain budget. This is especially useful if several applications (both hard and soft real-time) have to share the same architecture.

The simulator has been used to evaluate several architecture alternatives for a set-top-box application, using different hardware components and scheduling policies.

As the experiments show, using the simulator we can chose very quickly the right architecture and scheduling policy. The simulator can also help in deciding the scheduling parameters, such as the bandwidth for the CBS servers.

Resumé

!!! Danish abstract !!!

Preface

This master thesis was written at the department of Informatics and Mathematical Modelling at the Technical University of Denmark DTU. The project was completed in the period September 2008 to March 2009, under the supervision of associate professor Paul Pop. The thesis is a 30 ECTS point course.

I would like to thank Paul Pop, and Ph.d. Prabhat Kumar Saraswat for guidance and support throughout the project. I would also like to thank my father and Sune Jakobsen for proofreading the thesis.

Rolf Esbjørn Kristensen
rolf@lightsaber.dk
Kongens Lyngby, March 2009

Contents

Abbreviations

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

CAS Cycle-Accurate Simulator

DCT Discrete Cosine Transformation

DiMAS Distributed Multimedia Application Simulation API

FIFO First In First Out

IDCT Inverse Discrete Cosine Transformation

ISA Instruction Set Architecture

ISS Instruction Set Simulator

MP Motion compensation Prediction

MPSoC Multi Processor System-on-Chip

P_e Processing Element

RTL Register Transfer Level

SAM System Architectural Model

TLM Transaction Level Modelling

VCD Value Change Dump

vLan Virtual Local Area Network

VLC Variable Length Coding

VLD Variable Length Decoding

VoIP Voice over Internet Protocol

WiFi Wireless Fidelity

Chapter 1

Introduction

Distributed multimedia applications are more and more common nowadays. Distributed multimedia can be many things from streaming over the Internet to a home entertainment system and portable MP3 players.

Internet TV and radio are widely used today, multimedia is however much more demanding in terms of service compared to ordinary Internet surfing and e-mailing. The Internet is based on a best effort service, which is not able to make any guarantees, as to when the network packets will arrive at the destination. Therefore the Internet is far from optimal for streaming multimedia, but this is mostly a problem, when the route from source to destination is saturated, or even resource limited.

The new thing in home entertainment is that all the devices are interconnected using a wireless network, thereby reducing all the cabling. WiFi is however far from an optimal solution for multimedia, since the quality of the links is even worse than a wired Internet. Many solutions are however built in such a way that the multimedia devices have a dedicated network, where no other devices are connected[1]. But this does not remove interference from other wireless networks in the vicinity.

Multimedia applications consist of various tasks that are mapped onto different processing elements. The purpose of partitioning the functionality is that dedicated decoding hardware devices can be used for doing some specific tasks in order to optimize performance. A further advantage of this distribution is that multi purpose processors also can be used to reduce cost. In this thesis we look into how different scheduling algorithms affect the Quality of Service (QoS) of multimedia. Possible QoS metrics for video streams are for example *frame rate* and *resolution*. Section 2.2 will elaborate on the various metrics, which define the QoS for multimedia applications.

Multimedia covers both video, audio and a combination of the two. This can be DVD film, Internet radio or TV. Conferencing systems and other types of communication like Internet phones are also defined as being multimedia applications. Generally multimedia is *Continuous Media* (CM), which differs from ordinary files, in the

[1] For example using a vLan for the home entertainment centre and a vLan for the rest.

sense that multimedia files are accessed at a specific rate throughout the playback. Multimedia therefore requires that the multimedia data is ready at the output device at a certain rate, in order ensure satisfactory playback.

The requirement of delivering data at a certain rate is the reason for multimedia applications being defined as real-time systems. Real-time systems can however be divided into two classes, there are hard real-time and soft real-time systems, where multimedia falls within the soft real-time systems class.

Hard real-time systems are time critical systems, where missing deadlines can have catastrophic consequences. A typical example is the Anti-lock Braking System (ABS) in the car industry, where the breaking distance can be considerably greater if the ABS tasks is not completed within the required deadlines.

Soft real-time systems are time sensitive systems, where missing deadlines do not have catastrophic consequences, but will have a negative effect on the performance of the task. Clearly multimedia tasks fall into the soft real-time class, since degradation of the playback is not a catastrophe, but simply an annoyance for the end user.

1.1 Variability in Multimedia

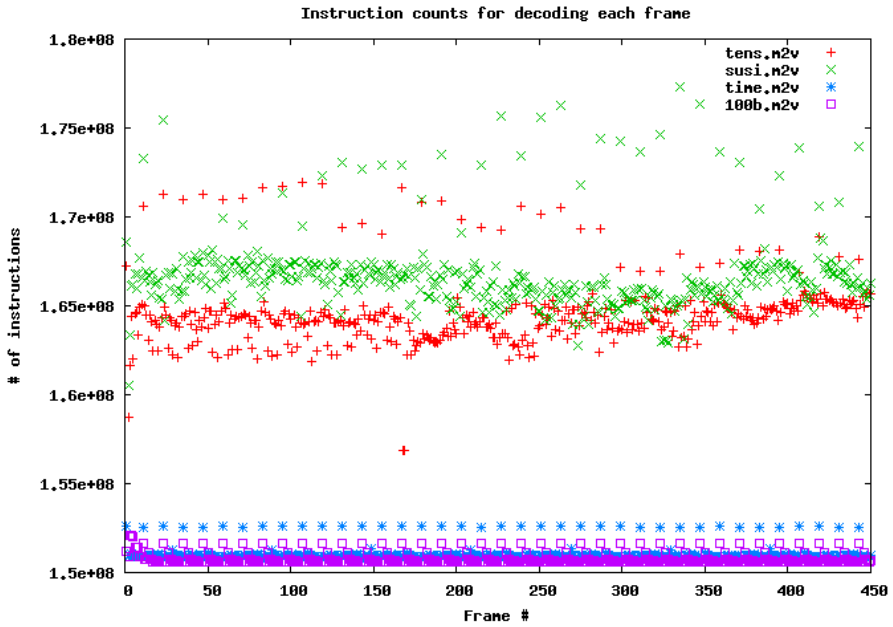

Figure 1.1: *Plot of the instruction count for decoding a film over time. Both the intra- and the inter variability is clearly seen.*

The problem with multimedia is the variability in the resource demand in terms of execution times. This thesis has identified two different types of variability, one is *intra variability* and the other *inter variability*. The intra variability is within the same multimedia stream, and inter variability is across different multimedia streams. Figure 1.1 shows the variability in terms of number of instructions required

for decoding frames for four different MPEG2 streams. It is clear that each of the four streams have intra variability, since they each have high peaks once in a while during the 100 frames. Further the inter variability is obvious since two streams require less instruction counts for processing the frames. The streams that require the most instruction counts have motion in the video, while the other two videos simply show a still image.

Designing the system based on the worst case execution time will lead to a system where the QoS is ensured. If the system on the other hand is designed based on the average execution times for all four streams in figure 1.1, then the performance for the processor heavy streams will be very bad, and many frames will probably be lost.

1.2 Motivation

The variability mentioned in the previous section is the main motivator for implementing this simulator. Systems designers might have an idea of what the resource requirements are for a specific appliance, but this is often only qualified guesses. For example there are many factors which are in play when creating a portable MP3 player. The player must not be over- or under designed in terms of resources such as memory and processor speed. If the player is under designed the player is useless, while if the player is over designed it costs more and will consume more power. Generally all portable devices must be as power efficient as possible, thereby making it possible to either reducing the battery size or lengthening the play time. A reduction in the battery size will further reduce the weight and size of the device.

Prototyping is a way of creating the next portable MP3 player, but this is an expensive process, where many prototypes might be built before the optimal device is created. It is further a time consuming process since it may take some time before a working prototype is built.

Simulating a device is therefore a very efficient way of estimating the resource requirements needed for a specific device. This is exactly the purpose of the simulator that has been developed during this thesis. The simulator is able to simulate a given set-up, with multiple tasks and multiple processing elements. The simulator can further simulate the processing elements using four different scheduling algorithms, being *Fixed-Priority*, *Earliest Deadline First*, *Constant Bandwidth Server* and the *Linux O(1)* scheduler.

1.3 Related work

Much work has already been done relating to defining QoS for multimedia. The main area of interest has been in identifying the relevant metrics and how the QoS formally can be expressed. The paper [JN04] by Jin and Nahrstedt looks into various QoS classification languages and their properties. They propose a three layer model, which also is discussed at the end of section 2.2 in this thesis.

A further research area related to QoS for distributed multimedia is creating a framework which can be used for guarantying some kind of QoS. The paper [ACH98] by Aurrecoechea et al. presents a survey of some of these frameworks, that have been developed in order to ensure a certain QoS for multimedia applications.

The Phd. thesis [AM05] *Modeling Multimedia Workloads for Embedded System Design* by Alexander Maksyagin looks into the variability of multimedia, and how this variability can be modelled using Variability Characteristics Curves (VCC). These curves are created for various metrics related to a multimedia stream, which for example are production curves, execution curves and arrival curves. The production and arrival curves relate to the token reception and dispatch at the task level, while the execution curves correspond to the execution time of a task. The curves basically represent best case timing and worst case timing, and are used in real-time calculus to analyse the workload of the multimedia.

Simon Perathoner developed a System level simulator Application Programming Interface (API) named *PErformance SIMulation of Distributed Embedded Systems* (Pesimdes). This simulator API is presented in his master thesis [SP06]. Pesimdes is used for constructing a model of a multi processor embedded system at the system level, using the modelling library SystemC. Pesimdes is however designed for modelling hard real-time systems, and not soft real-time systems. Pesimdes and other simulators are further introduced in section 3.3.

1.4 My work throughout the thesis

This section presents some of the areas I have been working on during this thesis.

DiMAS The Distributed Multimedia Application Simulator is capable of simulating various combinations of multimedia applications. The focus of this thesis is on the DiMAS. Latter sections will present more details on the *design*, *implementation* of this simulator.

Modified SimpleScalar The debugger which is part of the SimpleScalar tool set was modified in order to make traces of execution times for various multimedia applications. The debugger will halt at some user specified breakpoints, and will at that time also write the breakpoint identifier and the execution time counter to a user specified file. More details of the modifications is given in chapter 6.

BreakDiff A utility which reads a trace file generated by the modified SimpleScalar simulator and takes a certain amount of arguments, depending on the number of breakpoints the user wishes the extract from the trace file. Using the arguments the user can specify sets of breakpoints and chain them to a block, the execution times of these blocks are then written to separate files. More details on this utility are given in chapter 6 and the appendix A.1.

MPEG stat This application is a slight modification of the free MPEG decoder software which can be downloaded from the MPEG organization [MPEGOrg].

The purpose of this application is to extract the most relevant attributes of MPEG videos, which are used for the simulator. The attributes are all stored in the various headers of the MPEG video files, so the utility simply extracts the information and prints it to a file. The application is able to extract various types of details, ranging from a simple summary of the most common attributes to more detailed traces. The details of this application are elaborated in appendix A.2.

VCDParser A small utility which parses a VCD (Value Change Dump) file and exports the data into a more readable format, which easily can be plotted using *gplot*. Appendix A.3 gives a description of this utility.

1.5 Structure of the thesis

Chapter 3 starts off by introducing various classes of simulators and different types of abstraction levels a simulator can work at. The last part of the chapter introduces some existing simulators.

Multimedia is the main topic in chapter 2, where the MPEG and MP3 coding and decoding algorithms are introduced. This chapter further looks into the variability in multimedia and how this can be modelled. Four scheduling techniques are introduced in the last part of the chapter, where there also is a discussion on whether or not they are suited for scheduling multimedia applications.

Chapter 4 contains the design of the DiMAS API. The chapter starts with identifying what needs to be simulated. The design of the Pesimdes simulation API, which DiMAS is based on, is further mentioned.

The implementation is contained in chapter 5, where the main focus is on the four different scheduling techniques.

Chapter 6 describes how the variability in multimedia files is captured and stored in trace files.

Chapter 7 contains the case studies, where 8 different case studies are identified and analysed using the DiMAS API.

Chapter 8 concludes the thesis and discusses future work.

Chapter 2

Multimedia

This chapter discusses some different types of multimedia, and especially looks into the variability in execution time for multimedia applications, which was mentioned in the introductory part of this thesis.

2.1 Types of Multimedia

For this thesis the focus has been on the MPEG and MP3 codecs, which respectively are used for video and audio encoding/decoding. The following subsections will introduce the MPEG and the MP3 codecs in more details.

2.1.1 MPEG

The Moving Picture Experts Group (MPEG) is a standard for multimedia files containing both audio and video. There are multiple versions of the MPEG standard, where one is widely in use today, namely MPEG-2. The MPEG-4 is a further extension, which makes it possible to have 3D videos, but is not as widely in use as MPEG-2. The MPEG-2 standard is a extension of MPEG-1 where the resolution of the picture is higher and the audio experience is extended from stereo to 6 channels surround sound. The MPEG-2 standard is used in digital television broadcasting and in DVDs. For the remainder of this thesis MPEG will be used interchangeably with MPEG-2.

Typically the audio in a MPEG-1 file is MP3 format, while it for a MPEG-2 is the Advanced Audio Coding (AAC) format, that supports up to 6 audio channels. Therefore when playing a MPEG file the player needs to demultiplex the MPEG file into a video and an audio part, where each part is decoded using its respective decoding software. During playback the audio and video is then regularly synchronized to ensure correct playback experience.

A MPEG video basically consist of a sequence of images or frames, which again are split into smaller parts called macroblock, that each are 16x16 pixels in size. The

pictures are typically grouped into a Group of Pictures (GOP), which are grouped together due to similarities, this could for example be that they are all part of a scene in a film.

The macroblocks in a MPEG video are composed by luminance (Y) and chrominance (U,V) instead of RGB[1] in a standard colour television. Luminance defines the brightness, while the two chrominance components define the colour part of a macroblock. The human eye is mostly sensitive to luminance, while less to the chrominance, therefore the chrominance is often subsampled, thereby reducing the bit rate of the video [TN95]. A three digit notation specifies how the subsampling of the chrominance is relative to the luminance. Examples of this form of notation are:

4:4:4 No subsampling is performed, each pixel has both a Y, U and V value.

4:2:2 Chrominance is horizontally subsampled by a factor of two relative to the luminance.

4:2:0 Chrominance is both horizontally and vertically subsampled by a factor of two, relative to the luminance.

The two latter are typically used in the MPEG encoding/decoding. The top format does not perform any reduction of the bit rate, while the lowest performs the highest reduction of the three.

Figure 2.1 shows how the MPEG video stream is organized in the various layers according to the MPEG standard. Where the sequence layer contains all the pictures, and the slice layer contains a set of slices where each slice corresponds to a set of horizontal macroblocks. The macroblock layer contains the 8x8 pixel blocks for luminance and chrominance. The MPEG video stream in the figure uses the 4:2:0 subsampling. Figure 2.2 shows how the picture, slice, macroblock and block layers are related.

The MPEG standard uses complex and computational demanding algorithms for compressing the raw video, where the compression ratio typically is in the order of 10. An example is that a video with a resolution of 720x576 and 25 fps has a raw bit rate of 166 Mbps, while the compressed MPEG will be approximate 15 Mbps [TN95]. Obviously the compression and decompression will result in a minor loss of picture quality, but this is often negligible.

The reduction of the bit rate can be obtained by taking advantage of the redundancy often found in video files. There is basically two types of redundancy:

Psycho visual redundancy Details near object edges or around shot changes are less visible for the human eye, this can therefore be exploited in the reduction of the bit rate [TN95].

Spatial and temporal redundancy Most often the value of neighbouring pixels in a frame are closely related, both within the same picture as well as across frames [TN95].

[1] Red Green Blue

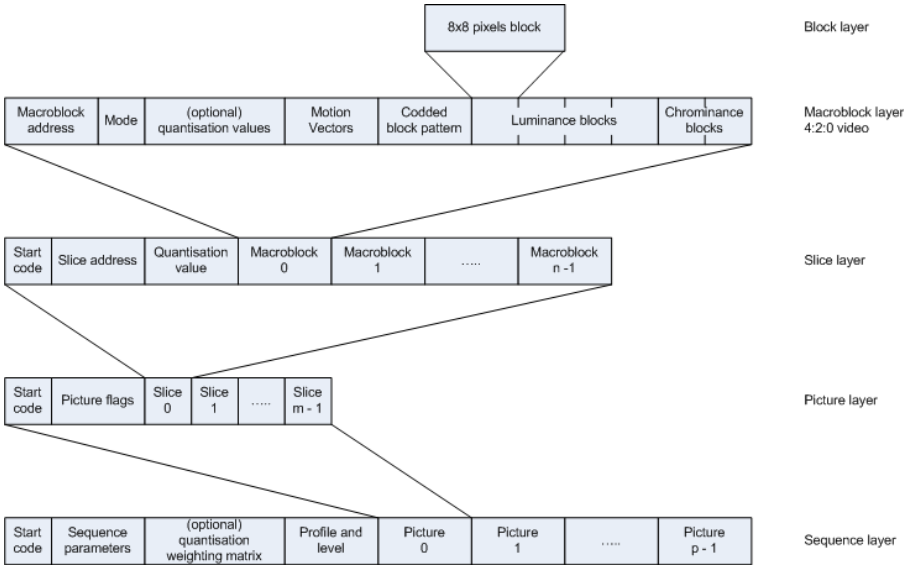

Figure 2.1: *The various layers in the MPEG video stream and how they are organized*

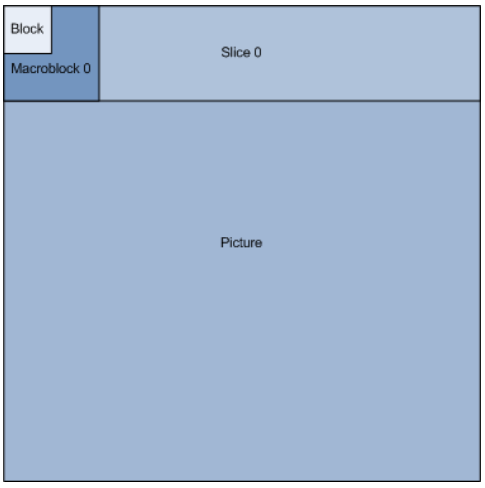

Figure 2.2: *Relation between a picture, slice, macroblock and block*

In the MPEG coding algorithm the two techniques used for utilizing the redundancies described above are inter-frame 2-dimensional Discrete Cosine Transformation (DCT) and Motion compensation Prediction (MP).

The coding process of a raw picture in a video starts off by a 2-dimensional DCT, which produces a data set that is larger than the original image, since each pixel now needs 11 bit, opposed to 8 bits at the raw image. Quantization is then used to reduce the amount of bits for each pixel. The quantization process degrades the image quality, and the degradation cannot be reversed by the decoder. Next the quantized values are coded using a variable length coding (VLC) process, where the image block is scanned in a diagonal zigzag pattern. The resulting VLC denote a run of zeros and a non-zero coefficient. The VLC are predetermined values found by looking up the value, and are based on the probability of certain values to appear. The last part of the MPEG coding process is motion compensation prediction, which utilizes the temporal redundancy. The MP needs to be performed on the decoded picture and not on the original picture, since the compression process will result in some loss of details, which the decoder will not have available. Therefore the coder needs also to have an local decoder in order to make the MP. The MP is performed by scanning the neighbouring pictures for similar blocks, and then generating motion vectors [TN95]. The MP process is what makes the encoding process much more resource demanding than the decoding process since this is a exhaustive search for similarities. Figure 2.3 shows how the various tasks of encoding from raw image to a MPEG stream are connected.

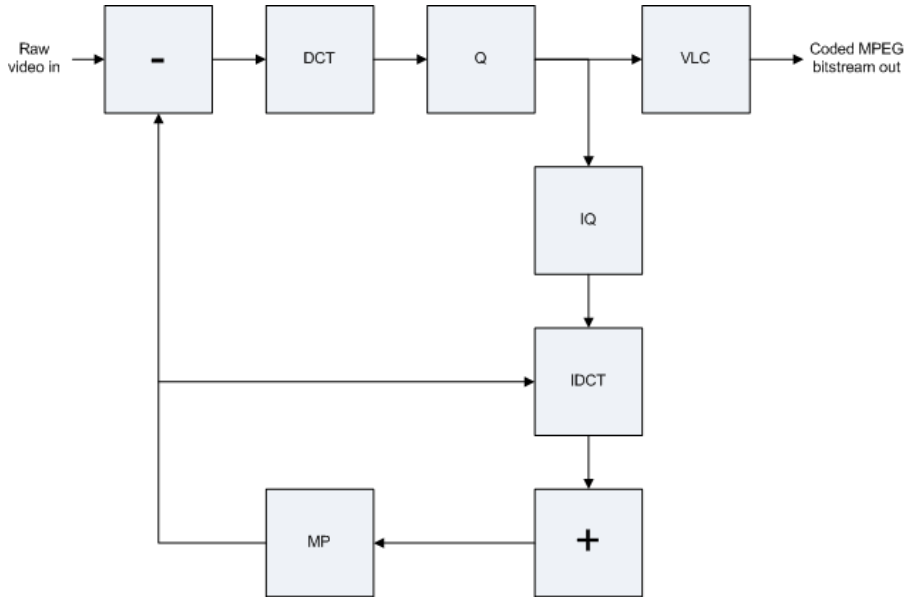

Figure 2.3: *Stages in the basic MPEG encoder*

The decoding process of a MPEG frame to a raw image is first doing a Variable Length Decoding (VLD) followed by an Inverse Quantization (IQ), an Inverse Discrete Cosine Transformation (IDCT) and then a Motion compensation Prediction. Just as the VLC the VLD is simply a lookup in a table for finding the correct values corresponding to the code. Figure 2.4 shows how the various tasks of decoding between from MPEG frame to raw image is performed.

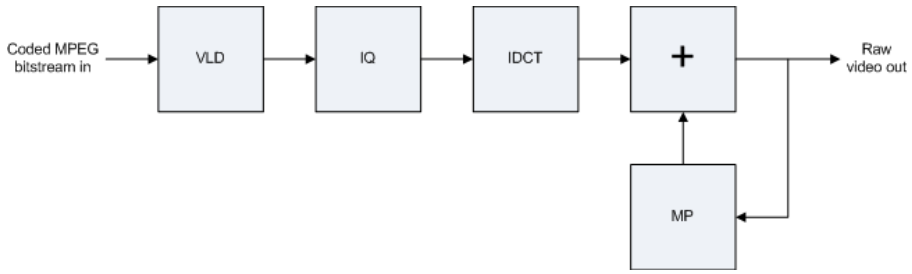

Figure 2.4: *Stages in the basic MPEG decoder*

The MPEG encoding and decoding processes described above are only the basic part of the MPEG standard. There are also more elaborate coding techniques which are used for enhancing the quality of the stream. These are however outside the scope of this thesis, and the reader is referred to the official MPEG website for further reading [MPEGOrg]. Both the MPEG encoder and decoder are available as a simple C implementation and can be downloaded from the MPEG homepage [MPEGOrg]. It is this implementation that has been used for generating the required traces for the DiMAS simulator.

2.1.2 MP3

The audio part of the MPEG-1 standard defines three layers of audio coding/decoding techniques, being MPEG layer I, II and III. The techniques rise in complexity from layer I to layer III, and the required bit rate for streaming audio is reduced. The MPEG layer III is typically abbreviated MP3 [RR02].

The MP3 compression algorithm utilizes a number of tricks making it possible to have compression ratios of up to 1:12 between MP3 bit stream and Pulse Code Modulation (PCM). The human ear is not able register sounds below 20 Hz and above 20 kHz, the top level is even reduced with the age, and most people are not able to register sounds above 16 kHz [BP00]. Therefore the audio sampling needs only be performed within this interval. Further the human ear has 24 frequency bands, therefore masking can occur when tones within the same band are played. This can be used to reduce the bit rate, since there is no need to play the tones which are non perceivable by the human ear.

The PCM signal is used for storing audio, and has two parameters, being sampling rate and bit rate. The sampling rate is given in Hz and and the bit rate is given as bit per second similar to the bit rate in MPEG videos. The size of the sampling rate determines the range of the frequencies, increasing the sampling rate will therefore present more tones. The bit rate determines the resolution of the sound, increasing the bit rate will reduce the noise introduced by the MP3 coding algorithm.

The MP3 stream structure is shown in figure 2.5. A MP3 file consists of a series of *frames*, which each correspond to 26 ms of audio, where each frame contains 1152 samples[2]. A frame is divided into two *granules* that each contain half of the samples

[2]Only the MPEG-1 layer 3, the layer 1 and 2 have less samples

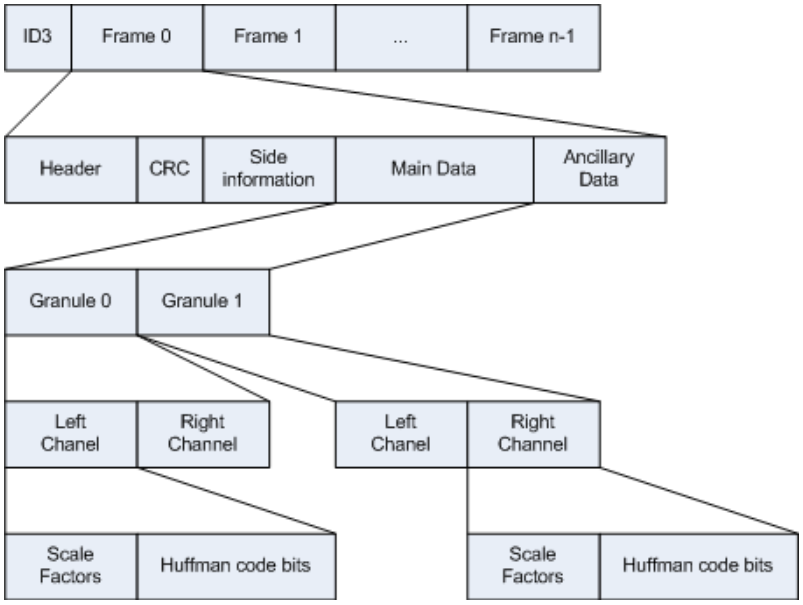

Figure 2.5: *Stream structure of a MP3 file*

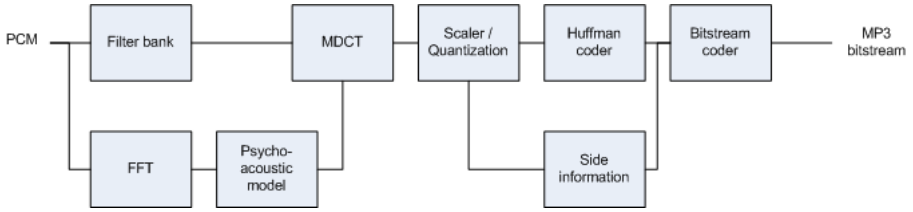

Figure 2.6: *Encoding process of a MP3 file*

in a frame. For stereo audio each granule contain samples for a left and a right channel. The last level in the MP3 stream contains the Huffman coded bits and the scale factors. The Huffman coded bits contain the actual data, while the scale factors are used for reducing the quantization noise.

The MP3 encoding process is shown in figure 2.6. The filter bank samples 1152 PCM signal and filters them into 32 frequency bands, that are equally spaced. In parallel a Fast Fourier Transform module converts the PCM signals from the time domain to the frequency domain. The psycho-acoustic module analyses the signal from the FFT module. Based on the human audio perception, as described earlier the psycho-acoustic module selects a *window type* the MDCT should use for processing the incoming signals. The window type is based on the difference between the succeeding and the proceeding frames. The quantization and scaler module is an iterative module where the sampled values are quantized and scaler values are used for minimizing the noise induced by the quantization. Finally the values are Huffman encoded and the bit stream is created with side information that the decoder uses to identify the stream parameters.

Figure 2.7 shows the MP3 decoding process from MP3 bit stream to a PCM stereo signal. The decoder starts off by parsing the bit stream into header information

and the actual data. The data is then decoded using the Huffman algorithm. Next the data is rescaled using the scaling information parsed in the bit stream, and an inverse quantization is performed. The data is split into two channels if the incoming audio data is a stereo signal. Afterwards the IMDCT and filter bank conversion is performed on each of the signals, which then results in the decoded PCM signals.

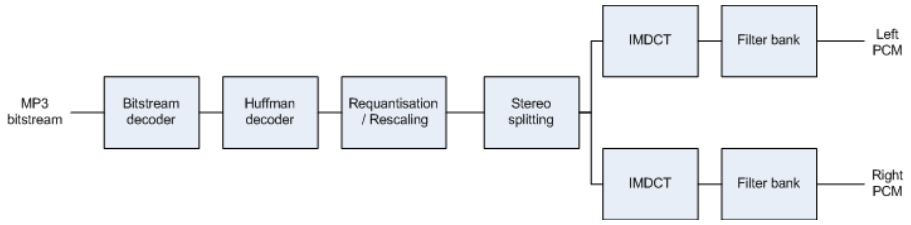

Figure 2.7: *Decoding process of a MP3 file*

The reader is referred to [RR02] for a more elaborate explanation of the MP3 encoding and decoding techniques.

2.2 Metrics

This section will present the general metrics that define multimedia files. These metrics can be used for estimating how the quality of service for a given multimedia application will be. There is much literature already addressing this subject, of the most interesting are [ACH98, JN04].

The following presents the metrics that were found most relevant for describing QoS for distributed multimedia applications. The network related parameters do not only apply to distributed multimedia applications, but also to locally run applications, they are however often negligible since local playback is many times faster than playback over a network.

Frame rate is an essential parameter for video. The frame rate needs to be sufficiently high in order for the video to be smooth. The frame rate is a rather subjective value, which can vary among people. The frame rate is first of all established, when the encoding is performed. But some of the later parameters can also influence the rate while playing the video due to loss of frames. There exists standards where the frame rate is defined. The PAL format used in Europe is 25 fps, while the NTSC format used in USA is 30 fps. Frames are also defined for MP3 streams, here the frame rate is a static and each frame contains 26 ms of audio [RR02].

Resolution is again a very important factor for video quality, the higher the resolution the better quality of the video is. This is typically a static value and will not change during playback. There could however be software that changes the resolution, based on the performance of the network connection. Often the user will before hand choose the resolution of the video being streamed.

Sampling rate is defined for audio, and determines the span of the frequencies. More tones are available as the sampling rate is increased. The sampling rate is defined when the audio is encoded, and is typically set to 44.1 kHz, since this is what is used for CD audio [RR02].

Bit rate is defined for both video and audio. The bit rate is based on the frame rate, resolution and how effective the compression algorithm is. Further a video can both be encoded using Variable Bit Rate (VBR) or Constant Bit Rate (CBR). For distributed multimedia applications it is imperative that the bandwidth of the network is larger than the bit rate of the file in order to ensure that frames are not lost.

Delay or response time can be interesting for distributed multimedia applications. It is however more for real-time communication like Voice of Internet Protocol (VoIP) and other phone services like Skype, Ventrilo and Teamspeak, where the delay must be minimal. It is crucial that the delay is low since the delay from one person starts talking to the other people hears him talking must be low in order not to talk all at once. If the delay is more or less constant this problem can be overcome for ordinary multimedia applications, like video or audio playback, by using a sufficiently large buffer.

Response time delay is similar to the *delay* mentioned above, but only includes the time a stream token has spent either during transmission and the time it has spent waiting in buffers. This is one of the metrics which are used for evaluating the performance in the case studies later in this thesis. This metric is useful due to the inter variability in multimedia, such that the performance of two streams can be compared, regardless of the execution time needed for processing each of them.

Jitter is like delay a timing related parameter, which must be low for both VoIP and more ordinary distributed multimedia applications. If the jitter is too high the quality of the sound or video may be greatly degraded due to discontinuities in the stream. And in worst case the application may even terminate the connection if the jitter is too high. Again this can be overcome by using sufficiently large buffers for ordinary multimedia applications, but it might not be a trivial task to find this size.

Loss rate is also a central parameter that should be minimized since it greatly degrades the quality of the stream, when packets are lost, thereby losing some of the audio or video experience. The loss of packets is most often caused by packets not meeting the deadlines, due to congestion on the network. Loss rate can also be seen as how many frames in a video is lost during playback.

Synchronization is also relevant for multimedia applications. For combined video and audio applications a common synchronization is where the sound and the video must be synchronized correctly, this is called lip-sync. But you could also imagine synchronization in a distributed system, where two or more speakers need to be synchronized in order the avoid an echo effect.

Availability is of course also an important parameter, since the media must be available for streaming. The media is either available or not available, so this is not a gradient value opposed to many of the other parameters defined above.

In [JN04] they propose a three layer QoS mapping, where the top layer is the *User layer*, followed by the *Application layer* and at the bottom the *Resource layer*. The user layer contains the subjective metrics, which are not directly measurable. The application layer is said to be platform independent and contains measurable metrics. The selected qualities selected from the user layer can be translated into some overall metrics in the application layer. The resource layer is platform dependent and contains metrics at the hardware level. These metrics are also as the application layer measurable. Figure 2.8 show how the above mentioned metrics are mapped into this three-layer QoS model.

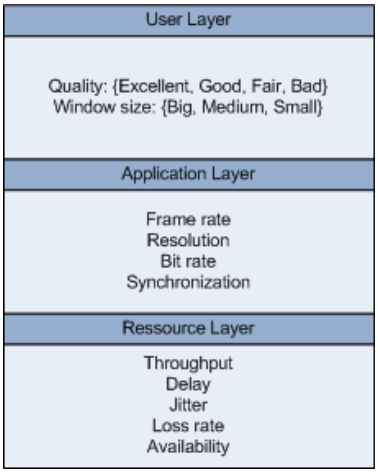

Figure 2.8: *3-layer model for QoS metrics*

2.3 Variability in multimedia

As mentioned in the introduction, multimedia applications are examples of soft real-time tasks, which have a great variability in terms of execution time. The discussion on variability for multimedia applications is based on MPEG video since this has been the main focus during this thesis.

The variability in execution time for MPEG video media sources is primarily based on five metrics, namely *frame rate*, *resolution*, *bit rate*, *frame type* and the *amount of motion*. The frame rate and resolution metrics are the main contributors, and influences the base of execution times, while bit rate, frame type and the amount of motion contribute somewhat less. In general there are two types of variability, the *intra variability* and the *inter variability*.

Intra Variability is the variability within the same multimedia source. The MPEG compression algorithm is a perfect example of this type of variability, where execution time depends on the type of frame that is being decoded and the amount of motion[3] there is at that time.

[3]For MPEG video this only applies for P and B frames, not for I frames as they do not use MP

Inter Variability is the variability across different multimedia sources. For the inter variability there are basically two cases. The first case is where the media sources have the same characteristics, in terms of frame rate, resolution and bit rate, and the second where the characteristics are different. For video media with similar characteristics the main contributors are as in the intra variability, the frame type and the amount of motion in the video. While for video media with different characteristics the main contributors are resolution and frame rate.

For many applications the most relevant types of variability are intra and the first type of inter variability. This is caused by the characteristics are often already predetermined for many appliances.

The various types of variability discussed above are shown in three figures. Figure 2.9 shows the intra variability in terms of execution time for a MPEG video with motion. The top plot of the figure represents the three different types of frames in the MPEG standard. The I-frames correspond to the value 1, P-frames correspond to 2 and the B-frames correspond to 3. It is clear from the bottom plot of figure 2.9 that there is a variability in the number of instructions required for decoding a MPEG video. Further it is clear that the type of frame has a significant impact on the variability from a intra variability point of view.

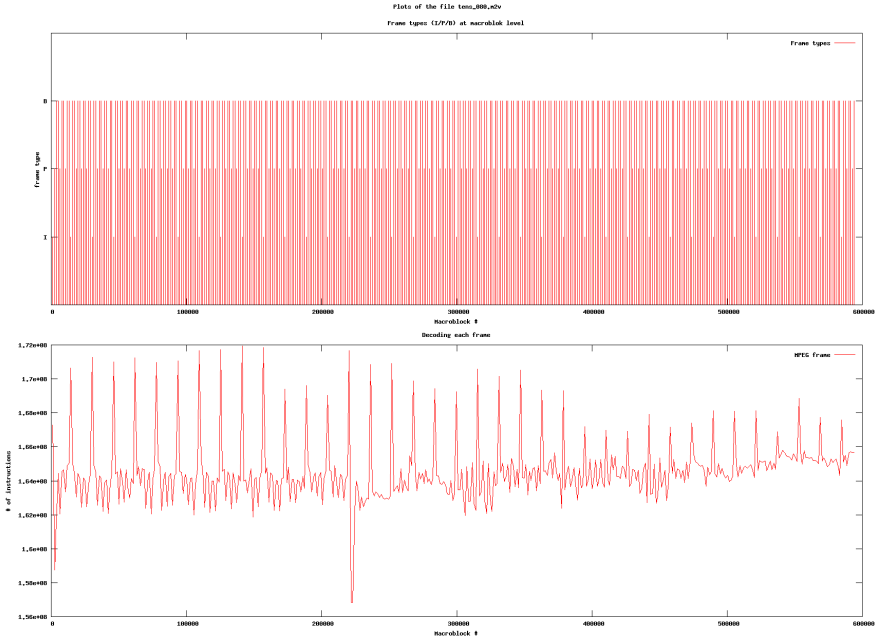

Figure 2.9: *Intra variability. The top plot shows the frame type for each frame in the video, these are either I, P og B frames. The bottom plot show the instruction count needed for decoding each frame.*

Figure 2.10 shows the inter variability for similar MPEG videos in terms of frame rate, resolution and bit rate, further the sequence of frame types is the same. The figure contains instruction counts per frame decoding, for four MPEG videos, where two are with motion, one contains a static image and a digital watch that is counting up, and the last video is simply a static image. Again it is clear how the type of

frame has an impact in terms of intra variability, but the motion in the video has an even greater impact on the execution time in terms of inter variability.

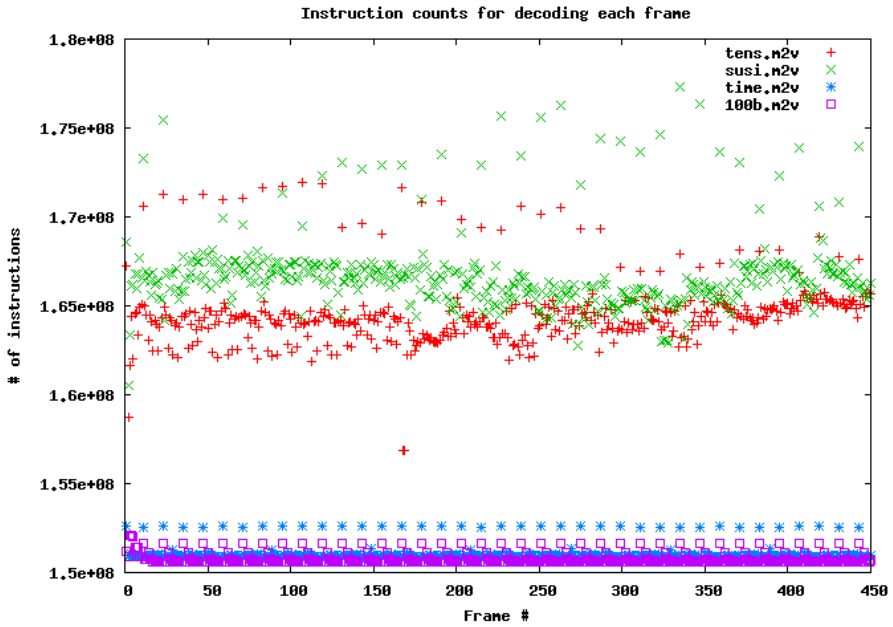

Figure 2.10: *Inter variability with similar characteristics. Number of instructions for four different MPEG video files. The first two data set (red and green) are MPEG videos with motion, while the next two (blue and purple) are MPEG videos with no and minimal motion respectively.*

Figure 2.11 shows the inter variability for MPEG videos with different characteristics. The characteristics that vary are bit rate, resolution and frame rate. The videos are clearly separated into three groups in terms of variability. The variability is mainly caused by the resolution, and next the frame rate. The bit rate is the cause of the variability within each of the three groups, and is basically and indicator of how noisy the frames are.

2.3.1 Modelling the variability

We need a way of modelling the variability of multimedia discussed in the previous section. The variability can be modelled using traces, these traces can either be simple ASCII files containing a *list of execution times* for a task or a *probability distribution of execution times* for each task.

Retrieving the execution times from a list, is a very detailed way of simulating. The simulation will be performed for a very specific media file, while we often would be interested in simulating for a broad range of media files. In order to use the simple trace files for simulation you need to generate them, which is a time consuming process and the resulting trace files can be huge for long media files[4]. Section 6 describes how the traces were generated in this thesis.

[4]The trace files generated for the MPEG videos, which are 15 seconds in length and have a resolution of 720x480 and 30 fps are each 30 MB in size

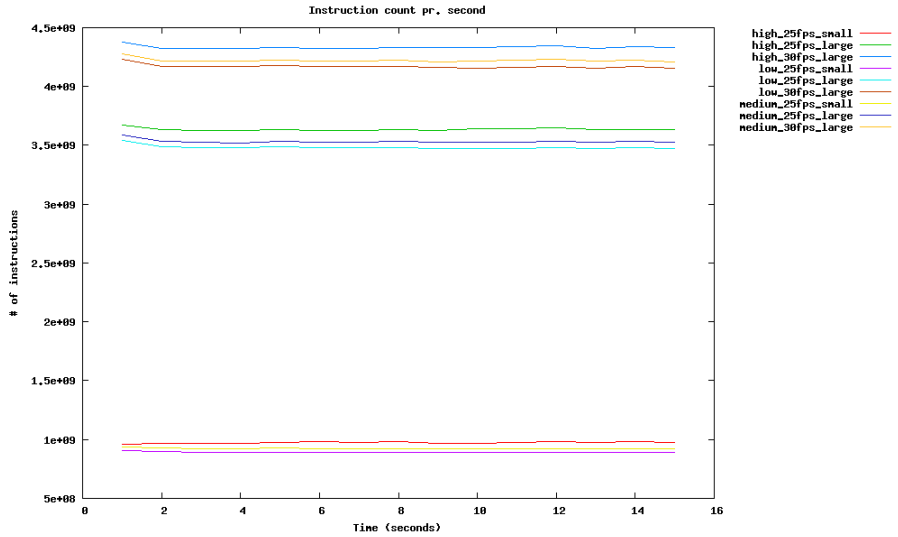

Figure 2.11: *Inter variability with different characteristics. The characteristics are bit rate (high/medium/low), fps and resolution.*

The distribution describing the variability in execution times for multimedia is a nice way of providing the necessary input data for a simulation. The distributions are of course based on measured execution times as the traces described above, but they are smaller in size and can easily be altered by hand to represent a broader range of multimedia files. During the simulation the execution time is obtained by using the distribution in conjunction with a random number generator.

The large variability in multimedia files is unavoidable, therefore special consideration must be taken, as to what the characteristics are of the target media, when running the simulations. Both forms of traces discussed above can be used during simulation, but the distribution is the most dynamic, and can give a broader range of execution times. The DiMAS API does however only provide tracing functionality using a list of values.

2.4 Distributed Multimedia

The variability discussed in the previous section was towards multimedia run both locally and in a distributed manner. There is additional variability for distributed multimedia, which is caused by the communication. The communication can both be in terms of networked applications over traditional Ethernet and in Multi Processor System-on-Chip (MPSoC) devices where the communication is performed over a bus. For distributed multimedia the variability is mainly caused by limited bandwidth, too high latency and jitter on the channel.

Ethernet is a best effort service, and it is not possible to provide any guaranties that the network packets will arrive at a certain time. So the arrival time of multimedia being streamed over a network is likely to have a very high variability, depending on the distance and the load of the channels from sender to receiver.

Modelling the variability in distributed multimedia is done in the same way as discussed in section 2.3.1. Again there are the two possible ways of using traces, either a full trace of when each packet will arrive, or using a random number generator together with a probability distribution.

2.5 Scheduling

Many multimedia systems have multiple tasks to perform. For example a set-top box needs to decode both the video and the audio of a film. At the same time it must be able to handle input from the user and display a Graphical User Interface (GUI) on the TV. In order to keep the expenses low, the set-top box will have a general purpose processor and dedicated ASICs for some of the more demanding functions. These tasks need to be performed concurrently, therefore we need a scheduler, which can schedule the various tasks onto the processor.

Some of the most common scheduling policies for embedded systems with hard real-time tasks are *Fixed-Priority (FP)* and *Earliest Deadline First (EDF)*. These scheduling policies do however require that the WCET of the tasks are known. Schedulability analysis techniques can then be used to guarantee that the timing constraints are respected. As we have already seen in the previous section the execution times vary a great deal depending on the multimedia file being processed, and using the WCET will lead to over designing the system, so that it is idle most of the time.

The following sections introduce four different scheduling schemes and looks into how they perform for multimedia applications. These schemes have all been implemented in the simulator, and will be the base of the case stories presented in section 7. FP and EDF are only lightly introduced as these are simple and well know scheduling algorithms.

2.5.1 Fixed priority scheduling

Fixed Priority scheduling is widely used in real-time operating systems. It is one of the simplest scheduling techniques, where each task is given a unique priority. The task with the highest priority is scheduled onto the processor. FP scheduling can be either preemptive or non-preemptive. The preemptive scheduling is based on the priority, so a higher priority task can always preempt a lower priority task running on the processor, and thereby gain access to the processor.

Equation 2.1 is used for making a schedulability analysis for a number of tasks being scheduled using the FP algorithm. The symbols in the equation are given as: c_i : WCET and t_i : period of task i.

$$\sum_{i}^{N} \frac{c_i}{t_i} \leq 2 \cdot (2^{\frac{1}{n}} - 1) \tag{2.1}$$

This type of scheduling is probably not very suited for multimedia where tasks are computational heavy. Further there will always be one process which has precedence over another, which can leave to starvation if the system is under designed.

2.5.2 Earliest Deadline First

The Earliest Deadline First (EDF) scheduling algorithm is a dynamic algorithm, which also is used in embedded real-time systems. The EDF is typically used as a preemptive algorithm where the priorities are based on the deadlines of the active tasks. The task with the earliest deadline is assigned the highest priority and is scheduled onto the processor. For periodic tasks the scheduling analysis is done using equation 2.2, where C_i is the WCET and t_i is the period of the task i.

$$\sum_{i}^{N} \frac{c_i}{t_i} \leq 1 \tag{2.2}$$

Multimedia applications are basically periodic tasks with jitter and scheduling this kind of tasks using EDF will probably be slightly better than for FP, since the priority of the tasks are assigned dynamically.

2.5.3 Constant bandwidth server scheduling

The Constant Bandwidth Server (CBS) is a resource reservation algorithm, which ensures that aperiodic soft real-time tasks can be scheduled together with hard real-time tasks, without jeopardizing the deadlines of the hard real-time tasks. Resource reservation servers both exist for fixed priority and dynamic priority schedulers. The CBS is used in conjunction with a dynamic priority scheduler as EDF.

Resource reservation algorithms are generally characterized by having a budget Q, and a period P, much like hard real-time tasks having an WCET C and a period T. The budget specifies how much time the resource reservation server at most can assign to aperiodic tasks, within the period P. There exist both soft and hard resource reservation techniques, [BLAC05] presents the definitions of these two types.

Definition 2.5.1. A hard reservation is an abstraction that guarantees the reserved amount of time to the served task, but allows such task to execute at most for Q_i units of time every P_i.

Definition 2.5.2. A soft reservation is a reservation guaranteeing that the task executes at least for Q_i time units every P_i, allowing it to execute more if there is some idle time available.

The CBS implements soft reservation, that uses tasks deadlines to optimize processor utilization, while providing temporal protection, so hard real-time tasks do not miss deadlines. The CBS controls the deadlines of soft real-time tasks, which the EDF scheduler then uses to find the next task to execute. A CBS server has a bandwidth $U_s = \frac{Q_s}{P_s}$ that can be used for traditional scheduling analysis using equation 2.2.

When a soft real-time task goes to a running state, then it is queued into a FIFO queue on the CBS. The tasks will be assigned a deadline if the queue is empty, or it will wait until the previous tasks have been served. The server keeps track of the server deadline d_k, which also is assigned as the deadline of the currently running soft real-time task.

The CBS algorithm is defined in the following

- The server bandwidth is defined as $U_s = \frac{Q_s}{P_s}$, which is used in scheduling analysis.
- The server has a deadline d_s, which the active CBS task is assigned.
- The server budget is decreased by the same amount of time a CBS task is running.
- When the server budget is depleted it is replenished, and the server deadline is incremented as: $d_s = d_s + P_s$. The budget will never be zero for a finite interval in time.
- The server is said to be active if there are pending jobs.
- A job is enqueued in the server queue as it arrives at time r_i. The queue is implemented as a FIFO queue.
- If the server is idle as a job arrives, then the deadline is updated if $d_s \leq r_i + (\frac{c_i}{Q_s}) * P_s$ the new deadline is $d_s = r_i + P_s$ and the budget q_s is replenished to Q_s. If not then the arriving job is served with the current server deadline and budget.
- When a job terminates then the next in queue is served using the current server deadline and budget. The server becomes idle if the queue is empty.

Listing 2.1: *CBS pseudo code*

```
1  When job t_j arrives at time r_j
2     enqueue the request in the server queue
3     n = n + 1
4     if(n == 1) // the server is idle
5        if(r_j + (q_s / Q_s) * T_s >= d_s)
6           d_s = r_j + P_s
7           q_s = Q_s
8     else // nothing else is done if the server is active
9  When job t_j terminates
10    dequeue t_j from server queue
11    n = n - 1
12    if(n != 0)
13       serve the next job with current server deadline and budget
14 When job t_j executes for a time unit
15    q = q - 1
16 When (q == 0)
17    // server bandwidth exhausted
18    d_s = d_s + P_s
19    q = Q_s
```

The purpose of the CBS is to provide temporal protection for hard real-time tasks not to miss their deadlines, while still serving soft real-time tasks as much as possible. The strength of the CBS is that it can ensure that a task is not hugging the processor thereby starving other tasks. This type of scheduling is specifically aimed at continuous media tasks, and will therefore probably be an effective scheduling algorithm.

The CBS algorithm can be used in a couple of different ways for systems where multiple soft real-time tasks need to be scheduled onto a processor:

- One CBS can schedule all soft real-time tasks

- One CBS for each soft real-time task can be used

If the first approach is used then the scheduling will more or less be done as plain FP scheduling for the soft real-time tasks, since the CBS unit provides a dynamic deadline for the first task in the queue. Only when the first task in the queue terminates, will the next task in the queue gain access to the processor.

The second approach is much more suited for soft real-time tasks, since this provides a degree of fairness, where all tasks are given processor time. The scheduling will probably look much like a time division multiple access algorithm, since the server bandwidth basically corresponds to a time quantum.

For more details on the CBS the reader is referred to the *Soft Real-Time systems* book [BLAC05].

2.5.4 Linux scheduler

Two different scheduling algorithms are used in the Linux 2.6 kernel. In this thesis we will look at the scheduling algorithm used in the kernels up to 2.6.23, since this is what the *Windriver linux*[5] kernel uses. The scheduler used in the 2.6.23 and upwards is the Completely Fair Scheduler (CFS), more information can be found at [MH07]. For the remainder of this section the 2.6 kernel will refer to kernels below 2.6.23. The scheduler for this kernel is is named O(1), since it runs in constant time regardless of the number of tasks.

The Linux kernel 2.6 works with two classes of tasks, *real-time tasks* and *normal tasks*. Every task is given a priority, in the range of 0 to 140, where 0 to 99 are assigned to real-time tasks, while 100 to 140 (the *nice range*) are assigned to normal tasks. The scheduler uses a preemptive, priority based time-sharing algorithm. The tasks are assigned time quanta according to their priority, the high priority tasks have the largest time slice [SGG09].

The scheduler works with two queues, one for tasks which are runnable and one for tasks that have spent their time quantum. Normal tasks are only considered runnable in the case that they have not already spent their time slice, whereas real-time tasks are runnable until they are done. When all the tasks in the runnable queue have been moved to the expired queue the counters are reset and execution starts over. At the reset the priorities are further dynamically incremented or decremented with a maximum value of 5, depending on how much time each task has spent sleeping. This is however not performed for the real-time tasks [SGG09]. The reason for assigning tasks with high idle time higher priorities is that they often are interactive tasks, and they should therefore have a high priority, so the user has a good experience using the system.

The real-time tasks are not influenced by the time sharing and will run to completion before any of the normal tasks will have a chance to acquire the processor. The real-time tasks are scheduled with preemptive fixed-priority, where the highest priority

[5]The Windriver linux is widely used in embedded systems.

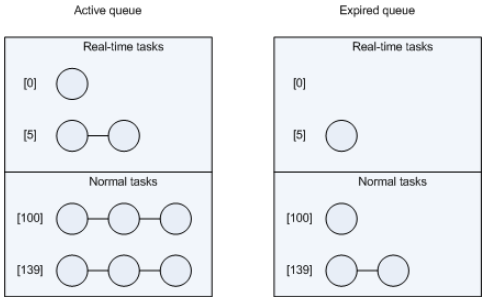

Figure 2.12: *Linux scheduling queue for active and expired tasks within the same epoch. The scheduler in this example is assigned 4 real-time tasks and 9 non real-time tasks.*

task is scheduled onto the processor. For real-time tasks with the same priority, the scheduling is performed either using a First-Come-First-Served (FCFS) or a Round-Robin (RR) algorithm. FCFS is a FIFO style algorithm, were the active task keeps the processor until it is done. RR is a time sharing algorithm, where each task is assigned a time quantum, and when it is expired, the task is put at the end for the queue and the time quantum is replenished. Real-time tasks are only moved from the active queue to the expired queue once they have completed [SGG09].

Multimedia applications are defined as soft real-time tasks and can therefore be assigned as real-time tasks in the Linux operating system and have the highest priorities. Assuming the system is not under designed, then the large variability in the multimedia tasks will make it possible for the normal tasks also getting some processor time. The multimedia tasks are however probably the most processing intensive tasks in the system, while some of the normal tasks might be shorter. With the Linux scheduling the short normal task will have to wait for the long real-time tasks to finish, before it can be scheduled.

Next there is the FCFS and RR scheduling algorithms which are used for real-time tasks that have the same priority. Both algorithms serve the task, which has been waiting the longest time in the queue. This kind of scheduling seems to be a good choice for soft real-time tasks like multimedia, since some deadline misses are acceptable.

2.5.5 Hypothesis

The Fixed-Priority and the Earliest Deadline First scheduling algorithms are both hard real-time scheduling algorithms, and are not well suited for soft real-time systems where the resources are scarce. The FP is probably the worst, since systems with limited resources can end with starvation of the lower priority tasks.

Scheduling using the Linux O(1) scheduler will be similar to the FP scheduler for multimedia tasks that are scheduled as real-time tasks. As already mentioned this can lead to very poor performance for other applications that for example provide a user interface. Therefore it is more interesting to look at how the multimedia performs when assigned as user tasks, and have a lower priority than the user interface application. This makes sense under the assumption that the user interface application is only seldom active.

The Constant Bandwidth Sever is a resource reservation service, which ensures that hard real-time tasks can operate on the same processor as soft real-time tasks, while not jeopardizing the deadlines of the hard real-time tasks. The CBS is however not a scheduling algorithm in itself, the CBS works on top of another deadline oriented scheduler like EDF. Having one or more CBSs controlling the deadlines of soft real-time tasks on a processor, where no hard real-time tasks are running, might not turn out to be much different from an ordinary EDF. It should however ensure that only the tasks which have very high variability suffer for this, while the rest are unaffected.

Chapter 3

Simulators

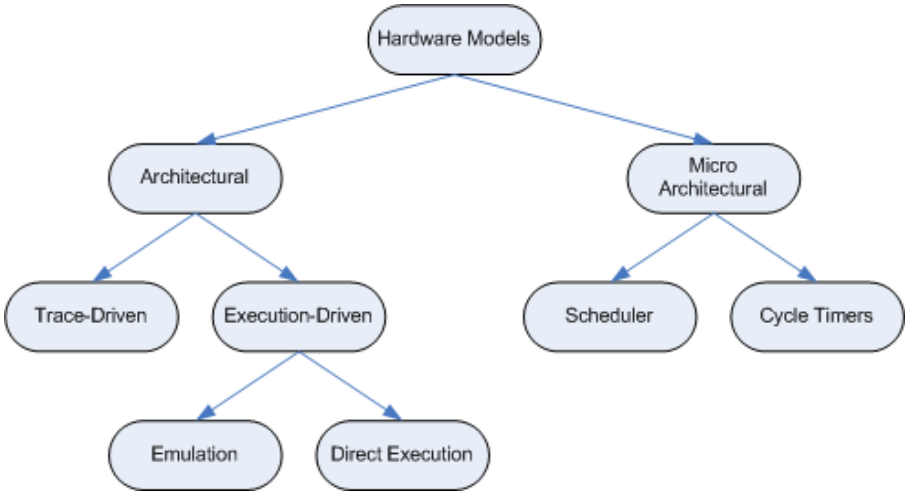

Figure 3.1: *Classification of simulators*

Generally simulators can be classified into two groups, one being architectural and another being micro-architectural, these are respectively concerned with functional and performance simulation. A micro-architectural simulator is more or less a software implementation of a microprocessor with instruction set, cache and pipeline emulation.

The micro-architectural simulators can further be classified into scheduler and cycle time simulators. The scheduler simulators are the simplest of the two, in the sense that the instructions are executed in-order according to resource availability. This type is called an Instruction Set Simulator (ISS). The cycle time simulators are cycle accurate and emulates the processor into details with cache hits and misses and pipeline stages. This type of simulator is also called a Cycle Accurate Simulator (CAS).

In addition to the ISS and CAS we also have Full System Simulators (FSS), the three types are elaborated below [MY07, ALE02]:

ISS An Instruction Set Simulator (ISS) is a simulator which emulates a processor at the micro architectural level. An application is run on the simulator where

each instruction is executed in-order and the simulator can thereby simulate the functionality of the applications. This leads to fairly accurate simulations, but performance measurements are however not that reliable.

CAS A cycle-accurate simulator is a software implementation of a hardware device, typically a processor. Here cache hit and misses, branch predictions and the flow of every instruction through the various pipeline stages is simulated. A cycle-accurate simulator basically simulates all the internal states of the micro-architecture. Therefore it is even able to simulate out-of-order instructions, where instructions can be delayed for several machine cycles for example due to floating point operations. This kind of simulator is able to make very detailed performance measurements of the processor, but it has a cost in terms of the time required for running simulations.

FSS Full System Simulators typically include an ISS or preferably a CAS, and combines it with peripheral entities like memory and disc I/O. Typically a FSS is built as a virtual machine, which makes it possible also to run the applications on top of a real operating system. Obviously a FSS is a very resource demanding simulator, and it is able to present very detailed performance metrics.

As seen in figure 3.1 the architectural simulators can also be classified into various groups and subgroups. At the first level we have trace driven opposed to the execution driven simulators. The trace driven simulators uses traces recorded from previous program execution for measuring performance of an application. This way all functionality of the application has been saved and the simulator need not worry about the functional execution or the I/O devices. Therefore similar simulations using a trace driven simulator will always end up with the same performance measurements. The execution driven simulation is more complex than the trace driven since the functionality of the application is simulated as it is executed. Further the simulator needs to supply the various I/O devices that the simulated application might need. Specially execution driven simulators using networked devices can have varying performance measurements, since the performance of the network can vary from simulation to simulation. This variation can however be dealt with using traces for the I/O devices while running the execution driven simulation [ALE02, SSTutorV4].

Finally an execution driven simulator can either be performed using emulation or direct execution. Simulators which use emulation for simulating an application typically requires some kind of ISS. While a direct execution simulator utilizes the host architecture for executing the application, this is often referred to as co-simulation. Obviously direct execution requires that the host architecture is similar to the target architecture of the application which is simulated [MY07].

3.1 Levels of Abstraction

As seen in the previous section simulators vary a great deal in levels of abstraction. The abstraction pyramid in figure 3.2 shows the connection between *accuracy*, *cost* and *abstraction* of a model. The accuracy and cost of the simulation is low at the high abstraction levels, while it is high at the low levels of abstraction. Accuracy is meant

as the level of details in the model which is simulated, meaning how accurate the model is compared to the architecture. Cost covers how much resources are needed for the simulation both in terms of the simulation time and amount of memory usage. Further the abstraction pyramid shows how the design space for very abstract models is large, whereas the design space is narrowed as the abstraction of the model is lowered [ZL02].

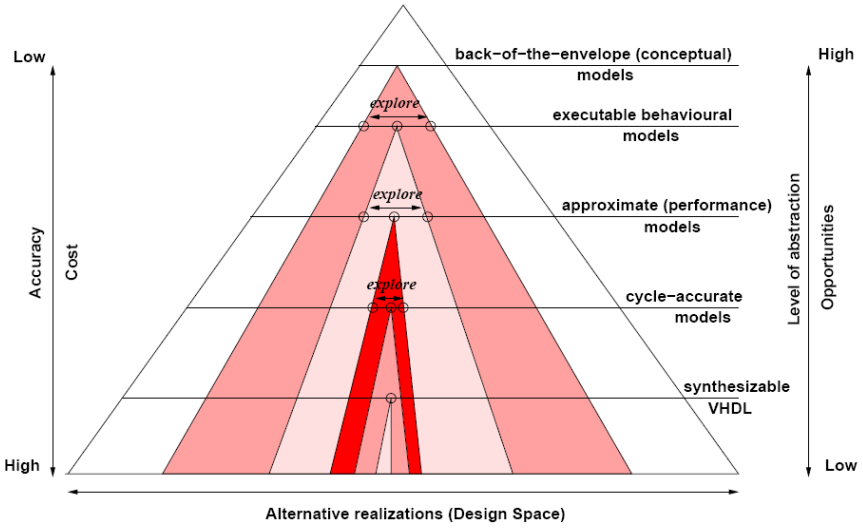

Figure 3.2: *The Abstraction Pyramid [ZL02]*

The micro-architectural simulators which were mentioned in the previous section are at the low level of abstraction and therefore reside in the area around *approximate models* and *cycle-accurate models*. The architectural simulators are at a somewhat more abstract level and reside around *executable behavioural models* and *approximate models*. As we will see in section 3.2 the architectural simulators can also reside at the top level of the abstraction pyramid namely *back-of-the-envelope*.

For prototyping it is preferable to have a model of high abstraction, since the model must be fairly simple and quick to create, further the simulation must be done within a reasonable time. Once the target architecture and the application code is written it is interesting to have a more detailed model, which can be used for extracting various forms of performance metrics like WCET, power consumption or code verification.

3.2 Design-Level Simulator

The previous section presented the various forms of simulators shown in figure 3.1 and the abstraction pyramid in figure 3.2. However there is still one missing piece namely the top level of the abstraction pyramid the *back-of-the-envelope model*. This kind of simulator is an architectural model, but does not use executable code for the target architecture during simulation. Instead the emulator can be seen as a high level description of how the architecture works and the user application as code

describing what needs to be done. This type of simulation is referred to as *Design-Level Simulation*.

As mentioned in section 3.1 the resulting design space for models of high levels of abstraction is large, this is what is needed at the design-level simulation. Since the simulations are merely conceptual to see how it would be possible to construct a given system. Later on in the design and implementation process it will be interesting to create more accurate models, in order to get more reliable performance measurements.

The DiMAS API that is developed in this thesis is used for creating design-level simulators, which are trace driven. The traces are used both for generating tokens from input devices, while other traces are used for reading the execution times for each request.

3.3 Existing Simulators

This section will present some of the existing simulators, which are able to simulate both at the architectural and the micro architectures level. These simulators are well known and are used both commercially and academically. Likewise some are commercial while others are open source projects. Many of the simulators can be configured to simulate applications at various levels of abstraction, and thus not bound to one area of the abstraction pyramid in figure 3.2.

3.3.1 SimpleScalar

SimpleScalar from SimpleScalar LLC is an open source project, which dates back to 1994 and is still under development. The simulator is highly used in the academic world and is often used in research papers for performance evaluation.

The SimpleScalar is a complete tool set with a series of different simulators which vary in the level of details. The simulators range from very simple ISS, which only emulates the instructions to simulators which take cache hit and misses, pipeline stages and even branch prediction into account. Figure 3.3 summarizes the various simulators which are part of the SimpleScalar tool chain.

Only the sim-outorder simulator is a CAS, which therefore can simulate a processor with relative high accuracy. The simulation is however very time consuming and a simulation, which takes a couple of hours using a simpler simulator like sim-profile can take a day with the sim-outorder simulator.

As part of the tool chain is a simple text driven debugger *dlite!*. Only the sim-safe simulator is not able to use the debugger. Simulation is done interactively when using the debugger, where the user can specify breakpoints for either instructions or for global variables. The simulation is then performed either by stepping over each instruction or by running until a breakpoint is hit.

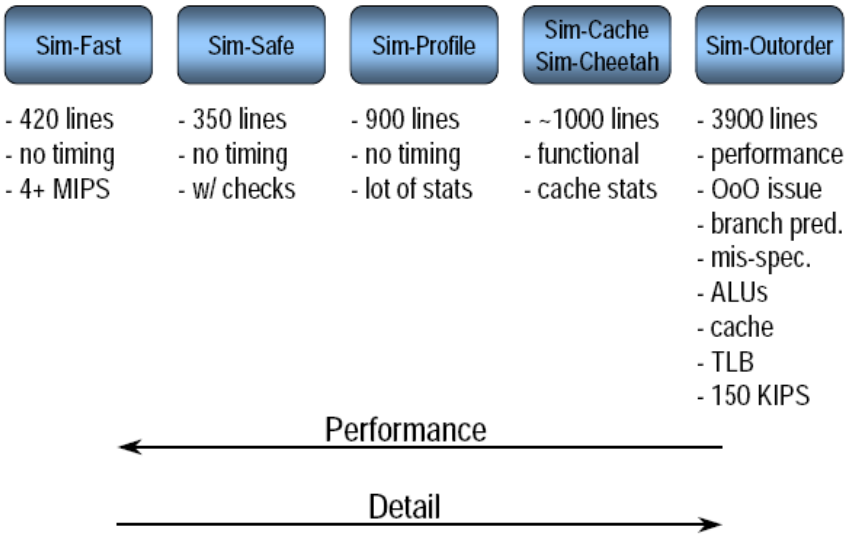

Figure 3.3: *The Various Simulators within the SimpleScalar Tool chain [SSTutorV4]*

The simulator source code is very modular and can therefore easily be customized to support specific features like tracing. During the work on this thesis the debugger was extended to implement a tracing feature based on breakpoints, section 6 elaborates how this extension is implemented.

The reader is referred to [SShome, SSTutorV4] for more details on the SimpleScalar tool chain.

3.3.2 Simics

Simics from Virtutech is a commercial simulator, which supports PowerPC, x86, ARM and MIPS architectures. The Simics simulator is a FSS, which emulates a chosen hardware device, both with the processor as well as various types of I/O devices. Basically the simulator is a virtual machine. The virtualization makes it possible to execute the same binaries in the simulator as on the physical hardware. A further advantage of the virtualization is that the simulation can be run with an operating system and multiple applications [SimicsDS].

Simics is an execution-driven simulator that even is able to emulate multi core processors and other types of System on Chip (SoC) hardware. The simulator is deterministic in the sense that the same execution can be performed multiple times with the same timing parameters, thereby making it easier to debug timing related bugs. When using breakpoints during simulation the whole system is halted, including timers and interrupts. The user is then able to extract debug information of the system, and continue the simulation without the simulated applications and operating system noticing the delay [SimicsWP].

Simics can capture detailed information regarding the target hardware architecture for making cache analysis or general performance analysis. It is also possible to perform network simulations using Simics.

3.3.3 PTLsim

PTLsim is a cycle accurate simulator, the PTLsim/X is an extension making it a full system simulator, which incorporates a virtual machine environment like Simics. The details of the simulator range from Register Transfer Level (RTL) up to full speed execution on the host architecture, which is done through co-simulation. A great advantage of PTLsim over Simics is that the simulator source code is publicly available, and can therefore be altered to suit the users needs.

The PTLsim simulator is restricted to 32/64-bit x86 Instruction Set Architectures, but with PTLsim/X the simulator can simulate 64-bit multiprocessor architectures with up to 32 processors [MY07].

3.3.4 SystemC

SystemC is an API for *C++* which features classes that can be used for modelling both software and hardware. It can therefore also be used for modelling systems as a combination of software and hardware. SystemC is capable of modelling systems at multiple abstraction levels and can even model complete systems, where various components within the same system are modelled at different levels of abstraction.

Figure 3.4 show various modelling languages and what they can model ranging from transistor level to system requirements. SystemC is both capable of modelling as low as the Register Transfer Level (RTL) like VHDL and Verilog and through functional and up to architectural level. This makes SystemC a very attractive language, and seeing that the language simply is a set of C++ classes most C++ programmers will be able to learn the language fairly easily.

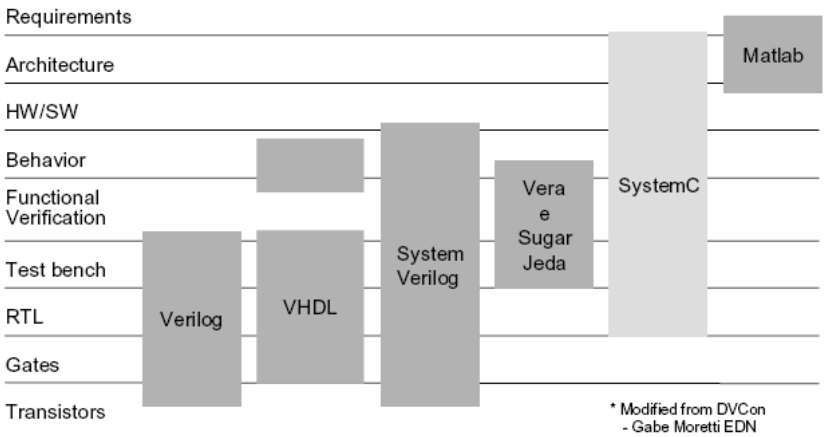

Figure 3.4: *Comparison of SystemC and other design languages [BD04]*

SystemC includes a simulation engine, which is capable of running a simulation of the model. The engine is able to simulate timing constraints if the various modules in the model are augmented. A specialized class *sc_time* is used for representing time in

the models. Logic in the modules are performed in zero time[1], unless augmentation is inserted to give a notion of time. The engine can use delta cycles[2] in the case, where modules are performing logic operations within the same time window. This can be used by the user for making sure that the simulation of the model is deterministic.

Since SystemC is written in C++ it is possible to incorporate application code written in C/C++ into the simulator and run it as part of the simulation. This makes it a very flexible tool for implementing simulators.

SystemC is however simply an API, so building a simulator requires some more elaborate programming compared to some of the other simulators presented in this section.

The basic idea of SystemC is to create various modules, the modules communicate using channels which typically are signals or FIFO buffers. Customized channels can also be created, where specific behaviour is required. This could for example be a specific communication bus, like the CAN bus. The modules can have a notion of time, which can be implemented either as a global clock signal or simply using wait statements.

There are many books about SystemC and how it can be used for creating simulators, for more elaborate description of SystemC the reader is referred to [BD04, GLMS02].

3.3.5 Pesimdes

Pesimdes is a simulation library, which is an extension of SystemC. Pesimdes is designed for creating simulators that can simulate distributed hard real-time embedded systems. The maximum backlog[3] in buffers is analysed after a simulation and used for evaluating if deadlines were missed. Further the response time[4] for task communication can be analysed. Multiple tasks can be mapped onto one processing element, and the user can choose which scheduling policy the processing element should use.

Pesimdes does not use an ISS as part of the simulation, but uses WCET, which typically is obtained using static analysis techniques with programs like aiT from AbsInt. The simulation is therefore performed at a higher level of abstraction, and no functional user code is required for the simulation.

Pesimdes is implemented in SystemC and will be the base of implementation for the simulator in this thesis. Section 4.3 will elaborate further on specific details of the simulator.

3.3.6 ARTS

ARTS is a system level simulation framework, which has been developed at the Technical University of Denmark (DTU). The framework is implemented in SystemC

[1] No time is passed in the simulation environment. This is not the actual time it takes to perform a simulation.

[2] There can be multiple delta cycles within the same simulation time window.

[3] Backlog defines how much space is occupied in a buffer

[4] Time from the token is generated until it arrives at the final destination

and is used for both modelling and simulation of Multi Processor System-on-Chip (MPSoC). The framework makes it possible to analyse different layers and their interaction in a system. This could for example be the application, middleware and architectural layer.

The framework uses a set of input files containing the application models and the platform models, which are expressed in the *ARTS scripting language*. The application model file describe the various tasks in each application and how they are connected using task graphs. The platform model file describes processing elements and network components. For simulation another file is used for making the mapping between applications and platform.

ARTS is very complex and is capable of collecting a large amount of statistics, examples of these are peak memory usage, average and maximum utilization of processing elements and deadline misses. For a more in-depth explanation of ARTS the reader is referred to [MVM07].

3.3.7 Summary

The six simulators presented in this chapter, range from cycle accurate simulators to system-level simulators, equivalent to the bottom and top of the abstraction pyramid, seen in figure 3.2. The simulators have been summarized in table 3.1, where the two main categories micro-architectural and architectural from the classification tree in figure 3.1 are included. The table has been divided into two parts, a lower part with simulators for doing performance analysis at a low level of abstraction, and the upper part for system-level simulators, which are located in the top of the abstraction pyramid.

The SystemC simulator is a special case, in the sense that it is not a pre-built simulator, but rather an API for modelling both hardware and software. But SystemC is typically used for modelling systems at the architectural level, and often as system-level simulators, therefore it has been included in the upper part of the table. But SystemC could just as well be implemented using an instruction set simulator, or even modelling a microprocessor at the register transfer level.

Name	Micro-Architectural	Architectural	Type	License
Pesimdes	No	Yes	System-level	Open Source
SystemC	(Yes)	(Yes)	All	Open Source
ARTS	No	Yes	System-level	Open Source
SimpleScalar	Yes	Yes	ISS/CAS	Open Source
Simics	Yes	Yes	ISS/CAS/FSS	Commercial
PTLsim	Yes	Yes	ISS/CAS/FSS	Open Source

Table 3.1: *Short summary of the simulators from this chapter*

Pesimdes was chosen to be the base of the DiMAS simulation API, due to the modular and simplistic design. Pesimdes can therefore easily be extended to support soft real-time tasks, instead of only supporting hard real-time tasks.

Chapter 4

Design of the Simulator

This chapter presents the design of the DiMAS simulation API. The first section presents a overview of what should be simulated. Next the requirements for the simulator are presented in section 4.2. Section 4.3 elaborates on the design of Pesimdes, while section 4.4 contains the actual design of the DiMAS API, where the main focus is on the scheduling techniques.

4.1 What needs to be simulated

It was made clear in chapter 2 that multimedia is highly variable in terms of execution times. Multimedia is a continuous media, which due to its variability is characterized as aperiodic. The purpose of the simulator is to simulate multimedia and other minor tasks for example user interface applications, which are highly aperiodic. The simulator therefore needs to simulate aperiodic tasks, which vary in execution times.

The multimedia that needs to be simulated is both video, audio and other general applications. Basically the simulator must be able to simulate both soft and non real-time tasks. It might however be valuable if hard real-time tasks also can be a part of the simulation.

A key objective for the simulator is the ability to simulate distributed multimedia applications. Therefore the simulator must be able to model a system with multiple processing elements and multiple tasks running on each processing element. Further it must be possible to model the variability on the input side for example from a network connection, where jitter and burst of data can occur.

The purpose is to create a model of a given multimedia system, and check how powerful the processing elements needs to be in order for the system to obtain a sufficient QoS. Task mappings, processor scheduling algorithms, processor speeds and buffer sizes can be customized in order to obtain the best result.

The end result is a system-level simulator, that presents a good estimate of how well the system will perform. The performance details are narrowed into only the most

vital metrics, and the input data can be qualified guesses of the resource requirements for the various tasks. This high abstraction level is desired both in order for simulation speeds to be kept low, but also due to the high variability in the multimedia. The high variability is the source of why the simulation must be done at a high level of abstraction, since lowering the abstraction level only will result in more detail performance estimates for a specific set-up with a specific set of input.

The metrics that are used for evaluating the QoS for a system model are given in the following.

Lost frames This is probably the most obvious metric of them all. Frame loss both for video and audio is a central part of the QoS metrics. It is crucial that the loss of frames is kept at a minimum, in order to deliver a good user experience. Recall that this metric is at the user layer in the three-layer QoS model presented in section 2.2.

Response time is a measure of how much time the token has spent waiting from it was generated until it arrived at the output. The response time is a very good measure for comparing the schedulers on similar set-ups, where parameters have been customized.

Buffer backlog is the third metric, and it gives an idea of whether there are any serious bottlenecks in the model. If this is the case, then the parameters need to be changed in order to have a satisfactory system.

4.2 Requirements

A fully functional simulator requires a number of different modules, namely input devices, processing elements, tasks and output devices. All but the processing element modules need a notion of time in order to function during a simulation. This notion is given as input data and summarized in the following.

Input rate determines how often a token should be generated in the test bench. For MPEG video files, a token corresponds to a macroblock. The input rate is related to the input devices for a simulation test bench.

Consumption rate is similar to the input rate, but this is however the rate at which the tokens should be consumed at the output end of the test bench. This rate can however in some situation be unnecessary if there are not constraints as to the consumption.

Execution times is a central parameter for this simulator, since the focus is on scheduling techniques and how they perform. The execution time can both be specified in time units or in cycle counts, the latter requires a processor speed for the processing element.

The input rate can both be strictly periodic, periodic with jitter, or aperiodic using traces from a text file. It is assumed that the consumption is constant both in terms

of period and amount. Further if there is no interest in comparing the performance of the system relative to a specific consumption, then this part can be disregarded.

The various tasks that are running in the system must be assigned an execution time. This can either be a static execution time as in hard real-time systems, where the WCET typically is used, or it might be some kind of trace. When running a simulation care must be taken as to the timing specified in all three types of input in order to avoid either buffer over- or underflow.

Processing elements are in this simulator referred to as resources. All resources in a simulator test bench are associated with a scheduling technique of some kind. The scheduling techniques that are part of the DiMAS framework have already been introduced section 2.5, being *Fixed-Priority*, *Earliest Deadline First*, *Linux O(1)* and *Constant Bandwidth Scheduler*.

The output files which the simulator can generate are buffer backlog for specific buffers in the system, response time for the various input tokens and logging of lost frames. Performance analysis of the simulated set-up can then be based on these three output, where graphs can be generated, but the minimum, average and maximum values are also useful for a quick overview of the performance.

The DiMAS simulator is basically an extension to the Pesimdes simulation API, and is like wise an API. Therefore a simulation requires a test bench, which is created using the various modules supplied in the API, and a C++ compiler is used in order to create the final simulator.

4.3 PESIMDES

PErformace SIMulator for Distributed Embedded Systems is a simulator written by Simon Perathoner in 2006 at the Swiss Federal Institute of Technology. The simulator is able to simulate hard real-time distributed embedded systems, and can be used for analysing the maximum backlog in buffers and the response times for tokens sent through the system. The Pesimdes simulator is basically an API, and the user uses various components to construct a test bench. The API is based on the SystemC library, which was introduced in section 3.3.4. Therefore the simulator requires that the SystemC library has been compiled successfully. The SystemC library is then linked into the users simulator at compile time. The basic components of Pesimdes are described below.

Input stream generators The purpose of these modules are to generate tokens, which simulate communication between two tasks. Pesimdes include four types of input generators, these are periodic, periodic with jitter, periodic burst with jitter, and using a trace file. All generators generate a token with a time stamp.

Resources These are the actual processing elements. The resources have embedded a scheduler, which either can be, Fixed-Priority (FP), Earliest Deadline First (EDF), or Time Division Multiple Access (TDMA), which are typical hard real-time scheduling algorithms. The scheduling algorithms are both implemented as preemptive and non-preemptive.

Tasks can be run on a resource, and are mapped onto this resource at compile time. A task is only assigned to one resource. The tasks can have multiple input dependencies and can send multiple tokens each time they are executed.

Buffers An extension to the standard *sc_fifo* channel has been implemented in order to register the maximum backlog that was registered in the buffer.

Display A class which consumes the tokens and is used for calculating the response times of the tokens, that is the time from the token was generated until it was consumed by the *output_display* object.

A test bench is constructed in a C++ file, where the Pesimdes header file is included. A tasks object needs be be created for each task in the system. Further the various resources are initialized, where the number of tasks each resource needs to serve is part of the parameters. WCET, period, deadline and priority of a task is assigned when the tasks is mapped onto a resource. The parameters depend on which type of scheduling algorithm a resource use, for example the FP needs WCET, period and priority, while the EDF needs WCET and dead line. The communication between tasks are performed using FIFO buffers, which have been extended to create statistical information as mentioned earlier.

Pesimdes is an API for creating system-level simulators, where no functional code is needed. The resulting simulators can be used for analysing the memory and timing requirements of a distributed system. The tokens that the tasks in the system communicate with use a data structure for storing the generation time of a token, and the last time a token was processed. These data are used for the timing analysis and for the EDF scheduler.

For further details on the Pesimdes API the reader is referred to the master thesis [SP06] by Simon Perathoner.

The previous sections have already discussed what the purpose of the DiMAS simulator is, and specifically what the requirements are, the next section will clarify what additional functionality is added and outlines how this is designed.

4.4 Structure of the simulator

The general modules, which are part of the DiMAS simulator API are show in figure 4.1. The modules which are encapsulated in a green box are modules from the Pesimdes API, which have been altered in the DiMAS API. The modules, which are encapsulated in a red box are new modules, which were implemented for the DiMAS API.

The Resources in figure 4.1 correspond to processing elements with different scheduling techniques. All resources can be assigned a finite number of tasks which they need to schedule according to their scheduling algorithm. The whole resource group has also been encapsulated in a red box, since all of the resources inherit from a

new abstract class, which has tracing functionality. The design of each resource is elaborated further in section 4.4.1.

The input stream devices include periodic, periodic with jitter, periodic with burst and input from a trace file. These are the same input types that are used in the pesimdes simulation API and no additional input devices are required for the DiMAS API.

The output devices are generally devices which consume tokens, and record statistical information from a simulation, which can be logged to output files. The *Display* and *Display and write* modules calculate the response time information for each token, further the *display and write* module writes the tokens to a buffer for further processing. The *Consumer* module is a module, which consumes a given amount of tokens at a given rate. The module logs an error if the input buffer did not contain sufficient tokens, and then flushes the buffer.

The various group contains modules which have no other related modules, but are just as important. The Pesimdes simulation API introduced an extension to the standard FIFO buffers in SystemC. These buffers record the maximum backlog the buffer has experienced. These buffers are further extended in the DiMAS API with the ability to trace the backlog into a trace file. A crucial part of the API is the task module, which corresponds to a single task. The task module can have multiple inputs and outputs for communicating. The Time status module is module that can be used for printing the simulation time progress to standard output during simulation. This is useful when running long simulations, where no other output is presented. The last module in the various group is the Constant Bandwidth Server which is a module that can be used in unison with the EDF scheduler to schedule both hard and soft real-time tasks on the same processing element. The CBS has not been included in the resources group since it is not a scheduler in it self. The CBS module is designed such that it also can be used with other deadline oriented scheduling techniques.

The simulators which are created using the DiMAS API are very modular, and as seen in the previous consist of only a few different types of modules. The general idea is that each task is mapped onto a single resource, and buffers are used in between tasks to model the communication flow. Figure 4.2 shows the general idea of how a test bench can be constructed with multiple tasks mapped onto a resource, and where buffers are put in between the communicating tasks. Further one of the tasks in the test bench requires two inputs before this task is ready for execution.

The communication between input, task and output devices is done using the same token structure as in Pesimdes, this token contains a sequence number, ID for which input generated it, time of generation and the time of when it last was processed. Specifically the generation time, and last processed time in the tokens are used by the various output devices to generate the statistical information.

4.4.1 Scheduling

This section continues the discussion in section 2.5, and describes how the scheduling algorithms are designed. The resources are similarly structured, but with some variations which are required for their specific functionality. The following describes the

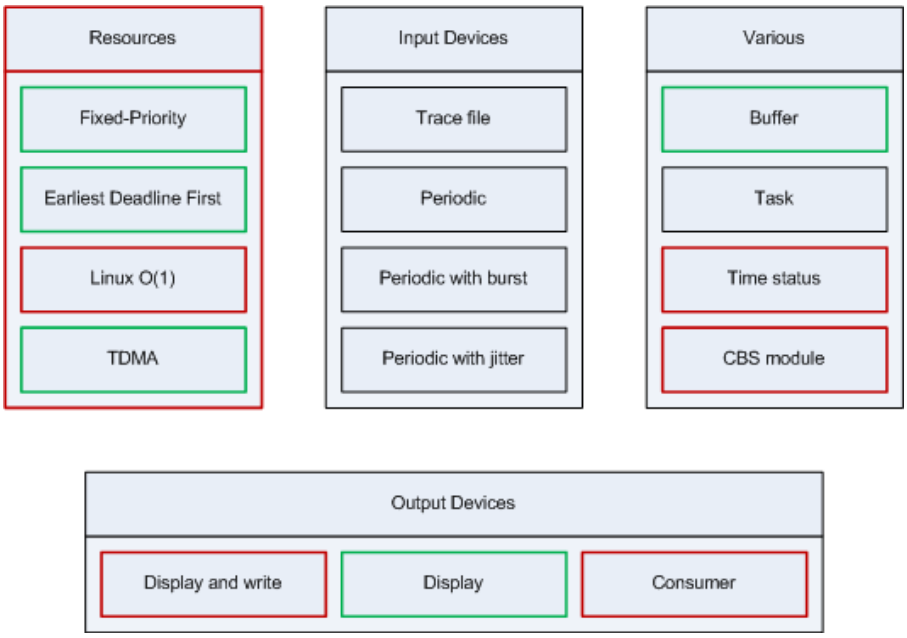

Figure 4.1: *The general modules which are part of the DiMAS API. The modules which are encapsulated in a green box are classes from the Pesimdes library, which have been altered in the DiMAS library. The modules that are encapsulated in a red box are new classes in the DiMAS library.*

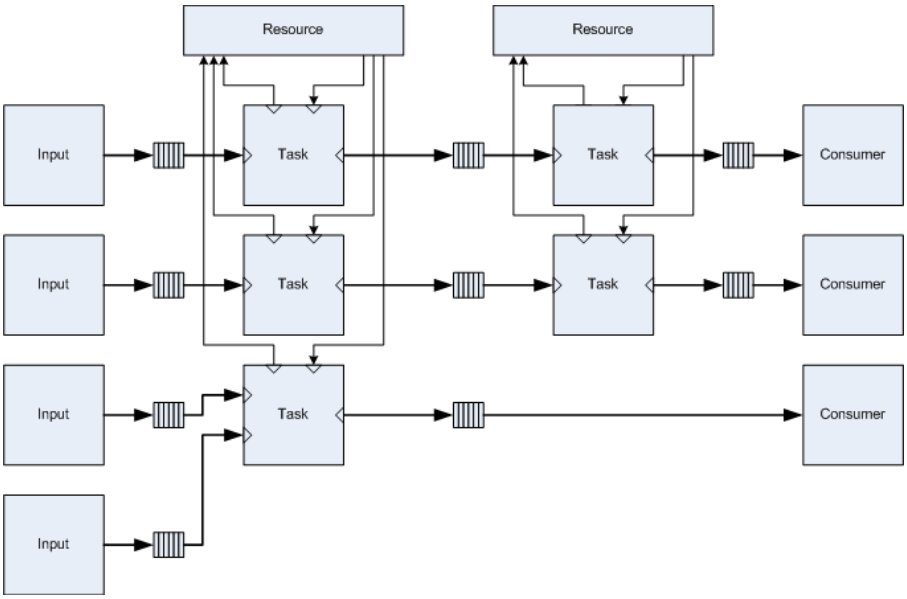

Figure 4.2: *Example of connections between the various modules in the DiMAS API.*

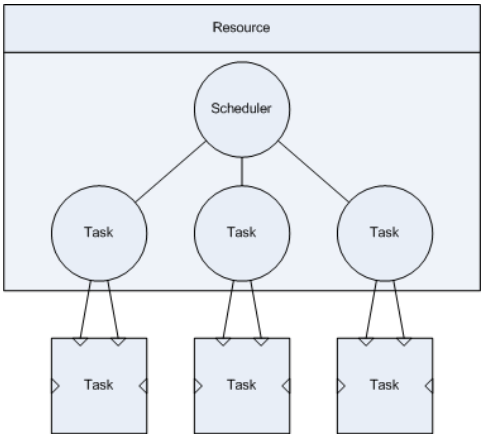

Figure 4.3: *Resources contain one thread for the scheduler and one thread for each task. Each task thread is connected to the corresponding task module.*

basic design, which is the same for all resources, and each subsection will elaborate on how each of the scheduling algorithms are structured.

All resources relies on a set of threads, one thread for the scheduling algorithm and one thread for each of the tasks that are assigned to the resource. The task threads are connected with the task modules as shown in figure 4.3, where each task thread has both an input and an output. When a task module detects a token on the input port, a request is sent to the task thread, which then signals the scheduler. This is however only performed if the task thread is idle. If the task is active the task module will wait until the task thread is done before signalling the new request. The task modules and task threads communicate using channels, where the token structure is sent back and forth, while the task threads and scheduler thread communicate using signalling in terms of global variables.

The design of the EDF and the FP resources have not changed significantly in terms of the scheduling algorithm. There is however a need for modelling the variability in execution time, and therefore the assignment of execution time for each task has been extended so that it can be based on an input trace file as well as a constant value.

4.4.1.1 Fixed priority scheduling

The scheduler uses a global list containing the remaining execution times for all the tasks assigned the resource, and selects the next task to be run by looping through the list in priority order and selecting the first task that has not finished execution. The task is signalled to start, and the scheduler will wait for a new event from the tasks. This event is either due to the running task finishing the execution, or another task trying to pre-empt the running task, due to it getting a new request from the task module. The task threads are each responsible for updating the remaining execution time in the list before the scheduler is activated. The design of the FP scheduler has not changed significantly from Pesimdes.

4.4.1.2 Earliest Deadline First scheduling

Similarly to the FP scheduler there is also a list containing the remaining execution time for each task assigned the resource. An additional list is used for the EDF scheduler, which contains the relative deadlines of each task. Every time the scheduler is woken up, either by the running task completing, or another task receiving a request, the scheduler starts by updating the relative deadlines for each task. The task with the smallest relative deadline is then signalled to run by the scheduler.

The EDF scheduling algorithm is typically used for hard real-time applications, where deadline misses are not tolerated. The DiMAS simulation API is however also concerned with simulating soft real-time applications where deadline misses are acceptable. Therefore all tasks assigned to an EDF resource must specify if it is a hard or soft real-time task. For hard real-time tasks the simulation will terminate if deadlines are missed, while for soft real-time tasks the deadline will be set to zero, and simulation will continue.

4.4.1.3 Constant bandwidth server scheduling

The CBS requires a EDF scheduler to do the actual scheduling. A processing element can have some tasks which are scheduled only using the EDF, but also some tasks which are scheduled by a CBS. Often there will be multiple CBSs, which have one or more tasks assigned. The basic idea is that each time the EDF scheduler needs to update the relative deadlines for the tasks, the CBS is used for assigning the deadlines of each task that is assigned the CBS. Figure 4.4 shows how the EDF scheduler can be combined with CBSs for scheduling four tasks. The figure is slightly misleading since it does not show that the task threads and the scheduler thread also communicate as is the case.

The EDF scheduling resource is modified in such a way that it can communicate with one or more CBS modules, thereby realizing the CBS scheduling algorithm.

The only addition to the scheduler thread is that the CBS modules must all update the relative deadlines for the tasks that they are assigned to, when the EDF scheduler updates the relative deadlines of ordinary tasks. The modifications in the task threads are some what more elaborate. Each time a request from the task module is received, the task thread adds a request to the CBS queue if the task is a CBS task. Every time the task thread receives a notification from the scheduler thread to start execution the task thread notifies the correct CBS that execution has started. And once the task execution is halted, the task thread notifies the CBS of this. Once a task is terminated the task thread removes the task from the CBS queue.

The CBS modules contain a list containing tokens that corresponds to task request. This list is utilized as a First Come First Served (FCFS) list, where a task token is not removed from the list until it has terminated its execution. The CBS thread waits for a task thread to signal that it has begone execution. Once started the CBS server waits either for the task thread to notify that the task has stopped execution or until the server bandwidth has exhausted. If the bandwidth was exhausted, the CBS notifies both the running task and the scheduler thread and updates the CBS deadline accordingly. The CBS thread now waits for a notification from one of the assigned tasks, that it is starting execution.

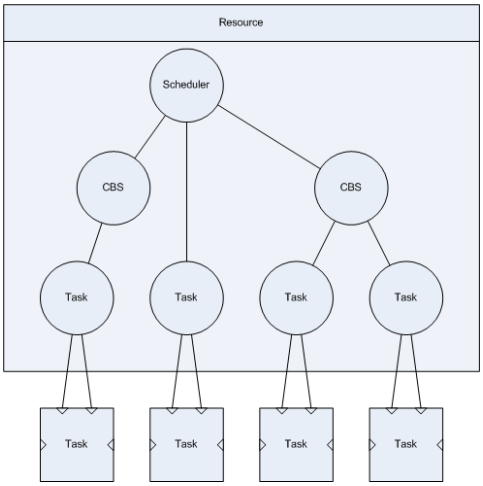

Figure 4.4: *Example of a processing element with two CBSs and four tasks.*

4.4.1.4 Linux scheduler

Only the non real-time scheduling part of the Linux O(1) scheduler is part of the DiMAS simulation API. The scheduler uses a time sharing algorithm, where the size of the time slots vary depending on priorities. The scheduler uses two queues for separating tasks depending on whether they have spent their assigned time quantum or not. These queues are called active and expired.

A structure for containing the ID, and the remaining time quantum for a task is used as member in the active and expired queues. It is these tokens that the scheduler uses every time it is activated. Each of the two queues are structured as an array of queues, since each part of the array characterizes the priority that all tasks at that index has. This structure is what is shown in figure 2.12 on page 23.

The tasks with the highest priority in the active queue is always scheduled onto the processing element, and scheduler removes the task token from the active queue temporarily. The running task is pre-empted either if the time quantum is expired, if the task terminates or if another task receives a request from the task module. If the time quantum of the running task has depleted, the remaining time quantum is updated and the task token is put in the expired queue. If the time quantum is not spent the remaining time quantum is updated and the token in reinserted into the active queue. If the active task terminates, the scheduler signals the task with the highest priority in the active queue. Once the active queue contains no more task tokens, the active and the expired queues are switched.

In this resource the scheduler thread does not know the remaining execution times for the tasks, only the individual task threads knows this. Once a task terminates a global variable is set by the task and the scheduler is notified. The task threads insert the task token into the active queue every time there is a request from the task module. The task thread will only receive a new task request from the task modules if the task thread is idle, which is similar to the FP and EDF algorithms.

4.4.2 Input and output

This section will present details on the information contained in the input and output files and how these files are to be structured.

All input and output files are simple text based ASCII files. The files follow the usual rule, that lines beginning with a # are assumed to be comments and will not be read. The # must however be put as the first character in the line.

As stated in section 4.2 the simulation API can use input trace files for token generation and for retrieving the execution time of a task. The trace files are of course only interesting when variability needs to be modelled. The token generators can only work with files where the first column contains the delta value for when the next token should be generated. The value is an integer and the time base is specified when the input generator module is initialized.

The various resource modules have three different ways of retrieving the execution time. The first is a static value that is set when the task is assigned the resource, another option is using a trace file containing the execution time, and the third option is a trace file containing the processor cycle count. The two latter, which use a trace file are able to read a predefined column in a trace file, and use this for simulation. Retrieving the execution time from a trace file requires that the time base is specified before the simulation is begun. Likewise if the cycle count from a trace file is used, the processor speed must be specified before simulation is begun.

Section 4.1 stated that the three output that the DiMAS simulation API can provide are buffer backlog, response times and lost frames. The buffer backlog is output into a Value Change Dump (VCD) file, which can be read either using a wave viewer like *GTKwave* for Linux, or using the VCDParser[1] application developed during this thesis. One vcd file is created, where all the traces are stored in.

The response times can be logged using the two output modules *display* and *display and write*, the file contains four columns being a sequence number, the total latency through the system, the time of arrival and the total execution time spent on the token. One file is created for each input generator in the system.

The last output file is used for logging the frame loss, this is done by the output module *Consumer*. The file contains three columns, where the first is a counter corresponding to the frame number, the second column contains the number of tokens in the buffer, and the last column contains the time at which the frame loss was registered. Notice that the file only contains entries for lost frames, so if no frames are lost, the file will be empty. The Consumer module can further print a summary at the end of the simulation, where the total frames lost and frames not lost is written.

[1] See appendix A.3 for more information

Chapter 5

Implementation of the Simulator

This section presents the implementation of the DiMAS simulation API, primarily based on class diagrams. Some parts of the DiMAS API are unmodified modules from the Pesimdes API. The unmodified modules from the Pesimdes are only lightly described, and the reader is referred to the master thesis [SP06] by Simon Perathoner for more details.

5.1 Communication

Each *Task* module has one or more input ports and one or more output ports. The ports are each connected to a buffer, either being of the type *sc_fifo* or *my_sc_fifo*. The latter is the buffer that was introduced in Pesimes, which has additional statistical features. The *task* class is the original from the Pesimdes API and has not been changes. The *task* class contains three constructors, where one is used for one input and one input port. The second constructor is used if the task should send the tokens to more than one receiver. The third constructor is used for *task* objects, which has multiple input- and multiple output ports.

The *task* modules wait for an input token from the input port and forwards the token to the resource that it is assigned, once it receives a token. Tasks which have multiple input ports either use a join operation where they wait for tokens on all the input ports before forwarding the tokens, or just forward the tokens as they are received. The resource returns a token to the *task* module once it has completed the execution. The *task* module then writes the token to the output port and is ready for reading a new token from the input port. These operations are handled by the *activation* functions, which are implemented as threads. The threads do blocking reads from the *in* port and writes the token to the internal buffer, which then sends the token to the resource. The class diagram for the *task* module is shown in figure 5.1.

The *my_sc_fifo* class from the Pesimdes API has further been extended in the DiMAS API to support tracing using the tracing functionality in SystemC. The object

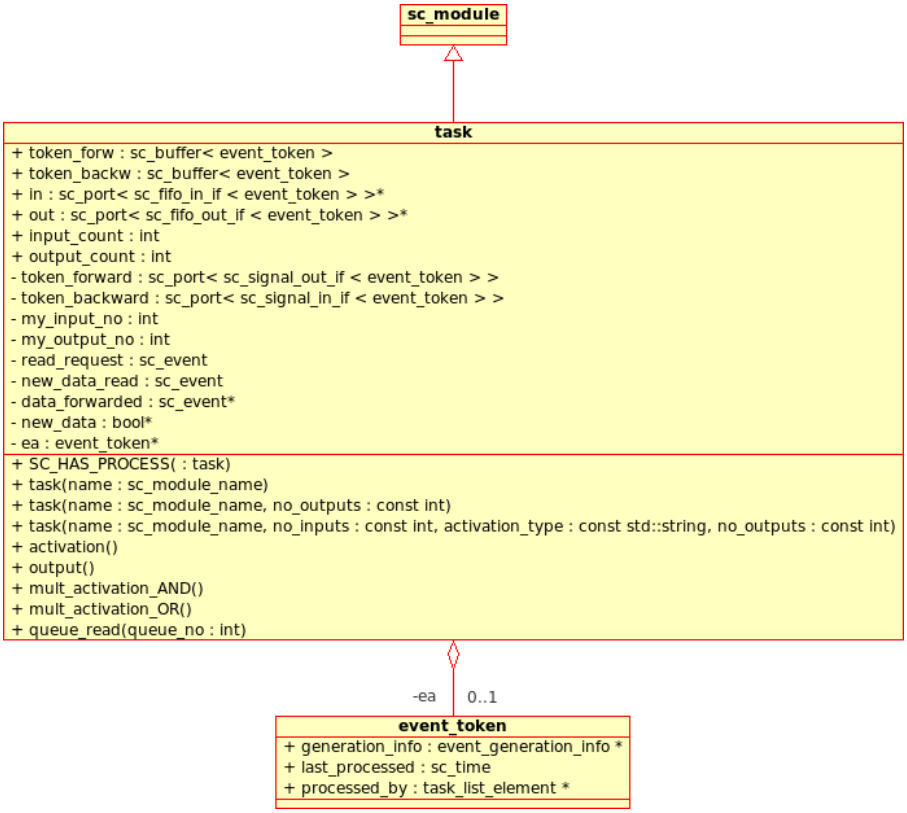

Figure 5.1: *Class diagram of the task class from Pesimdes, which is used unmodified in the DiMAS API.*

is capable of tracing the buffer backlog, the amount of free space and an overflow flag. The variables *free*, *usage* and *overflow* were added to the class. These variables are updated in the protected function *update*, which is a special function which is part of the *primitive channel* object, that the *sc_fifo* and *my_sc_fifo* classes inherits from. This function is executed at the end of each time step during simulation.

The function *logger_trace* has also been added and is used for adding the buffer variables to a trace file. The function takes three arguments. The first argument is a pointer to an open *sc_trace_file* object. The second argument is a string containing the name that should identify the variable in the trace file. The third argument is a string representing 3 bits, where each of the three variables from the buffer can be added to the trace file by setting the character to 1. The bit order is as given below:

"<free><usage><overflow>"

The bit string "111" will add all three variables, while the string "100" only adds the free variable to the trace file. Figure 5.2 contains the class diagram of the *my_sc_fifo* module.

A set of structures are used as tokens in the communication. The structure *event_token* is the main token, which the *task* and the *resource* modules communicate with.

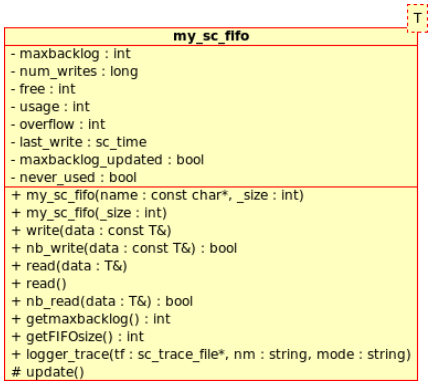

Figure 5.2: *Class diagram of the modified my_ sc_ fifo class, which is used instead of the standard SystemC FIFO buffer.*

All the buffers in the test bench that are used for connecting the various input generators, tasks and output devices need to be of the type *event_ token*. The *event_ token* structure contains three elements being *generation_ info*, *last_ processed* and *task_ list_ element*. The *generation_ info* is an additional structure, that the input generator creates. The structure is a linked list, where additional generating information is appended if there are joins in the test bench. The *event_ generation_ info* structure contains the name of the input generator, the generation time and a sequence number. The structure further contains the variable *total_ execution_ time*, which contains the total amount of time the tasks have spent executing the token. The *task_ list_ element* is simply a linked list containing the names of the tasks, which have processed the token. Figure 5.3 contains the class diagram of the three structures.

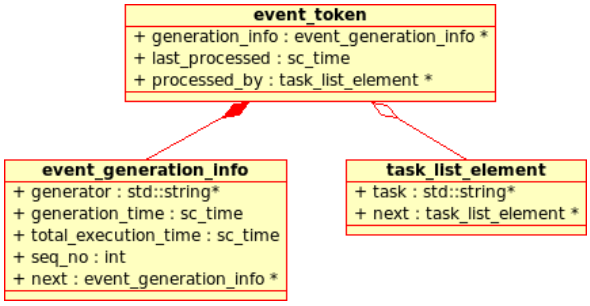

Figure 5.3: *Class diagram of the three structures which are a central part of the communication between the various modules in a test bench.*

The input generators from the Pesimdes API are used in the DiMAS API and have not been modified. The class diagram for the seven types of generators are in the appendices B on page 115. The classes are implemented as SystemC modules that all have a function *main*. This function is a thread, and simply calls the SystemC function *wait* with the time that the tread should wait as parameter. Once the period has passed the module creates a new *even_ token* with the name of the input generator, a sequence number and the time the token is generated.

All seven input modules takes a string representing a file name. The *input_ from_ trace*

uses this to read the delays that it should wait before generating a new event. The rest of the input modules on the other hand prints the time that they wait every time to the file. This is useful when running tests with random periods, if two or more scheduling algorithms need to be compared. The first test is used for generating the delays in a trace file, and the remaining of the tests are performed using the *input_from_trace* module, in order to ensure that the resources are presented with the same input data.

5.2 Scheduling

The greatest difference between Pesimdes and DiMAS are the schedulers, where the algorithms have been modified in order to support soft real-time tasks as well as hard real-time tasks. Further the resources are able to extract varying execution times from a trace file, and thereby not restricting simulation to a static execution time[1] as in Pesimdes. The feature of extracting the execution time is performed in the same manner for all the resources. Therefore polymorphism has been implemented where a base class *resource_n_tasks* has included the common variables and methods. This class can however not function as a regular resource, since the core functionality has been excluded. Figure 5.4 is the class diagram for the *resource_n_tasks* class. The object uses various arrays as data structure, where the indices in these arrays determine which task the data is related to.

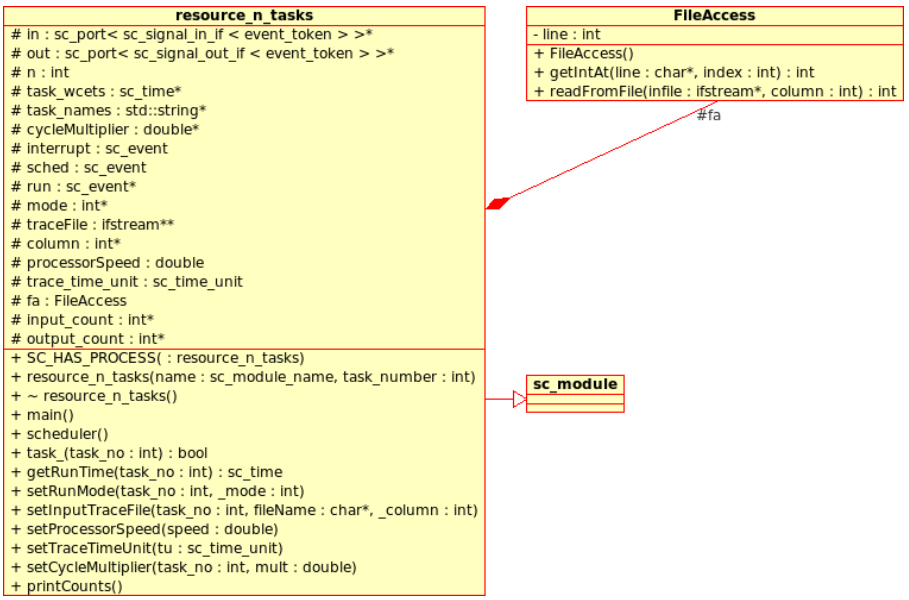

Figure 5.4: *Class diagram for the resource class all the scheduling class inherit from in the DiMAS simulation API.*

Since the *resource_n_tasks* class is used as an abstract class all variables are defined with the *protected* keyword in order for the inheriting classes to gain access to them. Each resource contains an array of input ports and an array of output ports. These

[1] This is typically the WCET for hard real-time applications.

ports are mapped onto the individual tasks that are assigned the resource. This assignment is performed with the *assign* function in the individual scheduling classes. The variable n defines the number of tasks that are assigned to the resource, and is assigned once the object is initialized. The *tasks_ wcet* array is only used for tasks which use a static execution time during simulation.

Three *sc_ event* variables are used in all of the resources namely *sched*, *interrupt* and *run*. The *sched* variable is used by the running task to signal the scheduler that an event has occurred. This is either caused by the task terminating or by another task requesting access to the resource. The *interrupt* variable is used by tasks that are woken up to preempt the running task. The *run* variable is an array, and is used by the scheduler to signal a task, that it is scheduled onto the processing element and can start executing. The communication between the various threads is shown in figure 5.5.

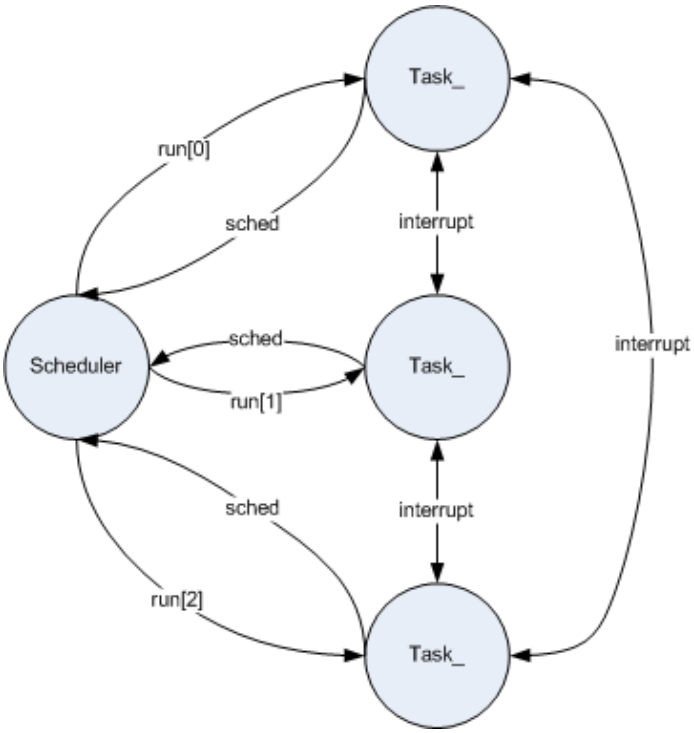

Figure 5.5: *Synchronization mechanisms between the scheduler and the tasks using the three sc_ event variables sched, interrupt and run.*

The main functionality in the *resource_ n_ tasks* class is the ability to retrieve execution times from trace files, either being a trace file containing time values or a trace file containing processor cycle counts. This can be chosen individually for each task assigned the resource. The *setRunMode* function is used for specifying for each task which way the execution times should be obtained. Table 5.1 shows the possible run modes that can be used during simulation. The *setTraceTimeUnit* function must be called for tasks using the execution time trace in order to specify which time unit the values in the trace files are stored as. For tasks using processor cycle count traces the function *setProcessorSpeed* and optionally *setCycleMultiplier* must be used, in order for the resource to be able to calculate the correct execution times. The function *setInputTraceFile* is used for specifying the name of the trace file, and in which

column the trace data is located. This function must be called for each of the tasks that retrieves the execution time information from a trace file. When this function is called the corresponding *ifstream* pointer in the *traceFile* array is opened.

Value	Description
0	Static execution time, based on the WCET specified when the task is assigned a resource.
1	Execution time is based on a trace file, where the values are given as time. The default time unit is NS.
2	Execution time is based on a trace file, where the values are given as processor cycles.

Table 5.1: *Possible run modes for the resources, which are specified using the* setRunMode *function.*

The *resource_n_tasks* uses the *FileAccess* class which given a pointer to an open file and a column number, can extract the next number from the file. The number in the file must be a valid integer.

The *resource_n_tasks* class contains three additional functions, *main*, *task_* and *scheduler*. These are just empty functions, which each of the resources inheriting from this class must implement. The *scheduler* function is a thread and contains the scheduling algorithm. The *task_* represents a task, and basically reads a token from the *in* port, and once it is scheduled by the scheduler it runs until it is preempted or until the task terminates. When the task terminates the token is written to the *out* port, and the thread is suspended waiting for a new input request on the *in* port. The *main* function is a thread which only is run at the start of the simulation, where there is generated a task thread for each task, that is assigned the simulator. This is done in a dynamic way using the *sc_spawn* function call, which is part of the SystemC library.

5.2.1 Fixed-Priority

Figure 5.6 shows the class diagram for the Fixed-Priority preemptive resource. The resource inherits from the *resource_n_tasks* class as all the other resources in DiMAS. The *resource_n_tasks_fixed_priority_preemptive* class adds an array containing the remaining execution times and an array containing the priority order of the tasks assigned the resource. Each array element corresponds to a task, where the index is determined at the elaboration phase[2] using the *assign_task* function.

The *assign_task* function sets the WCET and the priority of a given task, where the *pos* field is used for identifying which index in the various arrays[3] the task is located at. The highest priority starts at 1 and upwards, while the pos field starts at 0. The *wcet* field can simply be set to *SC_ZERO_TIME*, if the task is to retrieve execution times from a trace file.

[2]SystemC defines the elaboration phase as the time where the test bench is constructed, before any simulation is commenced.

[3]These are: executing_times_left, run, in, out, task_wcet, task_names, cycleMultipliers, mode, traceFile, column, input_count and output_count.

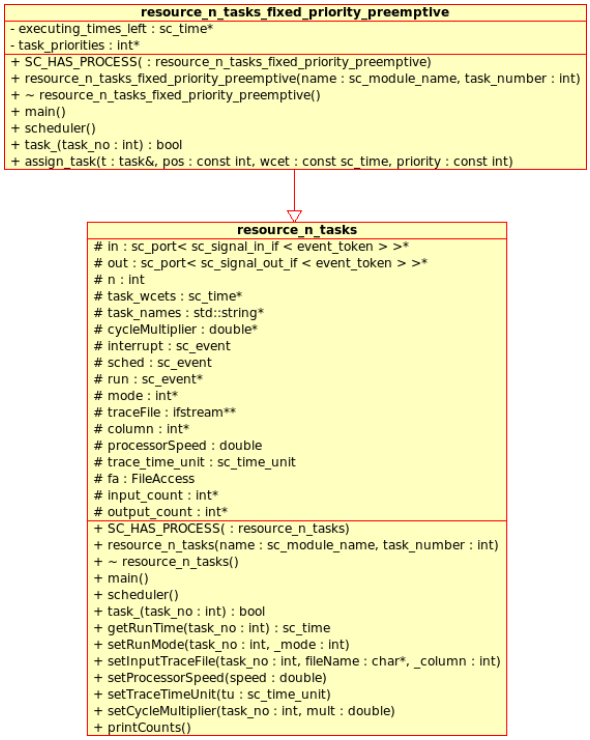

Figure 5.6: *Class diagram for the Fixed-Priority Scheduling scheme in the DiMAS simulation API*

The *scheduler* thread is activated every time a task either receives a request on the *in* port or when a task terminates. The scheduler decides which task that should be assigned the processing elements by starting with the top priority task and checking whether it has any remaining execution time and working downwards until there is a task, that can be scheduled. This operation is performed using the *executing_times_left* array in conjunction with the *task_priorities* array.

The *main* and the *task_* threads both perform as described earlier. Every time the task receives a request on the *in* port, the *task_* thread updates the *executing_times_left* array. The new execution time is found using the *getRunTime* function inherited from the parent object *resource_n_tasks*. It is the *task_* thread that updates the *executing_times_left* every time the task has been scheduled onto the processor.

5.2.2 Earliest Deadline First

Figure 5.7 shows the class diagram for the EDF preemptive scheduling resource. The resource is constructed such that it can be used in conjunction with one or more CBSs as is described in section 5.2.3. This section will only describe the EDF related variables and functions.

The variables which are related to the EDF scheduling resource are *task_rel_deadlines*, *executing_times_left*, *remaining_times*, *update_remaining_times* and *real_time_type*.

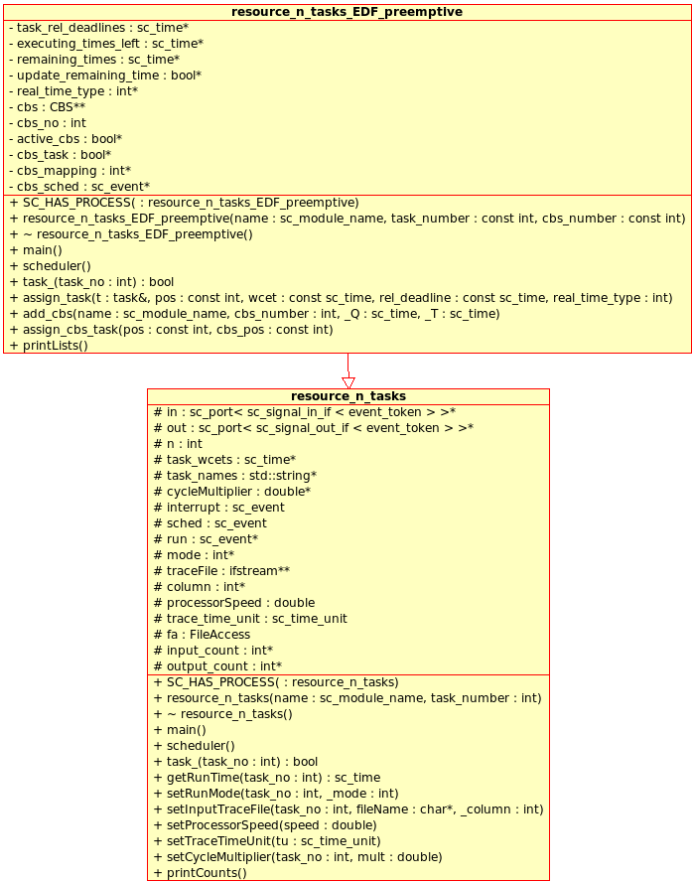

Figure 5.7: *Class diagram for the Earliest Deadline First scheduling scheme in the DiMAS simulation API*

All of the mentioned variables are arrays, where each array index corresponds to a task assigned the resource. The *task_ rel_ deadlines* contains the period for each task, which is specified in the elaboration phase using the *assign_ task* function. The *executing_ times_ left* contains the remaining execution time for each task, similarly to that in the FP resource. If a task is not active or have any pending requests the time will simply be *SC_ ZERO_ TIME*. The *remaining_ times* and *update_ remaining_ times* are related to the deadlines, where the *remaining_ times* array contains the relative deadline for each task. The *update_ remaining_ times* array is used for ensuring that the deadline is not updated the for the task which is preempting the running task.

When tasks are assigned the resource using the *assign_ task* the WCET and relative deadlines are specified for each task. The WCET can simply be set to *SC_ ZERO_ TIME* if the execution times are retrieved using a trace file. Further the task is either defined as a soft or a hard real-time task.

Every time the *scheduler* thread is activated it updates the remaining time of the relative deadlines for all tasks, except those that have been flagged[4] not to be updated. In addition the scheduler checks for all tasks if there are deadline misses. The

[4]This is relating to the *update_ remaining_ times* array.

scheduler writes an error message to the terminal, and terminates the simulation for hard real-time tasks, while it only prints the error message for soft real-time tasks. Once the deadlines have been updated, the scheduler searches for the task with the earliest deadline and signals this using the run variable with the corresponding task index.

5.2.3 Constant Bandwidth Server

The Constant Bandwidth Server (CBS) scheduling algorithm uses an additional module, which is used in conjunction with a deadline oriented scheduling scheme as EDF. The EDF resource has been extended to support the CBS scheduling module. Figure 5.8 contains the class diagram for the CBS module and the EDF resource.

A CSB module is initialized for each CBS that needs to be scheduled. The CBS module uses a *cbs_ queue*, which is a linked list where requests from tasks that are assigned the CBS are queued.

The *cbs_ queue* is sorted according to the time a task has been waiting[5], therefore when a task is added to the queue, the task ID and the time is was last processed must be given as arguments to the function *queue_ new_ task*. The *remove_ from_ queue* function in the queue data structure, takes the task ID as argument and removes the first item in the queue if the ID of the task in the queue is equal to the argument. The *peek* function returns the tasks ID of the first element in the queue.

The CBS class contains a series of pointers which are used to reference some of the data structures in the EDF resource. The *cbs_ tasks* points to the *cbs_ task* array in the EDF resource, which contain boolean elements defining whether or not a task is a CBS task. The *cbs_ task_ ids* points to the *cbs_ mappings* array, that defines which CBS server a given task is assigned to. The *task_ deadlines* points to the *remaining_ times* array, which contain the relative deadlines of the tasks that are assigned the EDF/CBS resource. The CBS updates the deadlines of the tasks that have been assigned to it using the *task_ deadlines* pointer, where the *cbs_ mappings* array is used by each CBS for identifying which task deadlines to update.

The last three pointers are used for synchronizing the CBS, EDF scheduler thread and all of the task threads assigned the resource. These are *cbs_ sched*, *sched* and *task_ interrupt*. The *sched* and the *task_ interrupt* respectively point to the *sched* and *interrupt* variables in the *resource_ n_ tasks* class. The *cbs_ sched* is also of type *sc_ event* and is used for synchronizing the CBS and its corresponding *task_* threads.

The variable n specifies the number of tasks queued in the CBS queue. The *cbs_ id* is the ID of the CBS, and the variables Q and T respectively correspond to the server bandwidth and period. The variable *bandwidth_ left* is the remaining server bandwidth and the *server_ deadline* is the absolute deadline of the CBS. Note that the CBS itself uses absolute deadlines, where the *scheduler* threads in the EDF resource use relative deadlines.

[5]The first element in the list is the task that has been waiting for the longest time.

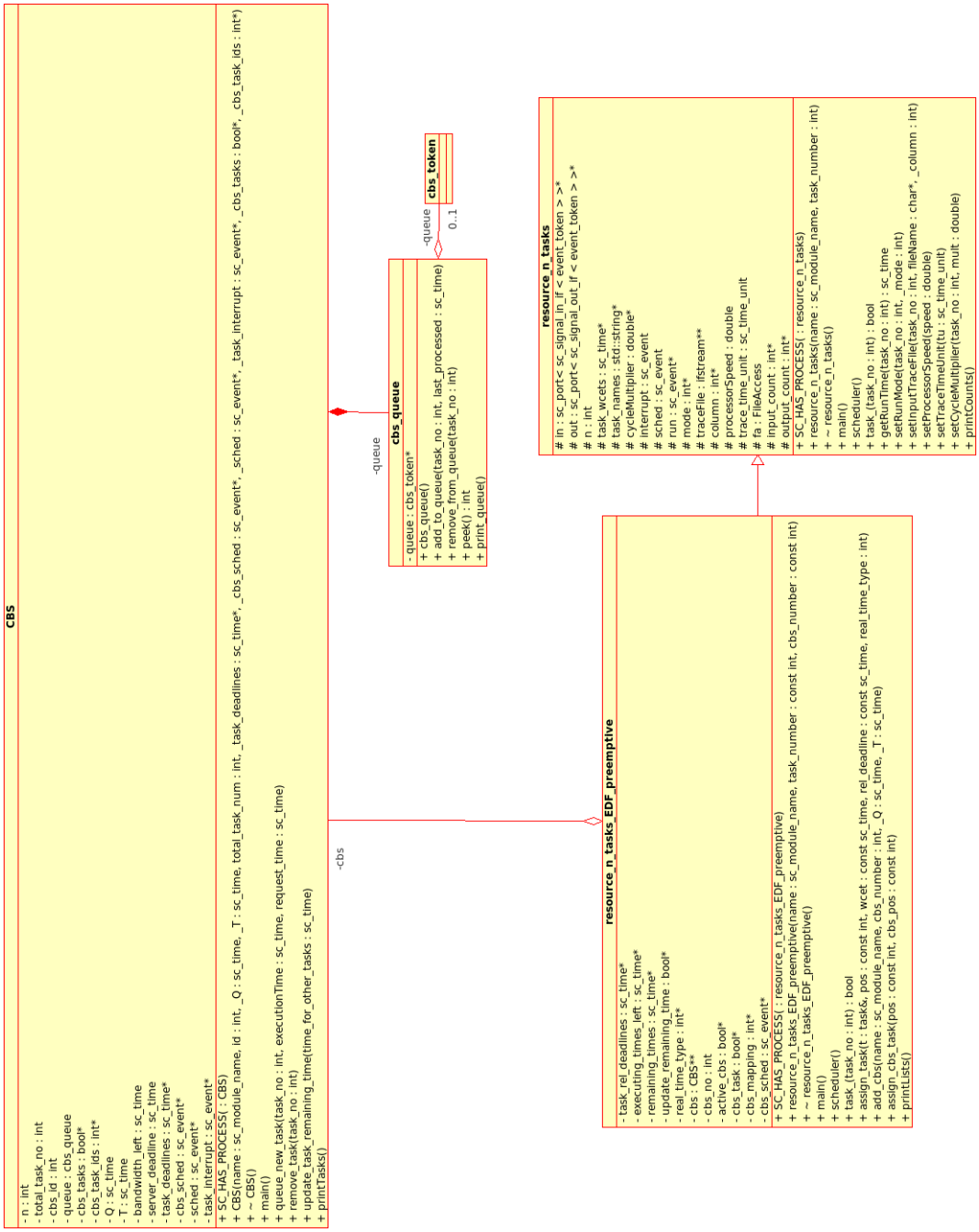

Figure 5.8: *Class diagram for the Constant Bandwidth Server scheduling scheme in the DiMAS simulation API.*

The function *main* in the CBS module is a thread, and simply waits for an event on the *cbs_sched* variable from one of the tasks assigned the CBS. The thread then waits until the server bandwidth is depleted or that the running thread stops executing. This can either be caused by the task terminating or by another task preempting it. When the running task stops execution it signals the *main* thread using the *cbs_sched* variable. Once the server bandwidth is exhausted the *main* thread signals the running task to stop execution using the *task_interrupt* variable.

It then replenishes the server bandwidth and updates the server deadline by adding the server period to the current server deadline. Finally it signals the *scheduler* thread using the *cbs_sched* variable, which then determines which task should be run next.

The function *queue_new_task* adds a task to the CBS queue. The execution time and the request time of the task is passed as arguments in order to calculate whether or not the server deadline needs to be updated. This is done as described in section 2.5.3. The *update_task_remaining_time* is the function that updates the deadlines of the tasks that are assigned the CBS server. The EDF scheduler calls this function when the deadlines for all the tasks assigned the resource are updated. The function takes a parameter of type *sc_time* and this time is assigned to all the tasks which are assigned the CBS server, except for the first task in the CBS queue, which is assigned the server deadline. This ensures that only the task that is the first element in the queue is served with the server deadline, while the rest must have a higher deadline in order not to be scheduled.

The six additional variables that have been added to the EDF resource are *cbs*, *cbs_no*, *active_cbs*, *cbs_task*, *cbs_mapping* and *cbs_sched*. The *cbs* variable is an array of pointers to the CBS modules which are assigned the resource. The number of CBSs is given when the resource is initialized, this is stored in the variable *cbs_no*. The *active_cbs* variable is an array of boolean elements, which specify whether or not any tasks have been assigned to the corresponding CBS. The *cbs_sched* is at the EDF resource class an array of *sc_event* elements, where each element corresponds to the *cbs_sched* variable the CBS module.

Two functions have been added to the EDF resource in order for it to support the CBS, these are *add_cbs* and *assign_cbs_task*. The *add_cbs* function is used for adding a CBS to the EDF resource. The function takes the server bandwidth and period as arguments. Further the position in the CBS related arrays must be specified as argument. The function will initialize the CBS module, where the addresses of the relevant data structures in the EDF resource are given as arguments. Further the ID, name, server deadline and period of the CBS is parsed on to the CBS module. A CBS task must first be assigned the EDF resource using the function *assign_task* and afterwards assigned a CBS using the *assign_cbs_task* function. The *assign_cbs_task* takes two arguments being the array positions of the task in the task related arrays, and the array position in the CBS related arrays[6] corresponding to the CBS that the task is assigned.

The *scheduler* thread is extended with a few features in order to support one or more CBSs. First of all the scheduler does not update the relative deadlines of the tasks which are assigned a CBS. This is avoided using the boolean array *cbs_tasks*. Further once the deadlines have been updated for the tasks not assigned a CBS, the scheduler calls the *update_task_remaining_time* function for each of the CBSs assigned the EDF resource.

CBS specific functionality has also been added to the *task_* threads. Once a *task_* thread receives a request on the *in* port, the event token is read. The field *last_processed*

[6] These are: *cbs*, *active_cbs* and *cbs_sched*.

in the event token is used as the request time, when the task is added to the corresponding CBS queue, using the function call *queue_new_task*.

The *task_* thread signals the CBS that it is assigned, once the scheduler signals the thread to start execution. The *cbs_sched* pointer array is used in conjunction with the *cbs_mapping* in order for the *task_* thread to signal the correct CBS. The *remove_task* function is called on the CBS object when a task has terminated, thereby removing the task from the CBS queue.

Note that the CBS scheduling implementation is not exactly as seen in figure 4.4 from the design section, where the tasks only seem be to registered at the CBS module. Tasks which are scheduled using a CBS is registered both at the EDF resource and at one CBS module. The EDF resource thereby know all tasks which are assigned the resource, and not only one task per CBS module, as you might interpret from the figure.

5.2.4 Linux O(1)

As seen in the class diagram in figure 5.9 the Linux resource has an additional data structure which is a linked list. This data structure is used for the task queues, where each priority level from 0 to 139 has a queue containing tokens representing a task request.

The variables which are used in the resources are *queue_1*, *queue_2* that represent the actual priority queues. The *active* and *expired* variables are the pointers to the two queues that switch each time all tasks in the active queue have exhausted their time quantum or have terminated. The variable *_slots* is an array containing the time quanta for each priority queue.

The *scheduler* thread implements the priority oriented time sharing algorithm that the Linux O(1) scheduler uses. This scheduling technique is based on the active pointer, where the task with the highest priority[7] is scheduled onto the processing element. Once the scheduler has signalled the task to run, it waits for either the time quantum to deplete, another task making a request or for the running task to terminate. When there are no more task request in the active queue array, the active and the expired pointers are switched. The time quantum of a task token is replenished and the token put into the expired queue, if a task is not completed within the given time quantum.

The *task_* thread has an internal variable defining the remaining execution time. This is updated every time the task receives a new request on the *in* port, or when the task stops execution either due to task termination or another task preempting it. When a request is received the *task_* thread further adds a token to the *active* queue, using the *add_to_queue* function in the *linux_queue* object. The *task_mappings* array is used to find the correct index in the *active* array where the function should be called.

[7] lowest value in the range [0:139]

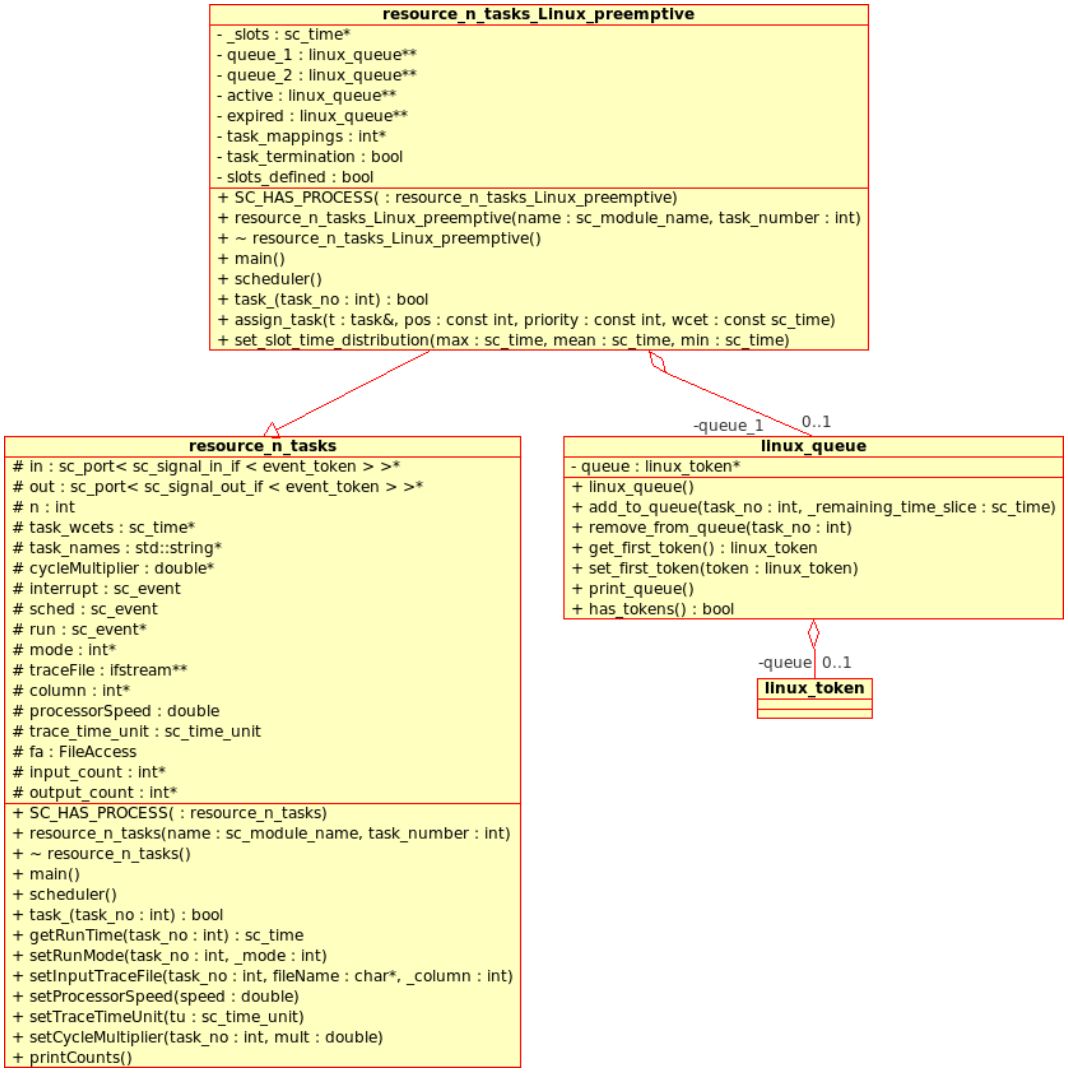

Figure 5.9: *Class diagram for the Linux O(1) scheduling scheme in the DiMAS simulation API.*

The priority of the task is specified in the *assign_ task* function. This value is in the range [0:139]. As for the two previous resources, the WCET can also be given, in the case that a task is using a static execution time during simulation.

The function *set_ slot_ time_ distribution* must be called during the elaboration phase in order for the resource to assign the correct time quanta to each priority queue. This function takes the max, mean and min time quanta the resource should use, which respectively corresponds to priority 0, 99 and 139 [8].

5.3 Output devices

Figure 5.10 shows the class diagram for the three classes which are the sinks that can be used in the DiMAS test benches. The *Consumer* class is used for emulating an

[8]Priority 0 is the highest priority, and has the largest time quantum.

output device such as a screen, which reads a certain amount of tokens from the buffer at a specific rate. The *output_ display* class is modified version from the Pesimdes API. The *output_ display* class is used for collecting statistical information regarding response times. The *output_ display_ write* class has the same functionality as the *output_ display* class, but further writes the tokens to an output port for further processing.

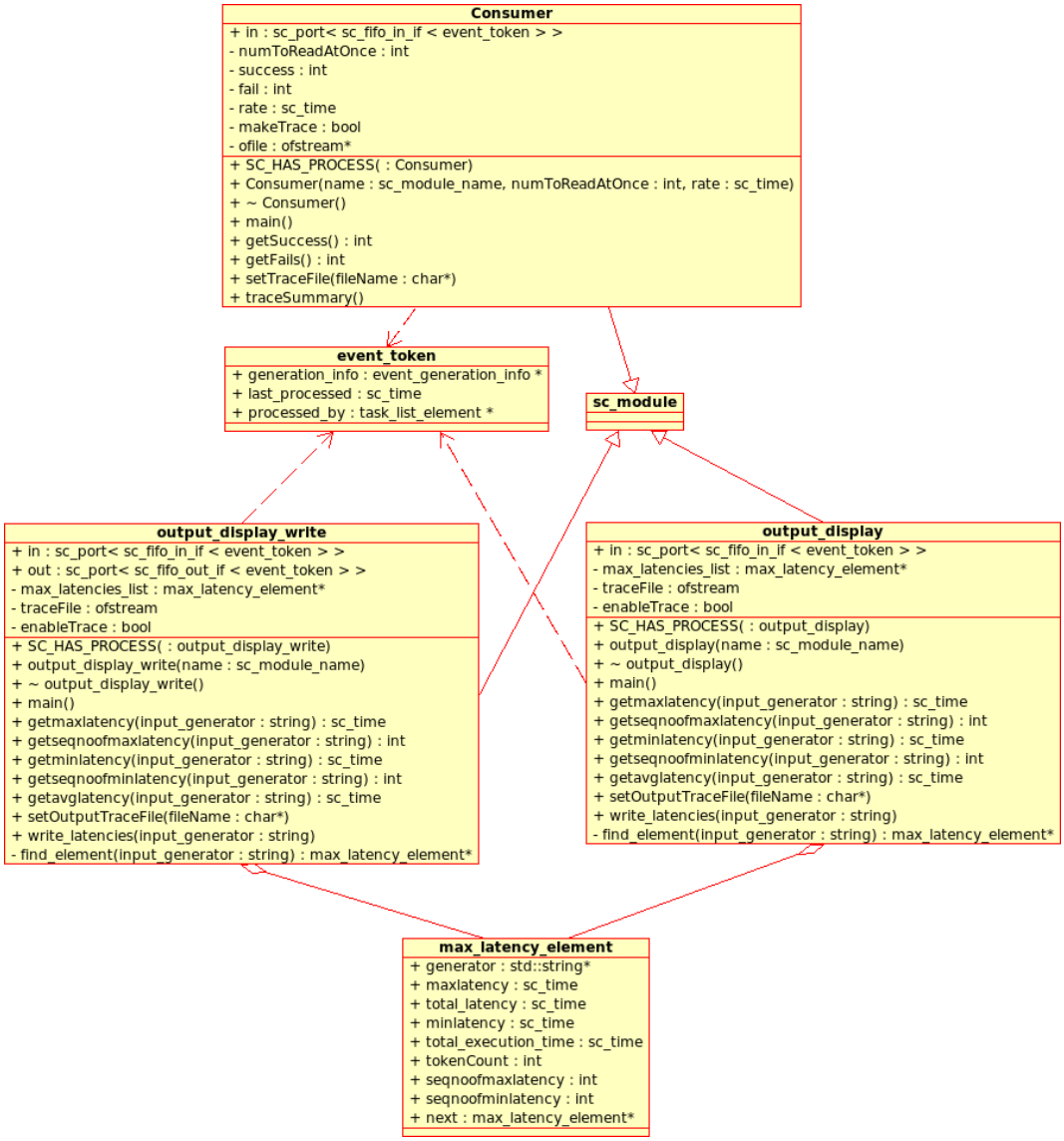

Figure 5.10: *Class diagram for the output devices which are part of the DiMAS simulation API.*

5.3.1 Consumer

The *Consumer* class reads a fixed amount of tokens from the input port at a fixed rate, both of which are specified once the object is initialized. The *Consumer* module uses the *main* thread, where the tokens are read. The period is implemented using the *wait* function that is part of SystemC, where the parameter is the period of the reads.

Initially the thread suspends waiting for the first token to arrive on the input port. Once this is received, the thread starts an initial buffering by waiting for the time equal to two times the period before reading any tokens. If there are not sufficient tokens in the buffer connected to the input port, the object will log an error to the terminal. The object will further log the error to a file if the function *setTraceFile* has been called. Every time the object tries to read the specified amount of tokens from the buffer, one of the two counter variables *success* or *fail* are incremented, depending on whether or not there are sufficient tokens in the buffer.

The *traceSummary* function is used at the end of simulation and will print a summary of the values of the *success* and *fail* variables to the trace file. The function *getSuccess* and *getFails* are respectively used for retrieving the number of successful frame reads and the number of failed frame reads. Note that the total number of frames which the *Consumer* has attempted to read during the simulation is the sum of the two variables.

5.3.2 Output display

The two classes *output_ display* and *output_ display_ write* are almost identical, save the *out* port in *output_ display_ write*.

The objects each contain a linked list *max_ latency_ element* which contains statistical information, where each element in the list is identified by the name of the input generator which generated the token. The *output_ display* and the *output_ display_ write* uses the *main* thread, which waits for a token on the input port. When a token is received, the thread searches through the *max_ latency_ element* list for the element corresponding to the generator which generated the token. The maximum, minimum and total latency information from the token is recorded into the corresponding *max_ latency_ element*. Further the total execution time is stored in the *max_ latency_ element* corresponding to the input generator, which generated the token. The *output_ display_ write* object then further writes the token from the *in* to the *out* port and then suspends waiting for a new input token similar to the *output_ display* object.

Getter functions are available for retrieving the maximum, average and minimum latencies. Further there are getter functions for retrieving the sequence number of the tokens which had the maximum and minimum latencies. All the getter functions take one parameter, being the name of the input generator of the stream. The reason for using the name of the input generator to track the statistical information, is that there can be *joins* in a test bench, thereby have input tokens from two or more different input generators.

The *output_ display* class in Pesimdes was only able to record maximum latencies. The minimum and average values are added in the DiMAS API. Further the sum of the execution times for all the tokens belonging to a stream are recorded.

5.4 Extending DiMAS

Other types of scheduling techniques can be implemented in the DiMAS API, this is done by implementing a new resource. The new resource should inherit from the

resource_ n_ tasks class, and must implement the functions *scheduler*, *tasks_* and *main*. Where the *main* function simply provides a dynamic way of creating a *task_* thread for each task that is assigned the resource. The *scheduler* thread contains the logic for invoking one *task_* thread into the running mode. The *task_* thread waits for an input token from the *in* port, and once it receives a token it gets the execution time for that specific token, using the *getRunTime* function. The *task_* thread can now start the cycle where it waits for the *scheduler* thread to signal it to start executing and either stop due to termination or to another *task_* preempting it.

The variables *run**, *sched* and *interrupt* are used for synchronizing the various threads in the resource as show in figure 5.5 on page 47. The new resource will then be able use the trace files in order to determine the execution time of each task request, as the four resources which have already been implemented.

The resources typically also implement a function *assign_ task*, which is used for adding a task to the resource. The function typically takes a pointer to the task module, and an array index which is used for internal mappings in the resource. Parameters like deadline, priority and others are typically also assigned the task as arguments in this function.

Chapter 6

Generating traces for the simulator

The traces play a central role in the simulator since we need some good traces to base the analysis on. For the simulator we need traces for the execution time of various multimedia tasks. These traces have been created using the SimpleScalar simulator. This chapter will present how the traces are generated, and what modifications in the SimpleScalar toolset are needed.

Basically the traces are generated using breakpoint sets and logging the number of cycles or instructions at each breakpoint. The log file is further processed where the breakpoints are paired and the difference is calculated and logged to the final trace file. Generation of traces requires four things:

- Global variables inserted into the code that needs augmenting
- Address of the breakpoints in the object file
- Modifying the SimpleScalar to log breakpoint data into text files
- Post processing of the log file to generate the final trace files

The SimpleScalar debugger provides breakpoint functionality for either instructions or for data. In this thesis data breakpoint was found as an optimal solution. The code segments that we are interested in measuring the execution time of, is surrounded by two global variables which are incremented. It is imperative that the global variables are put in such a way that there will always be a start followed by an end breakpoint. This is a manual task and will need some code analysis from the users side in order to place the variables the correct places.

As mentioned in section 3.3.1 the SimpleScalar tool set contains a very simple interactive text based debugger. The debugger already implements breakpoint functionality, but there is however currently no tracing support. The whole SimpleScalar tool set is however very modular, and adding new features is rather easy. The debugger can detect both read, write and executions of data breakpoints.

Three new commands have been added to the debugger:

tracefile This command is followed by the name of the trace file. The file name must not contain any spaces, other characters must be used if word separations are required.

tracecomm This command is used for adding comments the the trace file. This command suffers the same problem of the comments not being able to contain spaces. This command can only be used at the beginning before the actual simulation has begun, and requires that the *tracefile* command has been called.

quietTrace This command is used for enabling and disabling the breakpoint data being written to the terminal[1]. It is nice to have it enabled for debugging purposes, but is too resource demanding when long simulations are performed. The parameters are either *on*, *off* or *show* where the last displays the current state.

The commands are added to the debugger in a command data structure found in the *dlite.c* file. Here the command name, type of arguments and a short description for the help option are added for each command. Further a handle is provided to the function which should be called when the command is executed.

Listing 6.1: *Part of the command array in the dlite! debugger.*

```
1  static struct dlite_cmd_t cmd_db[] =
2  {
3  ...
4     { "tracefile", { "s", NULL }, dlite_traceFile,
5       "Specify the name of the tracefile" },
6     { "tracecomm", { "s", NULL }, dlite_traceComment,
7       "Write a comment in the trace file. Use '_' instead of spaces" },
8     { "quietTrace", { "s", NULL }, dlite_quiteTraceToggle,
9       "Toggle on or off is debug output should be printed to the console.
            Parameters are on/off/show" },
10 }
```

For logging the actual breakpoint data, an additional main function was implemented for the debugger, where the only difference is that it is not interactive and therefore does not wait for a command. The original main function is accessed at the start of the simulation, but once the simulation is initiated the new main function is called instead of the original. The new main function is then accessed every time a break point is encountered, at which point the break point number and number of cycles are logged to the trace file. The new main function requires that the simulator[2] which is used for creating the traces must be altered slightly in the sense that the debugger main function is called twice, once during initialization and once through each iteration of the simulation loop. It is the call in the main simulation loop, where the new main function is called instead of the original.

Before the simulation can commence, the addresses of the global variables need to be identified since the breakpoints are added based on the address. These addresses are found using the *objdump* command and *grep* for quickly finding the address. The global variables are located in the symbols table, which is accessed using the *-t* parameter for the objdump utility. The following command is used for extracting the addresses[3]:

[1] stdout.
[2] being sim-outorder, sim-profile, sim-cache, sim-cheetah or sim-fast.
[3] This is a Linux command.

```
objdump -t <executable> | grep global_
```

In this example all the global variable names have been prefixed with global_. Note that the variables might be assigned new addresses if the source code has been modified and recompiled.

Once the trace files with all the breakpoint data have been generated it is time for post processing where the absolute cycle counts for the breakpoints are turned into cycle counts from the start to the end breakpoint of each breakpoint set. For this the simple *breakDiff* application has been implemented, which can parse a file with up to 10 breakpoint sets. A new trace file is generated for each breakpoint set, where the name of the breakpoint set is used as extension. Section A.1 presents more information about the breakDiff application.

The commands that were used in the SimpleScalar debugger, when measuring the execution time of MPEG video files are listed in Appendix E.

6.1 Resulting trace files

The trace files which were created for the DiMAS simulator are all based on the *sim-profile* simulator. Therefore the values are given as instruction counts and not processor cycles, as we would like to have. Two additional simulations were therefore performed using the *sim-outorder* on a file with motion and one without motion. As already mentioned in section 3.3.1, the sim-outorder simulator present much more accurate result since it is able to simulate pipelines and caches. The results showed that there is a more or less constant ratio of about 1:0.7 meaning 1 instruction is processed in 0.7 processor cycles. This obviously shows how the pipeline has a positive effect on the decoder.

The processor cycle counts obtained are rather large, since it seems some what unlikely that there is need for a 3 GHz processor for decoding a film with a resolution of 720x480. The traces in this thesis was performed using the SimpleScalar simulator where the ARM architecture was given as the target architecture. Therefore the cycle counts can not be compared with a normal processor, which has many times more pipeline stages and on chip support for multimedia decoding.

In order to have test benches where the processor requirements are not well out of the current processor speeds a simple profiling was performed using *gprof* on the MPEG decoder. The profiling was performed on each of the multimedia files used in the test benches[4]. The accuracy of the profiler is however very limited since the measurements are given as seconds with two decimals. The purpose is however simply to get an estimate of a ratio in order to get more reliable results, therefore this is accepted since the variability still will be intact.

The profiling is again split into the VLD/IQ and the IDCT/MP tasks for decoding. The results are summarized in table 6.1.

[4]The two files are: flwr_080.m2v and high_25fps_320x240.m2v

Video	VLD/IQ	IDCT/MP
flwr_080	0.557	0.139
high_25fps_320x240	0.891	0.244

Table 6.1: *Resulting ratios from profiling between instruction counts from trace files and actual processor cycle counts. The values in the table are multiplied with the values from the trace files in order to obtain more accurate processor cycle counts.*

The table shows that the VLD/IQ tasks result in more or less the same multiplier factor as was found using the sim-outorder simulator. This makes sense since the VLD/IQ basically are table lookups and multiplications. The profiling information reveal that the IDCT/MP has a much smaller multiplier than found with the sim-outorder. It is the IDCT/MP that is the resource demanding process, it therefore seems likely that on chip support and advanced pipelining has increased the performance for these operations. Appendix F contains the profiling information and calculations on how the multipliers were obtained.

When applying the multipliers specified in table 6.1 the required processors speeds for decoding the *flwr_080* video file is reduced from approximately 3 GHz to 800 MHz, and for the *high_25fps_320x240* video file the requirements go from approximately 900MHz to 300 MHz, which for both cases seems more reasonable.

Chapter 7

Case studies

There are 8 case studies in this chapter. These are divided into two groups. The first four cases use a test bench with two processing elements, while the next four use a test bench with only one processing element. The four cases studies, which are present in each of the two groups examine the performance of the simulator under different scenarios which are summarized in the following.

TC1 The input streams are periodic, where the period is based on the parameters of the video being simulated.

TC2 There are two periodic input streams and one highly aperiodic stream, which corresponds to a GUI task, which for example is activated when the user needs to access the menu on the device being simulated.

TC3 Jitter is now applied to the two input streams. The jitter is in the area of $\pm 40\%$.

TC4 This test bench is similar to TC2. The focus in this test is to optimize the performance of the CBS scheduling algorithm. Five sub tests have been created, where different values are applied to the CBS parameters.

TC5 This is the same test as TC1, with two periodic input streams. All four tasks are however scheduled onto a single processing element.

TC6 This is the same test as TC2, with the fifth task corresponding to a GUI task. All five tasks are however scheduled onto a single processing element.

TC7 This is the same test as TC3, where jitter is applied to the two periodic inputs. All four tasks are however scheduled onto a single processing element.

TC8 This is a similar test to TC4, six different set-ups for the CBS parameters are simulated. All five tasks are however scheduled on one processing element.

7.1 Evaluation criteria

The test cases in this chapter are evaluated based on *frame losses*, *response time*, *buffer backlog* and *response time delays*. Graphs of the buffer backlog and response

times delays have been created for each case study and are included in the appendices. There is a summary section for each case study, where the results are summarized and briefly discussed.

7.2 Reading this chapter

This chapter is a rather large part of the report, and many of the test cases only show redundant information. All case studies but TC4 and TC8, have been performed both using a system which meets the average resource requirements, and an over designed system. The sections for the subtests are respectively called the *slow processing element(s)* and the *fast processing element(s)*. All of the test cases with the fast processing element(s) show more or less the same pattern, where there are neither buffer over- or underflow, and the response times are also more or less the same no matter which scheduling policy was used.

The summary sections for each of the case studies presents the most interesting results from the various test cases within the case study. The summary sections are therefore good for getting an overview of the results from each of the case studies without reading every test case.

Section 7.6 concludes the results found in the eight case studies.

7.3 General set-up

This section contains common information relating to the case studies that are presented in this chapter. All case studies use the same two stream for modelling the continuous media. The files which have been selected are chosen due to their high variability, where both inter- and intra variability are present. The files are summarized in table 7.1. Plots of the variability for the two streams are included in the appendices H.1 and H.2.

Stream	File name	Frame rate	Resolution
APE_1	flwr_080.m2v	30	720x480
APE_2	high_25fps_320x480.m2v	25	320x480

Table 7.1: *Summary of the MPEG video files, which the inputs to the simulator are based on.*

The initial inputs are periodic streams for all the test cases except for test case 3 and 7, where jitter is added on the inputs. For APE_1 the period is $\frac{1}{30x1320} = 25.252\mu s$, while the period for APE_2 is $\frac{1}{25x300} = 133.332\mu s$, these periods are based on the frames per second and the amount of macroblocks in each file.

The Consumers are set-up to consume tokens corresponding to the fps and resolution of the video streams they are consuming tokens from.
APE_1: 1320 tokens every $\frac{1}{30}s = 33.332$ ms.
APE_2: 300 tokens every $\frac{1}{25}s = 40$ ms.

Task#	min	avg	max
1	2,115	7,618	46,255
2	2,699	14,957	46,205
3	109,250	116,875	118,649
4	109,381	114,860	128,158

Table 7.2: *Instruction count for the various tasks, which are found using the technique described in chapter 6.*

Table 7.2 contains the min, avg, and max instruction counts, which were found using the technique described in chapter 6.

The cycle multipliers are those specified in section 6.1. Table 7.3 is similar to the one found in the generating traces chapter.

Stream	VLD/IQ	IDCT/MP
APE_1	0.557	0.139
APE_2	0.891	0.244

Table 7.3: *Resulting ratios from profiling between instruction counts from trace files and actual processor cycle counts. The values in the table are multiplied with the values from the trace files in order to obtain more accurate processor cycle counts.*

The minimum resource requirements for each task are given in table 7.4. These values are based on the average execution times for each task in table 7.2 and the multipliers given in table 7.3.

Task	Processor speed
1	168 MHz
2	100 MHz
3	635 MHz
4	210 MHz

Table 7.4: *Minimum processor speed requirements for the four tasks based on the average cycle count, and the corresponding cycle multipliers.*

7.4 Two processing elements

Processor speed	P_e1	P_e2
Slow	270 MHz	860 MHz
Fast	860 MHz	1.5 GHz

Table 7.5: *Processor speeds used in the following cases stories, unless otherwise specified.*

Task	Stream	Processing Element	Description
1	P_e1	APE1	flwr VLD/IQ
2	P_e1	APE2	high VLD/IQ
3	P_e2	APE1	flrw IDCT/MP
4	P_e2	APE2	high IDCT/MP

Table 7.6: *Mapping of the various task onto the two processing elements.*

Table 7.7 shows the actual min, avg and max execution times in μs. These times are based on the cycle counts given in table 7.2, the cycle multipliers given in table 7.3 and the processor speeds given in table 7.5.

Task	Slow processors			Fast processors			Period
	min	avg	max	min	avg	max	
1	4.4 μs	15.7 μs	95.4 μs	1.4 μs	4.9 μs	30.0 μs	25.3 μs
2	8.9 μs	49.4 μs	152.5 μs	2.8 μs	15.5 μs	47.9 μs	133.3 μs
3	17.7 μs	18.9 μs	19.2 μs	10.1 μs	10.8 μs	11.0 μs	25.3 μs
4	31.0 μs	32.6 μs	36.4 μs	17.8 μs	18.7 μs	20.8 μs	133.3 μs

Table 7.7: *Actual execution times given as min, avg and max for both the slow and the fast processor. Note that the period column specifies how often a token is generated, and it is this time that the task should complete within.*

7.4.1 Case study 1 - Periodic input

This test case has four sub tests, which each are run with a set of slow processing elements and a set of fast processing elements. There is a test for each of the scheduling schemes implemented in the DiMAS API, being FP, EDF, EDF+CBS and Linux O(1). These four scheduling schemes are tested where the two processing elements just have sufficient processing power, and another test where the processing elements are over powered. There are no tests where the processing elements are underpowered, since this only leads to the buffers overflowing. Table 7.5 contains the processor speeds that are used in the following test cases.

The table 7.12 in section 7.4.1.6 summarizes the results, where frame loss and average response times are listed.

All the figures which are referred to in the section are located in the appendices section H.2, which starts on page 138.

7.4.1.1 Test bench

The token generator trace files are:
APE_1: TC1_ape1_input.tra
APE_2: TC1_ape2_input.tra
Where the values are stored in picoseconds.

The execution times for each task are stored in the traces as specified in table 7.8. The trace information specifies the instruction counts that was logged using SimpleScalar as described in chapter 6.

Task	File name	column
1	TC1_ape1_0_exect.tra	2
2	TC1_ape2_0_exect.tra	2
3	TC1_ape1_1_exect.tra	2
4	TC1_ape2_1_exect.tra	2

Table 7.8: *Traces specifying the instruction counts for each of the four tasks.*

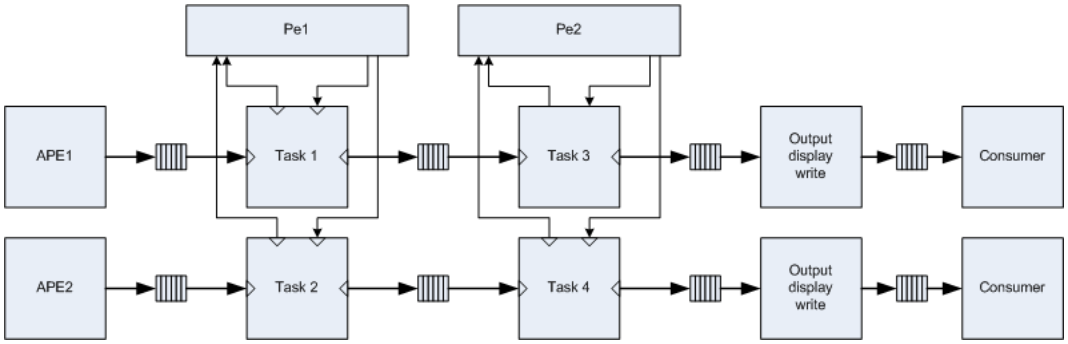

Figure 7.1: *Test bench for Test case 1 - Periodic input*

All the tests simulate 15 seconds of processing, which is equivalent to the length of the videos the execution time traces are based on. This leads to the APE_1 outputting 449 frames and APE_2 outputting 373 frames.

7.4.1.2 TC 1.0 - FP

The priorities are given such that the tasks that process the APE_1 stream are given the highest priority in each of the resources.

Slow processing elements Both figures H.3 and H.4 show the impact of the tasks processing the APE_1 stream having the highest priority. The APE_1 display buffer is seldom beneath the initial buffering level. The reason for the display buffer going beneath the initial buffering is that the system is designed based on the average execution times, and will therefore at times be too slow due to the intra variability. The APE_2 buffer is however often below the initial buffering, and frame losses are also seen once in a while. Figure H.4 shows the time the tokens for each of the streams have spent waiting in the buffers or on gaining access to the processing element. Again it clearly shows the impact of the priority scheduling where tokens for the APE_2 stream wait much longer that the APE_1 stream tokens.

The APE_1 stream lost 0 frames, while the APE_2 lost 17 frames during the simulation.

Fast processing elements This test shows that the priority does not have any impact on the buffer backlog once the processing elements are fast enough, since the processing elements are idle much of the time. Both display buffers are constantly above the initial buffering level as seen in figure H.5. The delays for the stream tokens are considerably smaller compared to the test with the slow processing elements. The delay of the tokens for the APE_2 stream are however still greater than for APE_1, which is as expected. The delays for this test are plotted in figure H.6.

No frame losses was detected during the simulation.

7.4.1.3 TC 1.1 - EDF

The deadlines which are used for the tasks in the EDF resource are equal to the period of the input tokens. The deadlines are given in table 7.9.

Task	Deadline
1	25 μs
2	132 μs
3	25 μs
4	132 μs

Table 7.9: *Deadlines for the tasks in TC1.1 that uses the EDF scheduling scheme.*

Slow processing elements The simulation results are similar to those found using the FP scheduling, where the APE_1 stream is more or less optimal, at the expense of the APE_2 stream. This is caused by the two streams having different periods, where APE_1 has a much smaller period than APE_2. Therefore the initial deadlines are most of the time smaller for APE_1, which leads to APE_1 having the highest priority. Figure H.7 shows the buffer backlog, while figure H.8 shows the delay of the tokens for each of the streams.

The APE_1 stream has no frame losses, while APE_2 has 18 frame losses.

Fast processing elements Again both the buffer backlog and the token delays are very similar to those found for the FP scheduling.

Both of the streams have no frame losses.

7.4.1.4 TC 1.2 - EDF+CBS

The typical parameters for continuous media running on a CBS serving only a single task are respectively setting the bandwidth and period equal to the average execution time and period. The CBS parameters for this test are given in table 7.10. Note that the parameters are the same for both the slow and the fast processing elements, where the average values are based on those from the slow processing elements.

Task	Bandwidth	Period	Utilization
1	15 μs	25 μs	0.60
2	50 μs	133 μs	0.38
3	15 μs	25 μs	0.60
4	50 μs	133 μs	0.38

Table 7.10: *CBS parameters for the CBSs running on the two processing elements. Note that tasks 1 and 2 are assigned P_e1 and tasks 3 and 4 are assigned P_e2. Therefore the total CBS utilization is 0.98 for both the processors.*

Slow processing elements The buffer backlog seen in figure H.11 is somewhat different than for the two previous tests with FP and EDF. The display buffer for the APE_1 stream is not favoured over the APE_2. This is exactly what the property of the CBS scheduling algorithm should do, by providing temporal protection. This is used in order to provide a more fair scheduling, where the tasks only suffer them selves, when they have a high intra variability. The response times shown in figure H.12 are also somewhat different, where we now see how the tokens for the two streams are more or less equal in terms of the delays. The APE_2 stream tokens do however still have a tendency to have a slightly greater delay, but the server periods for this stream are also greater than those for the APE_1 stream.

The frame loss for the APE_1 stream is 8, while it for APE_2 is 15. We see an increase in the total number of frame losses, but this time the losses are shared among the two streams, which is caused by the temporal protection, that the CBS scheduling algorithm provides.

Fast processing elements For the test with the fast processing elements, the buffer backlog and the response time delays are very similar to the two previous tests. This is caused by the tasks completing within the boundaries of the CBS parameters.

Both APE_1 and APE_2 have 0 frame losses.

7.4.1.5 TC 1.3 - Linux

The Linux resources have been initialized with time quanta for the *nice* level 0 of 1 ms, *nice* level 99 of 200 μs and the *nice* level 139 of 5 μs. This leads to the four tasks being assigned the time slots given in table 7.11.

Task	Nice level	Time quantum
1	120	102.5 μs
2	130	53.8 μs
3	120	102.5 μs
4	130	53.8 μs

Table 7.11: *The nice level and the time quanta that are assigned the tasks in the Linux scheduler.*

Slow processing elements Both the buffer backlog and the response time delays plots are very similar to those for the FP scheduling resources. This is caused by the APE_1 stream still having the highest priority, and the time quanta being sufficiently large for most of the tokens to be processed within the given time frame. The backlog is shown in figure H.15, and the response time delays are given in figure H.16.

The APE_1 stream had 0 frame losses, while the APE_2 had 18 frame losses.

Fast processing elements Again the over powered system shows that the scheduling has no problems servicing all the tokens, since the buffer backlog is above the initial buffering level. The backlog is shown in figure H.17, and the response time delays are given in figure H.18.

None of the streams have any frame losses.

7.4.1.6 Summary

Table 7.12 summarizes the results from case study 1. Two tests were performed for each of the scheduling policies. The first system was designed to service the average load of the two continuous media streams. The second test was an over designed system, which simply showed for all the scheduling schemes that the streams were serviced without any frame losses, and with more or less the same response times on each of the processing elements.

The results from the scheduling algorithms FP, EDF and Linux were very similar, both in terms of frame losses and in terms of the tendencies of the buffer backlog and the response time delays. The CBS scheduling algorithm did however show how the temporal protection ensured that streams that have high intra variability did not necessarily steal the processor time from other tasks, which for some reason had a lower priority. This can be seen since the the APE_1 stream also had frame losses.

TC	Speed	Frame Loss (%)		Average Response time	
		APE_1	APE_2	APE_1	APE_2
1.0	Slow	0	4.6	2,478 μs	114,051 μs
FP	Fast	0	0	16 μs	52 μs
1.1	Slow	0	4.8	2,733 μs	131,282 μs
EDF	Fast	0	0	16 μs	53 μs
1.2	Slow	1.8	4	19,130 μs	82,125 μs
CBS	Fast	0	0	16 μs	53 μs
1.3	Slow	0	4.8	2,634 μs	131,585 μs
Linux	Fast	0	0	16 μs	53 μs

Table 7.12: *Summary of the various sub test cases for TC1. APE_1 sends a total of 449 frames, while APE_2 sends 373 frames.*

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	957	960	1,481	1,101
	t1_t3_input	736	759	2141	868
	t3_output	1	1	1	1
	ape1_con	2,649	2,645	3,463	2,649
APE_2	t2_input	1,394	1,392	1,394	1,394
	t2_t4_input	979	1,267	1,046	1,302
	t4_output	1	1	1	1
	ape2_con	1,464	1,297	1,498	1,335

Table 7.13: *Buffer backlog for case study 1 with the slow processors.*

The CBS algorithm did however result in an overall larger number of total frame losses for the system that was designed based on the average resource requirements.

From table 7.13 it is seen how the temporal protection influences the maximum buffer backlog for the APE_1 stream, which in all of the test cases has a tendency to have the highest priority, and therefore lower buffer size requirements.

It is also clear from the maximum buffer backlog in *ape1_ con*, that the tasks processing the APE_1 stream have the highest priority in FP, EDF and the Linux schedulers, since the backlog is close to the initial buffering of approximate 2,640 tokens. While the backlog in the *ape2_ con* is much higher than the initial buffering of approximate 600 tokens.

Table 7.14 shows how the system with the fast processing elements is over designed, since there the buffer backlog is very low. Further the backlog in the two consumer buffers are very close to the initial buffering.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	2	2	2	2
	t1_t3_input	2	2	2	2
	t3_output	1	1	1	1
	ape1_con	2,648	2,648	2,648	2,648
APE_2	t2_input	2	2	2	2
	t2_t4_input	2	2	2	2
	t4_output	1	1	1	1
	ape2_con	603	603	603	603

Table 7.14: *Buffer backlog for case study 1 with the fast processing elements.*

7.4.2 Case study 2 - GUI task

This test bench has four semi periodic tasks and one aperiodic task distributed over two processing elements. The five tasks corresponds to two MPEG decoders and one GUI task. This test case uses the same two semi periodic tasks as described in section 7.3. A fifth task is added to the test bench which processes an aperiodic stream APE_3. The APE_3 stream is not a continuous media stream as opposed to APE_1 and APE_2. The fifth task is assigned to P_e1.

The execution times for each of the aperiodic tasks are based on the following:

APE_1 flwr_080.m2v

APE_2 high_25fps_320x480.m2v

APE_3 GUI (Random periods manually generated)

7.4.2.1 Test bench

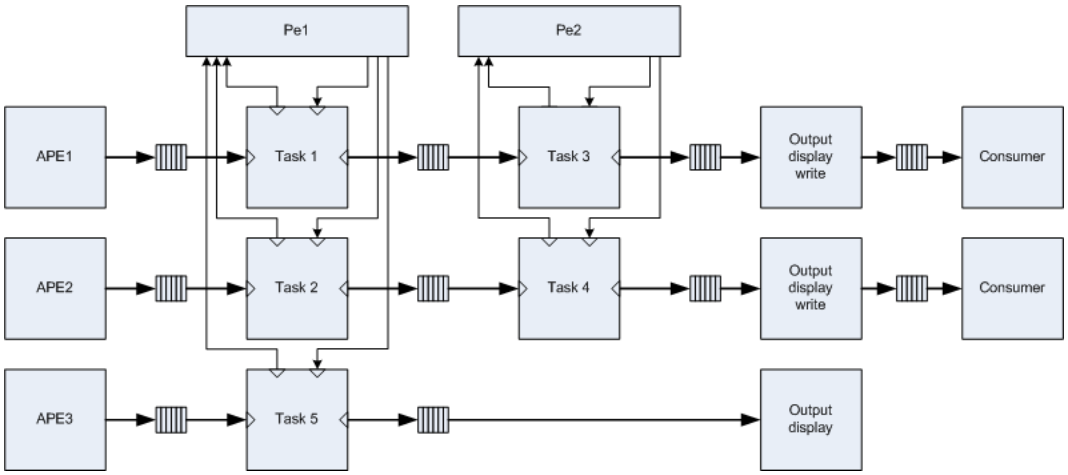

Figure 7.2: *Test bench for Test case 2 - GUI task.*

The token generator trace files are:
APE_1: TC2_ape1_input.tra
APE_2: TC2_ape2_input.tra
APE_3: TC2_ape3_input.tra
The values for APE_1 and APE_2 are stored in picoseconds, while the values for APE_3 are stored in μs.

The execution times for each task are stored in the traces as specified in table 7.15. The trace information specifies the instruction counts measured using SimpleScalar as described in chapter 6.

The instruction counts for tasks 1-4 use the multiplier factors given in table 7.3, while task 5 uses the default multiplier of 1, leading to no scaling.

Task	File name	column
1	TC2_ape1_0_exect.tra	2
2	TC2_ape2_0_exect.tra	2
3	TC2_ape1_1_exect.tra	2
4	TC2_ape2_1_exect.tra	2
5	TC2_ape3_exect.tra	1

Table 7.15: *Traces specifying the instruction counts for each of the five tasks in TC2.*

The APE_3 stream contains bursts of data with high processor cycle counts. The total processor cycle count for APE_3 during the 15 seconds is 216,164,500. On average the processor must be at least 14.4 MHz in order to service the stream within the 15 seconds. Figure H.121 in the appendices shows a histogram of the processor cycles required for each task request. It is apparent that most of the request only require few cycles, while there are some which require up to 40M cycles.

The APE_3 streams performs 58 request during the 15 seconds, which results in an average processor cycle count of 3,726,974. The average execution time of APE_3 within the 15 seconds is then 12,851 μs and the average period is 214,874 μs.

The processor speed of P_e1 is increased from 270 MHz to 290 MHz in order to accommodate task 5, without having an under designed system.

All the tests simulate 15 seconds of processing, which is equivalent to the length of the videos the execution time traces are based on. This leads to the APE_1 outputting 449 frames and APE_2 outputting 373 frames.

The plots of the buffer backlog and the response time delays, which are referred to in this section are found in the appendices H.3 starting at page 146.

7.4.2.2 TC 2.0 - FP

The GUI task (APE_3) is given the highest priority, followed by APE_1 and APE_2 with the lowest priority. The reason for this is that the task is not a continuous media stream, and the task will therefore be idle much of the time. Further the GUI must react instantly in order not to annoy the user, even though it has a negative impact on the playback of the two video streams.

Slow processing elements It is clear from the plot of the buffer backlog in figure H.19 that the high intra variability in task 5 has a negative influence on the tasks processing the two streams APE_1 and APE_2, due to the large fluctuations in the backlog.

The response time delay plot in figure H.20 further show how the task processing the APE_3 stream influence the the tasks processing the streams APE_1 and APE_2. The APE_3 stream does however also have a negative impact on the tokens belonging to it self, since they also have some delays.

The APE_1 stream lost 8 frames, while the APE_2 lost 28 frames.

Fast processing elements The buffer backlog in figure H.21 shows that the system is for the most time sufficiently fast for processing the three streams, but the two display buffers are at times below the initial buffering, which is caused by burst from the APE_3 stream.

The burstiness of APE_3 is more visible in the response time delay plot in figure H.22, where the delay for the most time is low, but there are some peaks where all three streams experience large delays.

The APE_1 stream lost 2 frames, while the APE_2 stream lost 3 frames.

7.4.2.3 TC 2.1 - EDF

The deadlines for the five tasks are based on the periods of the tasks. Note that the period for task five is in ms and not μs as for the four other tasks.

Task	Deadline
1	25 μs
2	132 μs
3	25 μs
4	133 μs
5	215 ms

Table 7.16: *Deadlines for various tasks in test case 2.1 that uses the EDF scheduling scheme.*

Slow processing elements The EDF scheduling algorithm does also create large fluctuations in the buffer backlog, which are caused by the APE_3 stream.

The large deadlines for the APE_3 stream tasks has the impact that the response time delay for the stream is much higher than was the case in the FP scheduling algorithm, since the APE_1 and APE_2 tasks will in most cases have a higher priority.

The APE_1 stream lost 4 frames, while the APE_2 stream lost 16 frames.

Fast processing elements Both the buffer backlog and the response time delay plots are very similar for the EDF and the FP.

2 frames was lost for the APE_1 stream, while the APE_2 stream lost 3 frames.

7.4.2.4 TC 2.2 - EDF+CBS

The total utilization factor for P_e1 is 1, while it for P_e2 is 0.99. The CBS parameters for task 5, were found in case story 4. The CBS parameters that are used for this test case are given in table 7.17.

Parameter	Tasks 1	Tasks 2	Task 3	Task 4	Task 5
Budget	15 μs	49 μs	16 μs	50 μs	12 μs
Period	26 μs	134 μs	26 μs	133 μs	215 μs
Utilization	0.58	0.37	0.62	0.38	0.06

Table 7.17: *This table contains the CBS parameters for each of the tasks that are in the test bench for TC2.2. Note that the utilization factors in the table are rounded down to two decimals and will therefore not sum as specified.*

Slow processing elements The true power of CBS in terms of temporal protection is seen in the plots of the buffer backlog and the response time delays in figures H.27 and H.28, where the backlog in the APE_1 stream display buffer is most of the time at the initial buffering level. The backlog is reduces due to the intra variability of the APE_1 stream itself. Likewise the buffer backlog of the APE_2 stream is more constant compared to the two previous test cases using FP and EDF scheduling algorithms.

The APE_1 stream lost 0 frames, while the APE_2 stream lost 8 frames.

Fast processing elements The CBS algorithm ensures that the task for the APE_3 does not steal processing time from the tasks for the APE_1 and APE_2 streams. This is seen in figure H.29, where the backlog in the display buffers are similar to the initial buffering. Further figure H.30 shows that the response time delays for the APE_1 and APE_2 streams are low, while they are somewhat higher for the APE_3 stream. This is caused by the APE_3 having some tokens that require long computation time, and therefore not able to complete within the budget of the CBS that the APE_3 task is assigned.

7.4.2.5 TC 2.3 - Linux

The Linux resources have been initialized with time quanta for the nice level 0 of 1 ms, nice level 99 of 200 μs and the nice level 139 of 5 μs. This leads to the four tasks being assigned the time slots given in table 7.18.

Task	Nice level	Time quantum
1	120	102.5 μs
2	130	53.8 μs
3	120	102.5 μs
4	130	53.8 μs
5	110	151.3 μs

Table 7.18: *The nice level and the time quanta that are assigned the tasks in the Linux scheduler for case study 2.*

Slow processing elements The buffer backlog in figure H.31 shows how the tasks for the APE_1 stream are serviced sufficiently often even though task 5 has a higher

priority. This is caused by the fairness of the Linux O(1) algorithm. The time slots assigned for the APE_1 stream tasks are 102.5 μs, which is much larger than the average execution time of each token in the stream. The tasks for the APE_2 stream are however only assigned timeslots of 53.8 μs, which does cover the average execution time, but will in many cases lead to the tasks not completing within the time slot. This leads to the APE_1 display buffer having a nice buffer backlog, which most of the time is similar to the initial buffering. The APE_2 stream does however not get sufficient time to execute especially in the beginning and at the end of the simulation.

The response time delay for the APE_2 stream tokens are very high compared to those for the APE_1. This is caused by the time slots being too small for the APE_2 stream tasks, and the assigned priorities.

No frames were lost for the APE_1 stream, while the APE_2 stream lost 27 frames.

Fast processing elements It is apparent from the buffer backlog in figure H.33, that the time slots for the APE_2 stream tasks are sufficiently large in this test since the backlog is similar to the initial buffering.

The response time delays for the APE_1 and APE_2 streams are significantly reduced, which again is caused by the tasks completing within the time slots that they are assigned. The response time delay for the APE_3 stream is also significantly lower, but the time slots are not large enough to accommodate some of the resource demanding request, which lead to high response time delays.

No frames were lost for either of the APE_1 and APE_2 streams.

7.4.2.6 Summary

TC	Speed	Frame Loss (%)		Average Response time		
		APE_1	APE_2	APE_1	APE_2	APE_3
2.0	Slow	1.8	7.5	17,150 μs	769,788 μs	33,996 μs
FP	Fast	0.4	0.7	524 μs	1,164 μs	10,797 μs
2.1	Slow	1.3	0.4	11,203 μs	339,511 μs	1,088,612 μs
EDF	Fast	0.4	0.6	478 μs	1,055 μs	13,872 μs
2.2	Slow	0	2.1	2,275 μs	87,035 μs	1,212,623 μs
CBS	Fast	0	0	16 μs	53 μs	15,946 μs
2.3	Slow	0	7.2	1,903 μs	381,169 μs	230,640 μs
Linux	Fast	0	0	19 μs	58 μs	15,908 μs

Table 7.19: *Summary of the various sub test cases for case study 2. APE_1 consist of 449 frames, while APE_2 consist of 373 frames.*

The EDF scheduling scheme had the least frame loss, closely followed by the CBS algorithm. The response times are however relative high for both APE_2 and APE_3 for the EDF, this is caused by the fact that the deadlines of the tasks processing

these streams are larger than for the tasks processing the APE_1 stream. The response times for both the EDF and CBS scheduling algorithms are however very high compared to both the FP and Linux scheduling.

For the fast processors there are some few frame losses for the FP and EDF scheduling algorithms, while the CBS and Linux scheduling algorithms have no frame losses at all. Further the response times are lowest for the CBS and Linux scheduling, which is caused by the fairness built into each of the two algorithms.

The average response time for the APE_3 stream of 1.2 seconds using the CBS scheduling algorithm seems a bit high. This could be decreased by assigning a greater bandwidth for task five on the cost of less bandwidth for the other tasks. Overall the Linux scheduling algorithm seems to be a fairly good choice for both the slow and the fast case, even though the frame loss is quite high using the slow processing elements. But the low over all response time is a critical factor for the APE_3 stream, and this is fairly low.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	10,318	5,211	864	809
	t1_t3_input	5,833	2,890	680	652
	t3_output	1	1	1	1
	ape1_con	11,476	6,389	2,641	2,642
APE_2	t2_input	5,234	2,439	468	3,879
	t2_t4_input	6,813	3,734	875	3,245
	t4_output	1	1	1	1
	ape2_con	5,534	3,589	851	2,520
APE_3	t5_input	5	19	20	7
	t5_output	1	1	1	1

Table 7.20: *Buffer backlog for case study 2 with the slow processors.*

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	1,842	1,841	9	21
	t1_t3_input	1,441	1,438	5	17
	t3_output	1	1	1	1
	ape1_con	2,828	2,835	2,642	2,642
APE_2	t2_input	432	432	3	6
	t2_t4_input	610	610	2	5
	t4_output	1	1	1	1
	ape2_con	741	741	603	603
APE_3	t5_input	5	5	5	5
	t5_output	1	1	1	1

Table 7.21: *Buffer backlog for case study 2 with the fast processors.*

The buffer requirements are very high for both FP and EDF both using the slow and the fast processing elements. The lowest buffer sizes requirements are seen for

the CBS scheduling algorithm, especially for the slow processing elements, while the requirements for the CBS and Linux scheduling algorithms are more or less the same for the fast processing elements.

7.4.3 Case study 3 - Input with jitter

The purpose of this test case is to have jitter on the input in order to simulate the MPEG videos streamed over a network connection as Ethernet.

The figures containing the buffer backlog and response time delays that are referred to in this section are in the appendices section H.4 and start on page 154.

7.4.3.1 Test bench

The test bench is similar to the one in test case 1, where there are two MPEG video streams running on two processing elements. The only difference is that there is jitter on the input, which is set as approximate ±40% of the period. Table 7.22 summarizes the input trace files used in the following test cases.

Stream	File name	Period	Jitter
APE_1	TC3_ape1_input.tra	25.252 μs	10 μs
APE_2	TC3_ape2_input.tra	133.332 μs	60 μs

Table 7.22: *Input trace file and the period and jitter on the token generators.*

7.4.3.2 TC 3.0 - FP

The tasks processing the APE_1 stream have the highest priorities, and the tasks processing the APE_2 stream have the lowest priorities, similar to case study 1.

Slow processing elements Both the buffer backlog and the response time delay graphs in figures H.35 and H.36 are very similar to the graphs found in TC1. The APE_1 stream is favoured over the APE_2 stream, since the display buffer for the APE_1 stream is most of the time above the initial buffer. Further the response time delays for APE_1 are constantly lower than those for APE_2.

No frames were lost for the APE_1 stream, while the APE_2 stream lost 17 frames.

Fast processing elements The buffer backlog and response time delay graphs in the figures H.37 and H.38 show an over designed system, since the buffer backlog in the display buffers are constantly above the initial buffering, and the backlog in the remaining buffers are negligible. Further the response time delays are very low, and even non existent much of the time for the APE_1 stream.

There are no frame losses for any of the streams.

7.4.3.3 TC 3.1 - EDF

The deadlines used in this test are the same as for case study 1 ad are given in table 7.9.

Slow processing elements The buffer backlog and response time delays given in figures H.39 and H.40 are similar to those found using the FP scheduling algorithm. Again this is caused by the APE_1 tasks have the lowest deadlines, thereby getting the highest priority.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 18 frames.

Fast processing elements The buffer backlog and response time delays again shows an over designed system, where the backlog is above the initial buffering, and the response time delays are low. The buffer backlog and the response time delay graphs are found in figure H.41 and H.42 respectively.

No frames were lost for either of the two streams.

7.4.3.4 TC 3.2 - EDF+CBS

The budget and period for each of the CBSs are the same as used in case study 1, and are found in table 7.10.

Slow processing elements The buffer backlog has very large fluctuations similar to those found in case study 1. These are caused by fairness of the CBS algorithm and therefore the tasks do not always complete within their corresponding CBSs budget. The response time delays for the APE_2 stream are at times also lower than for the APE_1 stream.

The APE_1 stream lost 8 frames, while the APE_2 lost 10 frames, which again shows how the fairness ensures that the streams with high intra variability are punished instead of potentially stealing processing time from another task.

Fast processing elements Again the buffer backlog and response time delays show an over designed system. The delays for the APE_1 stream are however greater than seen in the EDF and FP scheduling algorithms, which is caused by the APE_1 tasks not always completing within the CBS budgets.

There are not frame losses for either of the streams.

7.4.3.5 TC 3.3 - Linux

The priorities and time slots are for this test similar to those in case study 1, and can be found in table 7.11.

Slow processing elements The buffer backlog and response time delays are very similar to those found in case study 1, where the APE_1 stream is favoured over the APE_2 stream, which is caused by the tasks processing the APE_1 stream having the highest priorities.

There are no frame losses for the APE_1 stream, and 17 frame losses for the APE_2 stream.

Fast processing elements Not surprisingly the buffer backlog and the response time delay graphs show an over designed system in figures H.49 and H.50.

No frame losses are registered for either of the streams during the simulation.

7.4.3.6 Summary

The jitter has very little effect on the system in terms of frame losses. Further the buffer backlog and response time graphs are very similar to those found in case study 1, where there was no jitter on the input.

TC	Speed	Frame Loss (%)		Average Response time	
		APE_1	APE_2	APE_1	APE_2
2.0	Slow	0	4.6	2,479 μs	114,064 μs
FP	Fast	0	0	16 μs	52 μs
2.1	Slow	0	4.8	2,713 μs	131,289 μs
EDF	Fast	0	0	16 μs	52 μs
2.2	Slow	1.8	2.7	24,700 μs	67,360 μs
CBS	Fast	0	0	16 μs	52 μs
2.3	Slow	0	4.6	2,881 μs	132,320 μs
Linux	Fast	0	0	16 μs	52 μs

Table 7.23: *Summary of the various sub test cases for case study 3. APE_1 consist of 449 frames, while APE_2 consist of 373 frames.*

Both the frame losses registered in this test and the average response times are very similar to those found in case study 1. This leads to the conclusion that jitter of ±40% only has a slight influence the the performance of the system.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	957	960	1,791	1,101
	t1_t3_input	736	759	2,141	868
	t3_output	1	1	1	1
	ape1_con	2,649	2,646	3,100	2,648
APE_2	t2_input	1,394	1,394	1,394	1,394
	t2_t4_input	980	1,267	944	1392
	t4_output	1	1	1	1
	ape2_con	1,465	1,298	1,397	1,337

Table 7.24: *Buffer backlog for case study 3 with the slow processors.*

Even the maximum buffer backlogs are almost identical to those found in case study 1. The CBS scheduling algorithm has the biggest differences in the size of the buffer backlog of 10% to 20%. This difference is however only for the initial input buffers and the display buffers.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	2	2	2	2
	t1_t3_input	2	2	2	2
	t3_output	1	1	1	1
	ape1_con	2,648	2,648	2,648	2648
APE_2	t2_input	2	2	2	2
	t2_t4_input	2	2	2	2
	t4_output	1	1	1	1
	ape2_con	603	603	603	603

Table 7.25: *Buffer backlog for case study 3 with the fast processors.*

7.4.4 Case study 4 - Varying CBS parameters

This test is based on the set-up from case study 2 with two periodic inputs, which are continuous media streams. Further an aperiodic task, which represents a GUI task. This task has been assigned to the first processing element.

The third aperiodic stream APE_3 needs on average 14.4 MHz of processing power, therefore P_e1 is upgraded to a 290 MHz processor, in order to be able to service the tasks without encountering a buffer overflow.

Task	Parameter	TC4.0	TC4.1	TC4.2	TC4.3	TC4.4
1	Budget	15 μs	30 μs	8 μs	15 ns	15 μs
	Period	26 μs	51 μs	14 μs	26 ns	25 μs
	Utilization	0.58	0.59	0.57	0.58	0.60
2	Budget	49 μs	98 μs	25 μs	49 ns	10 μs
	Period	134 μs	269 μs	68 μs	134 ns	25 μs
	Utilization	0.37	0.37	0.37	0.37	0.40
3	Budget	16 μs	31 μs	8 μs	16 ns	15 μs
	Period	26 μs	50 μs	13 μs	26 ns	25 μs
	Utilization	0.62	0.62	0.62	0.62	0.60
4	Budget	50 μs	99 μs	25 μs	50 ns	10 μs
	Period	133 μs	267 μs	67 μs	133 ns	25 μs
	Utilization	0.38	0.37	0.37	0.38	0.40
5	Budget	12 ms	20 ms	6.5 ms	13 μs	1.5 μs
	Period	215 ms	432 ms	107.5 ms	215 μs	25 μs
	Utilization	0.06	0.05	0.06	0.06	0.06

Table 7.26: *This table contains the various parameters for budget and period, which were tested in order to find an optimal set-up in case study 4. Note that the time units for task 5 differ from those for tasks 1-4. Further note that the time units in TC 4.3 are in ns instead of µs for tasks 1-4, as the other tests. Not all the utilization factors in the table adds up to the total, due to rounding of 2 decimals.*

Processing element	Total Utilization				
	TC4.0	TC4.1	TC4.2	TC4.3	TC4.4
P_e1	1.00	1.00	0.99	1.00	1.00
P_e2	0.99	0.99	0.99	0.99	1.00

Table 7.27: *Total bandwidth utilization factor of the processing elements for each of the test cases.*

Table 7.26 summarizes the CBS parameters, budget and period for each of the test cases. In order for the five tasks to be schedulable, the total utilization factor of the tasks in each test case must be less than or equal to one. Table 7.27 contains the summed utilization factors of the tasks.

TC4.0 In this test the budget and period are set to the average execution time and the average period respectively, as proposed by Buttazzo et al. in [BLAC05].

TC4.1 The budget and the period for each CBS have been doubled compared to TC4.0.

TC4.2 The budget and the period have for each CBS been halved campared to TC4.0.

TC4.3 The budget and the period have been divided by 1000 in each test.

TC4.4 The period for each of the CBSs has been set to 25 μs. The budget for each of the CBSs has been updated accordingly, in order for the total utilization factor to remain less than or equal to one, making the tasks schedulable.

All of the test cases in this section have only been performed using the slow processing elements, since the fast processing elements have only shown a marginal difference in the previous case studies.

7.4.4.1 TC4.0

The CBS parameters budget and server period are in this test set equal to the average execution time and the period of each stream. For the APE_3 stream which is aperiodic the average execution time and period from section 7.4.2 is used.

The buffer backlog and the response time delay plots show that the APE_1 stream tasks have difficulties in the beginning of executing the tasks within the given budget. The APE_1 stream is however back at the initial buffering level after a quarter way through the simulation time. The APE_2 stream tasks are regularly not completing within the deadlines throughout the whole simulation.

The APE_1 stream lost 1 frame, while the APE_1 stream lost 28 frames.

7.4.4.2 TC4.1

The CBS parameters, budget and period, are in this test doubled compared to the average values used in the previous test.

Both the APE_1 and the APE_2 stream tasks are not completing within their deadlines, which leads to frame losses, and some large fluctuations in the buffers. This is seen in figure H.53. The response time delays are however lower for the APE_2 stream, which indicates that there is a greater sharing between the stream tasks.

The APE_1 stream lost 16 frames, and the APE_2 lost 23 frames.

7.4.4.3 TC4.2

In this test the budget and the period of each of the CBSs are halved.

This leads to high fluctuations in the buffer backlog as seen in figure H.55. The APE_1 stream tasks are do however perform better than the APE_2 stream tasks.

3 frames were lost for the APE_1 stream, while the APE_1 stream lost 12 frames.

7.4.4.4 TC4.3

The budget and period have in this test been divided by 1000. This test will see how well the scheduling performs if the CBSs forces the tasks to scheduled and unscheduled often. Note however that the scheduling latency is not simulated, therefore the response times would in practice be higher.

The APE_1 stream tasks are much more efficient, since the backlog for the APE_1 stream is most of the time at the initial buffering level. The APE_2 stream tasks also seem to be meeting more deadlines. The buffer backlog is shown in figure H.57.

The response time delay graph in figure H.58 shows how the APE_1 stream tasks are favoured over the APE_2 stream tasks. The response times for the APE_3 stream are in most cases above both the APE_1 and the APE_2 streams.

As indicated in the buffer backlog graph, the APE_1 and APE_2 streams have experienced a very large improvement, since there are no frame losses for the APE_1 stream, and only 8 frame losses for the APE_2 stream.

7.4.4.5 TC4.4

In this test case the periods are set to 25 μs for all the tasks, and the budgets are updated accordingly.

In this test there are high fluctuations in all the buffers in the test bench. The backlog in the APE_1 display buffer is however much of the time at the initial buffering level. The backlog in the APE_2 display buffer is not as constant as the APE_1 display buffer, but seems to be most of the time large enough for not loosing frames.

The response time delays in figure H.60 shows that the APE_1 stream tasks are still favoured over the APE_2 stream tasks.

The APE_1 stream lost 5 frames, while the APE_2 stream lost 9 frame.

7.4.4.6 Summary

The five tests performed in this case study show that the CBS parameters have a large impact on the performance of the overall system, when the system is under more or less constant load. Using the average execution time and average period of a task respectively as the budget and period of a CBS server, does not necessarily lead to the optimal performance. In these five tests the optimal performance in terms of frame losses was found to be TC 4.3, where the budget and period was divided by a factor of 1000. The frame losses and average response times of the five tasks are given in table 7.28

The maximum buffer backlogs in the tests further showed that dividing the budget and period by a factor also had an improvement in terms of the required buffer sizes

TC	Frame Loss (%)		Average Response time		
	APE_1	APE_2	APE_1	APE_2	APE_3
4.0	0.2	7.5	6,451 μs	166,036 μs	1,076,465 μs
4.1	3.6	6.2	26,144 μs	106,748 μs	1,061,776 μs
4.2	0.7	3.2	12,084 μs	88,249 μs	1,231,334 μs
4.3	0	2.1	2,364 μs	87,342 μs	1,212,623 μs
4.4	1.1	2.4	12,359 μs	78,724 μs	1,211,215 μs

Table 7.28: *Summary of the various sub test cases for case study 4. APE_1 sends a total of 449 frames, while APE_2 sends 373 frames.*

Stream	Buffer name	Buffer backlog				
		TC4.0	TC4.1	TC4.2	TC4.3	TC4.4
APE_1	t1_input	1,214	1,214	1,506	907	1,971
	t1_t3_input	1,404	4,052	1,488	716	1,621
	t3_output	1	1	1	1	1
	ape1_con	2,726	5,260	2,836	2,649	3,081
APE_2	t2_input	4,256	4,313	1,583	458	351
	t2_t4_input	2,948	2,660	924	893	1,126
	t4_output	1	1	1	1	1
	ape2_con	996	2,545	1,107	868	1,424
APE_3	t5_input	20	20	20	20	20
	t5_output	1	1	1	1	1

Table 7.29: *Buffer backlog for case study 4 with the slow processors.*

in the system. The optimal result for the five test was once more found by dividing the budget and period by 1000. The maximum backlog for the buffers in the five tests are summarized in table 7.29.

The results found in this case study does point out that the optimal CBS parameters are not necessarily the average execution time and period of a task, but rather some smaller values. The scheduling latency has however not been simulated, therefore there is a limit of how large a value the parameters can be divided by. Further the parameters in these five test were based on the average execution time and period. This might not be the optimal choice since it can lead to a task receiving a very small utilization factor, as is the case for the APE_3 stream task.

7.5 One processing element

The section contains similar set-ups as in case studies 1-4, except for the system only containing one processing element, where all the tasks are scheduled onto.

The test cases are performed both with a *slow* and a *fast* processing element. The slow processing element has sufficient processing power to accommodate the average execution times, while the fast processing element is fast enough to process the WCET for all the tasks at once.

Processor speed	P_e1
Slow	1.12 GHz
Fast	1.5 GHz

Table 7.30: *Processor speed used in the case studies with one processing element.*

Table 7.31 shows the actual min, avg and max execution times in μs. These times are based on the cycle counts given in table 7.2, the cycle multipliers given in table 7.3 and the processor speeds given in table 7.30.

Task	Slow processor			Fast processor			Period
	min	avg	max	min	avg	max	
1	1.0 μs	3.8 μs	23.0 μs	0.8 μs	2.8 μs	17.2 μs	25 μs
2	2.1 μs	11.9 μs	36.8 μs	1.6 μs	8.9 μs	27.4 μs	133 μs
3	13.5 μs	14.5 μs	14.7 μs	10.1 μs	10.8 μs	11.0 μs	25 μs
4	23.8 μs	25.0 μs	27.9 μs	17.8 μs	18.7 μs	20.8 μs	133 μs

Table 7.31: *Actual execution times given as min, avg and max for both the slow and the fast processor. Note that the period column specifies how often a token is generated, and it is this time that the task should complete within.*

7.5.1 Case study 5 - Periodic input

This case study has four tasks which process two MPEG streams, and are scheduled onto one processing element.

The figures containing the buffer backlog and response time delays are located in the appendices H.6 starting at page 167.

7.5.1.1 Test bench

Figure 7.3 shows the test bench with the two input streams and four tasks, two for each stream. The periods of the streams are the same as in case study 1, see section 7.4.1.1 for more details on the trace files.

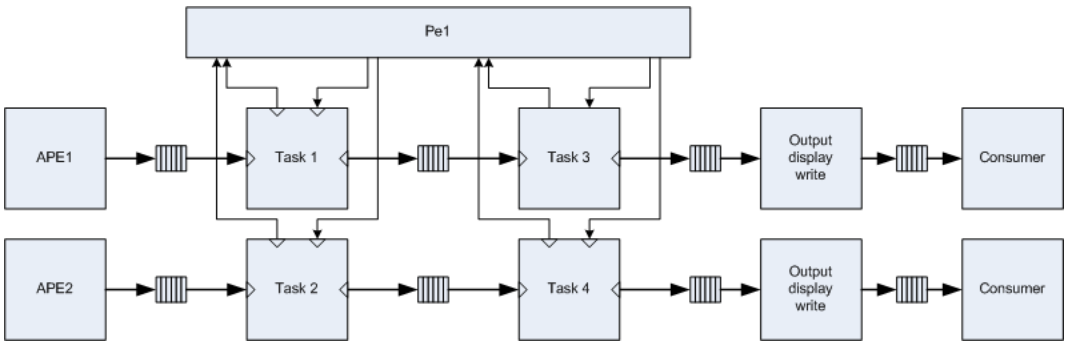

Figure 7.3: *Test bench for TC5, where all the tasks are scheduled onto one processing element.*

Both the input trace files and the execution time trace files from case study 1 are used in this case study.

7.5.1.2 TC5.0 - FP

Table 7.32 shows the priorities, which each of the tasks in the test bench are assigned.

Task	Description	Priority
1	APE_1 VLD/IQ	1
2	APE_2 VLD/IQ	2
3	APE_1 IDCT/MP	3
4	APE_2 IDCT/MP	4

Table 7.32: *Priorities of the four tasks for the Fixed-Priority scheduling algorithm in TC5.0. Priority 1 corresponds to the highest priority.*

Slow processing element The buffer backlog graph in figure H.61 shows that the APE_1 stream tasks are scheduled very well since the backlog in the APE_1 display

buffer is at the initial buffering level. The APE_2 stream does however not perform optimally. It is quite clear that task 4, which has the lowest priority is the cause of this, since the buffer backlog in the intermediate buffer between task 2 and 4 is rather large.

The response time delays in figure H.62 clearly shows how the APE_1 stream tasks are favoured over the APE_2 stream tasks.

No frames were lost for stream APE_1, while stream APE_2 lost 29 frames.

Fast processing element The buffer backlog in figure H.63 shows an over designed system, where all the tasks complete within their deadlines. Figure H.64 does however show that the tokens are delayed, but this is not sufficient to have an impact on the buffer backlog in the display buffers.

No frames are lost for either stream APE_1 or APE_2.

7.5.1.3 TC5.1 - EDF

The deadlines for each of the tasks in this test case are the same as in case study 1, since they are based on the period of the streams. Table 7.9 on page 68 contains the deadlines for each of the tasks in the this test case.

Slow processing element The buffer backlog is very similar to that found using the FP scheduling algorithm, which is caused by the APE_1 stream having a smaller period than the APE_1 stream. This leads to the APE_1 stream tasks having a higher priority most of the time.

The response time delay in figure H.66 also shows how the APE_1 stream is favoured over the APE_2 stream.

The APE_1 stream lost no frames, while the APE_2 stream lost 29 frames.

Fast processing element The buffer backlog and the response time delay graphs show a very efficient system, where the delay for the APE_1 stream is almost non existent.

No frames are lost for either of the streams.

7.5.1.4 TC5.2 - EDF+CBS

The CBS parameters for each of the four CBSs that the tasks are assigned to are given in table 7.33.

Task	Bandwidth	Period	Utilization
1	4 μs	25 μs	0.16
2	12 μs	133 μs	0.09
3	14 μs	25 μs	0.56
4	24 μs	133 μs	0.18

Table 7.33: *CBS parameters for the CBSs running on the processing element. The total CBS utilization factor is 0.99.*

Slow processing element The buffer backlog in figure H.69 shows how the budgets of the APE_1 stream tasks are in many cases too low, resulting in a bad performance. The APE_2 stream tasks do however seem to complete within the deadlines sufficiently often, in order to not suffer a degradation in the QoS for the stream. The initial input buffer for the APE_1 stream has a rather large backlog, which indicates that the problem lies with task 1.

The response time delay graph in figure H.70 also shows how the APE_2 stream is favoured over the APE_1 stream.

The APE_1 stream lost 11 frames, while the APE_2 did not lose any frames.

Fast processing element The buffer backlog in figure H.71 shows that the tasks all complete fast enough for the backlog in the display buffers to stay at the initial buffering.

The response time delay further shows how the fairness ensures that the streams are serviced more or less equally.

No frames are lost for either of the two streams.

7.5.1.5 TC5.3 - Linux

The priorities that each of the tasks are assigned are given in table 7.34. The tasks processing the APE_1 stream are given higher priorities than the other two tasks.

Task	Nice level	Time quantum
1	120	102.5 μs
2	130	53.8 μs
3	121	97.6 μs
4	131	48.9 μs

Table 7.34: *The nice level and the time quanta that are assigned the tasks in the Linux scheduler.*

Slow processing element The buffer backlog in figure H.73 is more or less identical to that found in the FP scheduling. This is caused by time slots being larger

than the actual execution times, thereby ending with a system that basically is using a fixed-priority based approach.

The response time delay graph also shows that the APE_1 stream is favoured over the APE_2 stream.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 29 frames.

Fast processing element The buffer backlog in this test case is constantly above the initial buffering level for both of the streams, as seen in figure H.75. The response time delays seen in figure H.76 shows how both of the APE_1 stream tasks have higher priority than the APE_2 stream tasks, since there is almost no delay for the APE_1 stream.

None of the streams lost any frames.

7.5.1.6 Summary

TC	Speed	Frame Loss (%)		Average Response time	
		APE_1	APE_2	APE_1	APE_2
5.0	Slow	0	7.8	87 μs	78,108 μs
FP	Fast	0	0	15 μs	62 μs
5.1	Slow	0	7.8	159 μs	77,734 μs
EDF	Fast	0	0	14 μs	62 μs
5.2	Slow	2.4	0	20,311 μs	43 μs
CBS	Fast	0	0	20 μs	30 μs
5.3	Slow	0	7.8	19 μs	78,108 μs
Linux	Fast	0	0	14 μs	62 μs

Table 7.35: *Summary of the various sub test cases for case study 5. APE_1 consist of 449 frames, while APE_2 consist of 373 frames.*

The CBS scheduling algorithm is the most efficient algorithm in this test case, both in terms of frame loss and the total average response time. The results from FP, EDF and Linux are very similar due to the APE_1 stream having a shorter period than the APE_2 stream, which leads to the APE_1 stream tasks having shorter deadlines, and therefore in most cases having a higher priority.

The fairness of the Linux scheduler did not show, since all the tasks were completed well within the time slots. Only the CBS scheduler could provide the needed fairness.

The CBS parameters were not optimal for the slow processing element, since the buffer backlog was very high in the initial input buffer of the APE_1 stream. The buffer backlogs for the slow processing element is shown in figure 7.36.

The maximum buffer backlogs for the fast processing element are shown in table 7.37. There is only a minimal difference between the four scheduling algorithms.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	1	3	2,138	1
	t1_t3_input	133	135	1	7
	t3_output	1	1	1	1
	ape1_con	2,648	2,646	2,642	2,648
APE_2	t2_input	1	2	1	184
	t2_t4_input	1,537	1,533	1	1,537
	t4_output	1	1	1	1
	ape2_con	1,008	1,008	603	1,008

Table 7.36: *Buffer backlog for case study 5 with the slow processors.*

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	1	1	4	1
	t1_t3_input	3	1	1	2
	t3_output	1	1	1	1
	ape1_con	2,648	2,648	2,648	2,648
APE_2	t2_input	1	2	1	2
	t2_t4_input	3	2	1	3
	t4_output	1	1	1	1
	ape2_con	603	603	603	603

Table 7.37: *Maximum buffer backlog for case study 5 with the fast processors.*

7.5.2 Case study 6 - GUI task

7.5.2.1 Test bench

In section 7.4.2 the minimum processor speed for the APE_3 stream was found to be 14.4 MHz. The processor speed, for the slow processor test is therefore increased to 1.14 GHz in order to avoid buffer overflows.

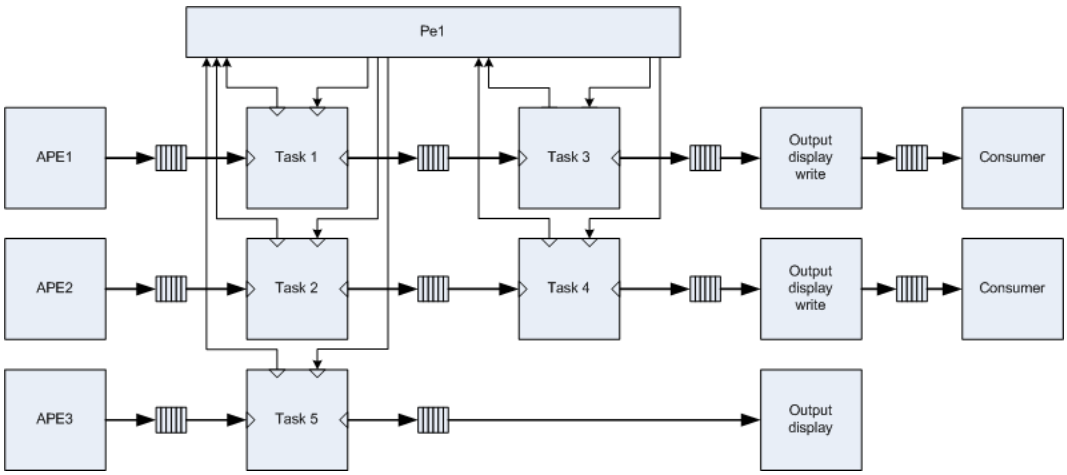

Figure 7.4: *Test bench for case study 6, where the GUI task has been added. All five tasks are scheduled onto one processing element.*

The input trace files and the execution time trace files are the same as in case study 2, see section 7.4.2.1 for more details on the trace files.

The buffer backlog and response time delay graphs referred to in the section are located in the appendices H.7 starting at page 175.

7.5.2.2 TC6.0 - FP

Table 7.38 shows the priorities, which each of the tasks in the test bench are assigned.

Task	Description	Priority
1	APE_1 VLD/IQ	2
2	APE_2 VLD/IQ	3
3	APE_1 IDCT/MP	4
4	APE_2 IDCT/MP	5
5	APE_3 GUI	1

Table 7.38: *Priorities of the five tasks for the Fixed-Priority scheduling algorithm in TC6.0. Priority 1 corresponds to the highest priority.*

Slow processing element From the buffer backlog in figure H.77 we see how task 5 that is processing the APE_3 stream is preempting the other tasks processing

the streams APE_1 and APE_2. The APE_3 stream has some very processing heavy request once in a while, which have a clear impact on the two other stream. The APE_2 stream does however suffer the most, since the tasks processing this stream have lower priorities.

The response time delay graph in figure H.78 shows how the APE_3 stream task has the highest priority, and therefore have fairly low delays.

The APE_1 stream lost 3 frames, while the APE_2 stream lost 9 frames.

Fast processing element The buffer backlog graph in figure H.79 shows a system that is capable of servicing the APE_1 and APE_2 stream sufficiently even for the WCET. The processing heavy request from the APE_3 stream does however have a negative impact on the APE_1 and APE_2 stream.

The response time delays in figure H.80 clearly shows how the APE_3 stream has the highest priority, followed by the APE_1, while the APE_2 stream has the lowest priority.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 3 frames.

7.5.2.3 TC6.1 - EDF

The deadlines for each of the tasks are set corresponding to their periods as in case study 2. Table 7.16 on page 75 contains the deadlines.

Slow processing element The buffer backlog in figure H.81 shows the effect of the fifth task not meeting the deadlines, since the display buffers for both the APE_1 and the APE_2 streams at times only are read from. This is caused by the implementation of the EDF scheduler, where the deadline of a task is set to 0 seconds, thereby giving it exclusive rights to the processor until it terminates.

The response time delay in figure H.86 shows how the deadlines of the APE_1 stream tasks are shorter, and therefore the scheduler favours the APE_1 stream tasks.

The APE_1 stream lost 1 frame, while the APE_2 stream lost 6 frames.

Fast processing element The buffer backlog and the response time delays in the figures H.83 and H.84 respectively, show a system that is not fully capable of servicing the large variability in the APE_3 stream. But it does however regain fast. The response time delays for the APE_1 are very low, and mostly above 0 when the APE_3 stream sends the processing heavy tokens.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 2 frames.

Parameter	Tasks 1	Tasks 2	Task 3	Task 4	Task 5
Budget	4 μs	12 μs	15 μs	25 μs	33 ms
Period	26 μs	135 μs	27 μs	135 μs	215 ms
Utilization	0.15	0.09	0.56	0.19	0.02

Table 7.39: *This table contains the CBS parameters for each of the tasks that are in the test bench. Note that the utilization factors in the table are rounded down to two decimals.*

7.5.2.4 TC6.2 - EDF+CBS

The total utilization factor is 1.14 for the five tasks. The CBS parameters were found in case study 8.

Slow processing element From the buffer backlog in figure H.85 we see that the CBS parameters are sufficient in order to ensure that the APE_2 stream tasks are serviced perfectly. The APE_1 stream tasks do however have some problems when the APE_3 has some processing intensive requests. This is both caused by the APE_3 task having a rather large budget, and the APE_1 having a small budget.

The response time delay graph in figure H.86 also shows how the APE_2 stream tasks are serviced quite well, while the APE_1 stream tasks are delayed quite often.

The APE_1 stream lost 5 frames, while the APE_2 did not lose any frames.

Fast processing element The buffer backlog in figure H.87 shows how both the APE_1 and APE_2 stream tasks are serviced very well. The backlog in the APE_1 display buffer seldom falls below the initial buffering.

The response time delay graph in figure H.88 shows how the temporal protection ensures that the APE_1 and APE_2 stream tasks are serviced more or less equally, which is caused by the tasks completing within the given budgets.

Neither the APE_1 or APE_2 streams lost any frames.

7.5.2.5 TC6.3 - Linux

The task priorities and time slots are the same in this test case as in TC2.3. Table 7.18 on page 76 lists the priority and resulting time quanta for the five tasks.

Slow processing element The buffer backlog in figure H.89 shows a very nice scheduling, where the APE_1 stream tasks are favoured, but the APE_2 stream tasks are also serviced sufficiently often.

Neither stream APE_1 or APE_2 lost any frames.

Fast processing element The buffer backlog in figure H.91 shows that the four tasks processing the APE_1 and APE_2 streams are serviced perfectly. The response time delays in figure H.92 shows that the APE_1 stream tasks have the highest priorities, and are in most cases serviced instantly at arrival.

7.5.2.6 Summary

TC	Speed	Frame Loss (%)		Average Response time		
		APE_1	APE_2	APE_1	APE_2	APE_3
6.0	Slow	0.7	2.4	1,485 μs	142,923 μs	8,053 μs
FP	Fast	0	0.8	304 μs	1,312 μs	6,030 μs
6.1	Slow	0.2	1.3	1,041 μs	46,380 μs	1,828,211 μs
EDF	Fast	0	0.5	281 μs	924 μs	18,076 μs
6.2	Slow	1.1	0	37,797 μs	42 μs	15,771 μs
CBS	Fast	0	0	309 μs	30 μs	12,033 μs
6.3	Slow	0	0	21 μs	8,037 μs	1,996,082 μs
Linux	Fast	0	0	15 μs	70 μs	25,316 μs

Table 7.40: *Summary of the various sub test cases for case study 6. APE_1 consist of 449 frames, while APE_2 consist of 373 frames.*

The CBS and the Linux schedulers are the most promising schedulers in this test case, when looking at the tests with the slow processing element. The CBS scheduling has a 1.1 % loss of frames, but with very low response time for all the three streams. The Linux scheduler had no frame losses at all, but the response time for the APE_3 task i very high, at nearly 2 seconds.

Tables 7.41 and 7.42 shows that the Linux scheduling requires significantly smaller buffers than the other three scheduling techniques. Two second response time for the APE_3 stream does however seem somewhat long compared to the 16 ms that the CBS scheduling algorithm can deliver. So the final choice is a trade off between chip area and responsiveness of a task. Changing the priorities in the Linux scheduler might reduce the response time and make this even more attractive.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	1,390	1,377	3,648	16
	t1_t3_input	2,733	1,800	1	18
	t3_output	1	1	1	1
	ape1_con	3,971	3,025	4,934	2,641
APE_2	t2_input	312	305	1	148
	t2_t4_input	2,632	1,410	1	273
	t4_output	1	1	1	1
	ape2_con	2,874	1,651	603	603
APE_3	t5_input	5	5	5	25
	t5_output	1	1	1	1

Table 7.41: *Buffer backlog for case study 6 with the slow processors.*

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	1,056	1,055	1,338	14
	t1_t3_input	1,305	1,301	1	15
	t3_output	1	1	1	1
	ape1_con	2,641	2,641	2,641	2,641
APE_2	t2_input	230	231	1	6
	t2_t4_input	846	495	1	7
	t4_output	1	1	1	1
	ape2_con	973	741	603	603
APE_3	t5_input	5	5	5	5
	t5_output	1	1	1	1

Table 7.42: *Buffer backlog for case study 6 with the fast processors.*

7.5.3 Case study 7 - Input with jitter

The test bench from case study 5 is reused, but in this test there is jitter on the input token generators. Table 7.22 from case study 3 contain the parameters for the two input generators.

7.5.3.1 TC7.0 - FP

The four tasks have the same priorities as in TC5.0, and are given in table 7.32 on page 89.

Slow processing element The buffer backlog in figure H.93 shows how task 1, which is processing the APE_1 stream, has the highest priority since the APE_1 stream display buffer for the most time at the initial buffering level. The buffer backlog in the APE_2 stream display buffer fluctuates since the tasks processing this stream have lower priorities.

From the response time delay graph in figure H.94 we see how the APE_1 stream tasks have higher priorities and therefore have lower response time delays.

Neither of the two streams lost any frames.

Fast processing element The buffer backlog in figure H.95 shows that the tasks are serviced sufficiently often, since the backlog is never beneath the initial buffering level.

The response time delay again shows that the APE_1 stream tasks are favoured over the APE_2 stream tasks, since the response time delay for the APE_1 stream is lowest.

No frames were lost in either of the two streams.

7.5.3.2 TC7.1 - EDF

The tasks have the same deadlines as in TC1.1 and are given in table 7.9.

Slow processing element The buffer backlog in figure H.97 shows that the APE_1 stream tasks are favoured over the APE_2 stream tasks. The APE_2 are even serviced worse than when using the FP scheduling algorithm. This is caused by the two APE_1 stream tasks having the highest priorities, since they have the shortest initial deadlines, while the two tasks processing the APE_2 have the lowest priorities. This is not the case for the FP scheduling algorithm where the first tasks processing the two streams had higher priorities than the second tasks processing the streams.

The response time delay graph in figure H.98 also shows that the APE_1 stream tasks have higher priorities than the APE_2 stream tasks.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 29 frames.

Fast processing element The buffer backlog in figure H.99 again simply shows that the tasks are serviced often enough to maintain the backlog in the display buffers at the initial buffering level.

It is clear from the response time delay graph in figure H.100 that the two tasks processing the APE_1 stream are favoured over the APE_2 stream tasks, since the delay is much lower than for the APE_2 stream. There is even no delay at all for some of the tokens in the APE_1 stream.

No frames were lost for either of the streams.

7.5.3.3 TC7.2 - EDF+CBS

The CBS parameters in this test are based on the average execution time and the period of the tasks. Table 7.33 from TC5.2 contains the parameters.

Slow processing element From the buffer backlog in figure H.101 it is clear that the CBS parameters for the APE_1 do not cover the high intra variability in the stream, while the CBS parameters for the APE_2 are sufficient to keep the backlog in the display buffer at the initial buffering level. The graph further indicate that the problem is concerned with task 1, since the backlog in the input buffer for the APE_1 stream fluctuates quite at bit.

The response time delay graph in figure H.102 also shows how the delay for the APE_1 stream is significantly larger than for the APE_2 stream, which is caused by the temporal protection that is provided by the CBSs.

None of the streams lost any frames, even though the APE_1 stream was close.

Fast processing element The buffer backlog in figure H.103 shows how the CBS parameters are sufficient for the four tasks to complete, since the backlog is constantly above the initial buffering level.

The response time delay in figure H.104 shows how the temporal protection ensures that the two tasks are more equal, since the APE_1 stream also has low delays.

None of the streams lost any frames.

7.5.3.4 TC7.3 - Linux

The tasks priorities in this test case are the same as in TC5.3 and are given in table 7.34 on page 91.

Slow processing element The buffer backlog in figure H.105 is very similar to that found when using the EDF scheduling algorithm. This is caused by the time slots being much larger than the actual processing time, therefore the fairness in the algorithm is not seen. Further the two tasks processing the APE_1 stream have the highest priorities, as was the case in the EDF algorithm.

The response time delay graph is also very similar to the one found using the EDF algorithm.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 29 frames, which is the same as for the EDF scheduling algorithm.

Fast processing element Again both the buffer backlog and the response time delay graphs in figures H.107 and H.108 are similar to those found using the EDF scheduling algorithm. The APE_1 stream tasks are favoured over the APE_2 stream tasks, which is caused by the APE_1 stream tasks having the highest priorities.

No frame were lost for either of the two streams.

7.5.3.5 Summary

TC	Speed	Frame Loss (%)		Average Response time	
		APE_1	APE_2	APE_1	APE_2
7.0	Slow	0	0	133 μs	13,487 μs
FP	Fast	0	0	15 μs	62 μs
7.1	Slow	0	7.8	160 μs	77,731 μs
EDF	Fast	0	0	14 μs	62 μs
7.2	Slow	2.4	0	20,311 μs	43 μs
CBS	Fast	0	0	20 μs	30 μs
7.3	Slow	0	7.8	20 μs	78,104 μs
Linux	Fast	0	0	14 μs	62 μs

Table 7.43: *Summary of the various sub test cases for case study 7. APE_1 consist of 449 frames, while APE_2 consist of 373 frames.*

Of the four tests the FP has actually performed best in this case study when using the slow processing element. But the CBS might perform better if the budget and period are altered. Further the Linux scheduling algorithm will probably also perform equally well as the FP, if the priorities of the tasks are set in the same way as in the FP scheduling algorithm.

It is quite clear how the EDF and Linux scheduling algorithms have very similar results, which is caused by the deadlines in the EDF prioritising the APE_1 stream tasks higher than the APE_2 stream tasks. This then results in the tasks being prioritised in more or less the same way, except that the Linux always prioritises task 3 over task 1, where the EDF is more random. Further the APE_2 stream tasks will also have higher priorities using the EDF, once they get near their deadlines.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	2	3	2,138	2
	t1_t3_input	180	135	1	7
	t3_output	1	1	1	1
	ape1_con	2,649	2,646	2,642	2,649
APE_2	t2_input	1	2	2	184
	t2_t4_input	350	1,533	1	1,537
	t4_output	1	1	1	1
	ape2_con	603	1,009	604	1,009

Table 7.44: *Buffer backlog for case study 7 with the slow processors.*

When the fast processing element is used there is no mentionable difference between the four scheduling algorithms.

When evaluating the requirements of the buffer sizes the FP is again preferred, when using the slow processing element, while the requirements are more or less the same when using the fast processing element.

Stream	Buffer name	Buffer backlog			
		FP	EDF	CBS	Linux
APE_1	t1_input	1	2	4	1
	t1_t3_input	3	2	1	2
	t3_output	1	1	1	1
	ape1_con	2,648	2,648	2,648	2,648
APE_2	t2_input	1	2	1	2
	t2_t4_input	7	2	1	4
	t4_output	1	1	1	1
	ape2_con	603	603	603	603

Table 7.45: *Buffer backlog for case study 7 with the fast processors.*

7.5.4 Case study 8 - Varying CBS parameters

The six tests in this case study have only been performed using the slow processing element, since there is very little or no difference, when using the fast processing element. This test is performed using the test bench in figure 7.4 from case study 6. The speed of the processing element is therefore increased to 1.14 GHz, in order to accommodate the third stream.

The average processor cycle count for the 58 task request during the 15 seconds period is 3,726,974. With the 1.14 GHz processor this will result in an average execution time of 3,269 μs. The CBS parameters which are tested in this case study are based on the average execution time and the average period of the stream APE_3.

Task	Parameter	TC8.0	TC8.1	TC8.2	TC8.3	TC8.4	TC8.5
1	Budget	4 μs	8 μs	2 μs	4 ns	4 μs	4 μs
	Period	26 μs	52 μs	13 μs	26 ns	26 μs	26 μs
	Utilization	0.15	0.15	0.15	0.15	0.15	0.15
2	Budget	12 μs	24 μs	6 μs	12 ns	2.3 μs	12 μs
	Period	135 μs	270 μs	67.5 μs	135 ns	26 μs	135 μs
	Utilization	0.09	0.09	0.09	0.09	0.09	0.09
3	Budget	15 μs	30 μs	7.5 μs	15 ns	15 μs	15 μs
	Period	27 μs	54 μs	13.5 μs	27 ns	27 μs	27 μs
	Utilization	0.56	0.56	0.56	0.56	0.56	0.56
4	Budget	25 μs	50 μs	12.5 μs	25 ns	4.8 μs	25 μs
	Period	135 μs	270 μs	67.5 μs	135 ns	26 μs	135 μs
	Utilization	0.19	0.19	0.19	0.19	0.19	0.19
5	Budget	3.3 ms	6.6 ms	1.65 ms	3.3 μs	399 ns	33 ms
	Period	215 ms	430 ms	107.5 ms	215 μs	26 μs	215 ms
	Utilization	0.02	0.02	0.02	0.02	0.02	0.15

Table 7.46: *This table contains the various parameters for budget and period, which were tested in order to find an optimal set-up in case study 8. Note that the time units for task 5 differ from those for tasks 1-4. Further note that the time units in TC 8.3 are in ns instead of µs for tasks 1-4, as the other tests. Not all the utilization factors in the table add up to the total, due to rounding, with only 2 decimals.*

The total utilization factors for all the tasks in each of the sub tests 0-4 are 1.00, thus making the system schedulable. The utilization factor for TC8.5 is however 1.14, which technically makes the system unschedulable, but since all tasks are soft real-time tasks and the APE_3 is not a continuous media stream, this is acceptable.

7.5.4.1 TC8.0

The CBS parameters, budget and period, in this test correspond to the average execution times and the average periods of the tasks.

The buffer backlog in figure H.109 shows how the CBS parameters are sufficient for the APE_2 stream, while the intra variability in the APE_1 as usual has a negative effect, which reduces the performance of the tasks processing the stream.

The response time delay in figure H.110 clearly shows how the parameters for the APE_2 stream tasks are acceptable, while the APE_1 stream has large delays. The APE_3 also has some high delays, which is caused by the low utilization factor for the task that is processing the APE_3 stream.

The APE_1 stream lost 9 frames, while the APE_2 did not lose any frames.

7.5.4.2 TC8.1

Doubling the budget and period of the CBSs did not have any positive effect on the buffer backlog for the APE_1 display buffer. This is caused by the utilization factor still being the same, and the increased period simply makes the APE_1 stream tasks wait for a longer time before they are granted access to the processing element again.

The response time delays are better for the APE_3 stream, while they are slightly worse for the APE_1 stream, compared to TC8.0.

The frame loss for the APE_1 stream is 13, while the APE_2 once again did not lose any frames.

7.5.4.3 TC8.2

Halving the budget and period of the CBSs has a positive impact on the buffer backlog in the APE_1 display buffer, while it has a negative effect on the buffer backlog in the APE_2 display buffer.

This is also seen in the response time delay in figure H.114, where the delays are low for the APE_1 stream, while they are larger for the APE_2 stream.

The APE_1 stream did not lose any frames, while the APE_2 stream lost 29 frames.

7.5.4.4 TC8.3

In this test the budget and period of the CBSs are divided by 1000, which should make the scheduler reschedule the tasks much more often.

The rescheduling of the task causes high fluctuations in the buffers as seen in figure H.115. The scheduler does however have a positive effect, since it provides fairness thereby giving all the tasks access to the processing element more often.

This is also seen in the response time delays in figure H.116. The delays are however on average higher than before, which is caused by the fairness, where the tasks have to wait quite often for the other tasks also gaining access to the processing element.

None of the streams lost any frames.

7.5.4.5 TC8.4

All five tasks have the same period, namely 26 μs. The budgets of each of the tasks have then been scaled in order for the utilization factors to remain the same.

As we have seen in most of the previous tests, the APE_1 stream still suffers from task 1 having too small an utilization factor, which then makes it unstable. The APE_2 has a utilisation factor which is sufficient, and therefore has a constant buffer backlog at the initial buffering level. The buffer backlog graph is given in figure H.117.

The response time delay graph in figure H.118 also shows how the APE_2 stream is favoured over the APE_1 stream.

The APE_1 stream lost 10 frames, while the APE_2 stream did not lose any frames.

7.5.4.6 TC8.5

This is a special case, where the sum of the utilisation factors for the five CBSs is 1.14 which is above 1, and the tasks will therefore not be schedulable according to the schedulability analysis. The tasks are aperiodic soft real-time tasks and will therefore not always spend all of the budget before terminating. With this reasoning we accept that the summed utilization factor is above 1.

The tasks processing the APE_1 and APE_2 streams all use the same CBS parameters as in TC8.0, which correspond to the average execution time and period. The fifth task, which is processing the APE_3 stream is given a bandwidth which is a factor of 10 larger than the average execution time. The period of the CBS remains as the average period of the APE_3 stream. The purpose is to lessen the response time delay of the APE_3 stream, without lessening the budget and period of the other tasks.

The buffer backlog in figure H.119 shows how the APE_1 tasks are not serviced regularly, due to the APE_3 stream. The APE_2 stream is however relative unaffected, which probably is caused by the stream having a larger period, and therefore more tolerable to the bursts from the APE_3 stream.

The higher bandwidth for the APE_3 stream has further resulted in the stream having a considerably lower response time delay than in the previous tests. The response time delay graph is shown in figure H.120.

The APE_1 stream lost 5 frames, while the APE_2 did not lose any frames.

7.5.4.7 Summary

The six tests performed in this case study produce very different results. The best result from the tests is a subjective matter, where the user either must choose to

prefer the least amount of frames losses or low response time of the tasks. The TC8.3 produced no frame losses, but this was at the expense of the APE_3 stream, which had an average response time of 2 seconds, which is very high.

The TC8.5 seems like the over all winner, since it has a very low amount of frame losses, and the average response time for the APE_3 is also very low. A static scheduling analysis would however conclude that the system is unschedulable. But we accept this based on the fact that the APE_3 stream is highly aperiodic, and that all of the five tasks are soft real-time tasks.

TC	Frame Loss (%)		Average Response time		
	APE_1	APE_2	APE_1	APE_2	APE_3
8.0	2.0	0	19,643 μs	42 μs	1,126,809 μs
8.1	2.9	0	36,249 μs	184 μs	70,675 μs
8.2	0	7.8	35 μs	124,573 μs	230,709 μs
8.3	0	0	1,128 μs	2,905 μs	2,014,329 μs
8.4	2.2	0	16,089 μs	41 μs	1,498,562 μs
8.5	1.1	0	37,797 μs	42 μs	15,771 μs

Table 7.47: *Summary of the various sub test cases for case study 8. APE_1 sends a total of 449 frames, while APE_2 sends 373 frames.*

In terms of the buffer requirements TC8.3 requires the least buffer sizes, while TC8.5 is among the set-ups which require the largest buffers. The TC8.5 is however seen as the test which has the best performance.

Stream	Buffer name	Buffer backlog					
		TC8.0	TC8.1	TC8.2	TC8.3	TC8.4	TC8.5
APE_1	t1_input	3,275	3,275	228	362	2,912	3,648
	t1_t3_input	1	1	5	274	1	1
	t3_output	1	1	1	1	1	1
	ape1_con	3,252	4,253	2,641	2,641	2,890	4,934
APE_2	t2_input	1	264	972	147	1	1
	t2_t4_input	1	291	2,261	227	1	1
	t4_output	1	1	1	1	1	1
	ape2_con	603	604	972	603	603	603
APE_3	t5_input	20	5	7	25	23	5
	t5_output	1	1	1	1	1	1

Table 7.48: *Buffer backlog for case study 8 with the slow processor.*

7.6 Conclusion

The general conclusion from the case studies is that the CBS is an efficient scheduling algorithm for soft real-time systems, when the system just meets the average resource requirements. This is caused by the fairness, which the CBS provides due to the

Case Study	Response time		Buffer	
	Slow	Fast	Slow	Fast
1	CBS	-	FP	-
2	CBS	CBS	CBS	CBS
3	CBS	-	FP	-
5	CBS	CBS	FP/EDF	-
6	CBS	CBS	CBS	Linux
7	FP	CBS	FP	-

Table 7.49: *This table shows the preferred scheduling algorithm from each of the case studies, except for case study 4 and 8. The cases where there is no name in the cell simply means that there is very little difference.*

temporal protection characteristics. The fairness of the Linux O(1) scheduler was not really experienced, since the task execution times in most cases, are below the window sizes that they were assigned.

Table 7.49 shows the scheduling techniques which performed best for each of the test cases. This is based both on the *frame losses*, *buffer backlog* and *response times* of the streams. Most of the test cases with the fast processing elements did not show any notable difference in the QoS, and either of the four scheduling schemes could be used. The buffer size requirements did however in many cases, for the fast elements, favour one scheduling scheme over the others.

Case studies 4 and 8 did not show any definitive answer to finding the optimal values for the CBS parameters budget and period. Case study 4 resulted with the best parameters in TC4.3, where the budget and period are 1000 times shorter than the average execution time and period for each task. Case study 8 did on the other hand show that the best values for the parameters are setting the budget 10 times longer than the average execution time, while the period was unchanged.

The CBS parameters in case studies 1, 3, 5 and 6 all use the standard parameters, where the budget and period are respectively assigned as the average execution time and the average period of the tasks. The results for the CBS test cases in these case studies are likely to be further improved with more optimal values for the CBS parameters.

Chapter 8

Conclusion

This thesis presented two types of variability for multimedia in terms of execution time. These are intra and inter variability, where intra variability is within each stream, while inter variability is across different streams. The inter variability is greatest for media streams with different characteristics in terms of resolution and frame rate.

The DIstributed Multimedia Application Simulator (DiMAS) was implemented, which is a C++ library. DiMAS is used for creating a design level simulator, which models a given multimedia system. This could for example be a set-top box or a DVD player. The simulator can be used when designing new multimedia appliances, where the resources must be kept low, in order to limit cost or power consumption. The costly and time consuming prototyping process can therefore be postponed or completely removed in the design phase.

The DiMAS library is able to model processing elements with four different types of scheduling techniques. These four techniques are *Fixed-Priority*, *Earliest Deadline First*, *Constant Bandwidth Server* and the *Linux O(1)* schedulers. The DiMAS library is an extension of the Pesimdes library, which is a simulation framework for hard real-time systems. The *Fixed-Priority* and the *Earliest Deadline First* scheduling techniques were already part of the Pesimdes library. The *Constant Bandwidth Server* and the *Linux O(1)* schedulers are new implementations.

The variability in terms of execution time for a multimedia stream is modelled using a trace file, which either contains the processor cycles or the actual execution times given in a user specified time unit.

The MPEG video stream was chosen as the main multimedia stream to be used in the examples and discussions throughout the thesis. This was partly caused by the obvious partitioning of tasks in the decoding algorithm, and partly by video having a strong association towards multimedia.

Eight case studies were created to show some examples of how the DiMAS library can be used. The examples are kept simple with two multimedia streams, and in some cases with an additional stream, which is highly aperiodic. The case studies either use test benches with one and two processing elements.

The case studies showed a great difference between the four scheduling algorithms. This was especially visible when the processing elements were designed based on the average execution times of the tasks. The over designed test benches did however result in very similar results for each of the four scheduling algorithms. In general the Constant Bandwidth Server scheduling algorithm performed in most cases better than the other three. The performance was evaluated based both on frame losses and the average response times. The temporal protection in the CBS scheduling algorithm ensured that all tasks were given a fair share of the processing element. The Linux O(1) scheduling algorithm was in most cases not able to provide the fairness it should, since the tasks finished within the assigned time slots.

Case studies 4 and 8 showed that the parameters *budget* and *period* of the CBSs have a big influence on the performance. And simply using the average values of the execution time and period for each task is not necessarily the best choice. The two tests ended with one performing best with very low budget and period, while the the other performed best with a high value for the budget while the value for the period was unchanged. There is from these tests no way of saying how the optimal parameters are determined without performing many simulations.

8.1 Future work

An obvious extension of the simulation library, would be adding new scheduling algorithms. The modularity of the library makes this a fairly easy task.

Modelling a multimedia system can be very complicated using DiMAS, if there are many tasks in the system. An abstraction from the C-code would therefore be preferable, either using a graphical user interface with drag and drop functionality, or using a description file for example in xml format.

The current version of the simulator library is able to model the variable execution time using trace files, where each line in the file correspond to a task request. It would however be preferable to also have the possibility of modelling the variability based on a probability distribution.

Currently network connections are modelled at the input generators. This could however also be modelled in the test bench, in order to be able to model a full distributed multimedia system, where amplifier, dvd, speakers, television and speakers all communicate using a wireless connection.

Bibliography

[ZL02] Vladimir D. Živković, Paul Lieverse: *An Overview of Methodologies and Tools in the Field of System-Level Design*, 2002.

[SSTutorV4] SimpleScalar LLC: *Simple Tutorial v4*, Cited: 25-6-08, `http://www.simplescalar.com/docs/simple_tutorial_v4.pdf`.

[SShome] SimpleScalar LLC: *SimpleScalar homepage*, `http://www.simplescalar.com/`

[MY07] Matt T. Yourst: *PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator*, 2007, `http://www.ptlsim.org/papers/PTLsim-ISPASS-2007.pdf`.

[ALE02] Todd Austin, Eric Larson, Dan Ernst: *SimpleScalar: An Infrastructure for computer System Modeling*, IEEE 2002.

[BD04] David c. Black, Jack Donovan: *SystemC: From the Ground Up*, Springer 2004

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, Stuart Swan: *System Design with SystemC^{tm}*, Springer 2002

[ML05] Marco Lohse: *Network-Integrated Multimedia Middleware, Services and Applications*, Saarbrücken University, 2005

[MPEGOrg] MPEG Organization: `http://www.mpeg.org/`

[TN95] Tudor P. N.: *MPEG-2 VIDEO COMPRESSION*, 1995, `http://www.bbc.co.uk/rd/pubs/papers/paper_14/paper_14.shtml`

[SimicsDS] Virtutech: *Simics: Virtualized Software Development - Data Sheet*, `http://www.virtutech.com/`

[SimicsWP] Virtutech: *Virtualized Software Development*, `http://www.virtutech.com/files/whitepapers/wp_simics.pdf`

[ACH98] Christina Aurrecoechea, Andrew T. Campbell, Linda Hauw: *A Survey of QoS Architectures*, Columbi University, Springer-Verlag, 1998.

[JN04] Jin Jingwen, Klara Nahrstedt: *QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy*, University of Illinois at Urbana-Champaign, IEEE, 2004.

[MVM07] Shankar Mahadevan, Kashif Virk, Jan Madsen: *ARTS: A SystemC-based frame work for multiprocessor System-on-Chip modelling*, Technical University of Denmark, 2007.

[BMP98] Andy C. Bavier, A. Brady Montz, Larry L. Peterson: *Predicting MPEG Execution Times*, The University of Arizona, 1998.

[SGG09] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, Eighth Edition, John Wiley & Sons inc., 2009

[BLAC05] Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni and Marco Caccamo: *Soft Real-Time Systems - Predictability vs. Efficiency*, Springer, 2005

[SP06] Simon Perathoner: *Evaluation and Comparison of Performance Analysis Methods for Distributed Embedded Systems*, Swiss Federal Institute of Technology, Zürich, 2006

[MH07] Max Hailperin: *Operating Systems and Middleware: Supporting Controlled Interaction*, 2007. Online resource to update information in the book: `http://gustavus.edu/+max/os-book/updates/CFS.html`

[RR02] Rassol Raissi: *The Theory Behind MP3*, 2002, `http://www.mp3-tech.org/programmer/docs/mp3_theory.pdf`

[BP00] Karlheinz Brandenburg, Harald Popp: *An Introduction to MPEG Layer-3*, Fraunhofer Instittut Für Integrierte Schaltungen (IIS), 2000

[AM05] Alexander Maksyagin: *Modeling Multimedia Workloads for Embedded System Design*, phd. thesis, Institut für Technishe Informatic und Kommunikationsnetze, Eidgenössische Technische Hochschule Zürich, 2005.

Appendix A

Developed utilities

A.1 BreakDiff

The breakDiff utility is used for extracting the cycle counts between up to 10 breakpoint sets from the trace files generated by the modified SimpleScalar tool set presented in section 6. The purpose of the utility is to calculate the time difference between the two breakpoints in each breakpoint set. The difference is stored in a new file, and is a number for either the instruction count or the cycle count between the two breakpoints. The sim-outorder simulator from SimpleScalar generates traces with processor cycle counts, while the rest of the simulators generate the traces based on the instruction count.

The utility takes a file name and a sequence of numbers defining the breakpoints as parameters. The program then outputs a file for each breakpoint set and an additional file which contains all the breakpoint data, and a summary section at the end. The parameters are structured as follows:

breakDiff <file name> <# breakpoint sets> <start> <end> <name>
[<start> <end> <name>]

A.2 mpeg2stat

The MPEG stat application is based on the MPEG open source implementation of the MPEG decoder at [MPEGOrg]. The open source implementation is already able to extract the header information in the various parts of an MPEG video file. The mpeg2stat application can either print a summary table containing the most interesting characteristics of a file to the terminal, or create a trace file where the type of frame each macroblock is part of. The frame type can either be I, P or B frames as explained in section 2.1.1. The last option is to make a full trace containing most of the header information in the MPEG video file.

The command for the mpeg2stat application is:
mpeg2stat -b <mpeg video filename> -v<level>

Where

-v0 Prints a summary of the MPEG video file to the terminal

-v1 Prints the same short summary as for the -v0 option. Further the application generates a trace file, where the type of each of frame is logged. The values range from 1-3 corresponding to I, P and B frames. The summary table is appended to the trace files, which is based on the input file name and suffixed with *.FrameTrace*.

-v2 Prints out many details regarding a video file to the terminal. No additional trace file is created. This is only useful if very specific information is required.

A.3 VCD Parser

The VCDParser parses a vcd (Value Change Dump) file and writes a new more readable file, which easily can be plotted using gplot. The parser can either be used simply with the file name as input thereby extracting all the variables in the file, or the specific variables can be specified as parameters thereby only extracting a subset of the variables in the file.

The first column in the output file is the time, while the consecutive columns each represent a variable. There is a row of data for all variables each time just one of the variables changes value. If events for the variables are important then the utility needs to be run once for each variable, and then saving one file for each variable. This is not a feature of the VCDParser, but can easily be done using a script.

Appendix B

Input generator class diagram

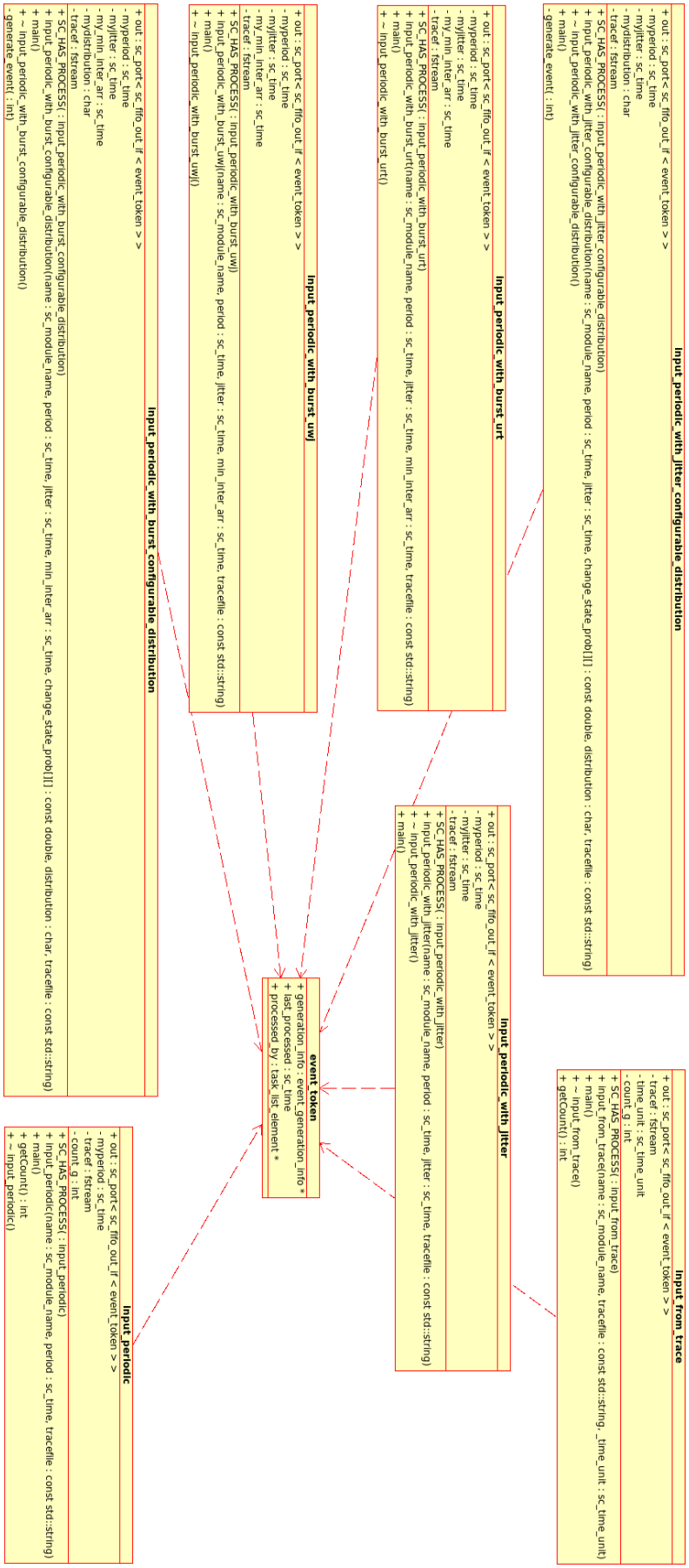

Figure B.1: *Class diagram of the various input token generators that can be used in the DiMAS API.*

Appendix C

Users guide

The DiMAS API requires that the SystemC library is compiled and working correctly. SystemC is downloaded for free at the official home page: `http://www.systemc.org/downloads/standards/`.

The DiMAS API has only been tested on the Linux platform, but should also work on Windows platforms. The DiMAS library was implemented with the KDevelopment programming environment, using an automake project. The SystemC library must be referenced as described in the KDevelop documentation, in order for the DiMAS to compile correctly.

Alternative the DiMAS library can be compiled in the terminal. In order for this to be possible the include paths in the two files */Makefile.am* and */src/Makefile.am* must be updated. These lines must be either the absolute or the relative path to the folder containing the header files for SystemC. Further the *LDFLAGS* must be used in the *make* command, where the path to the compiled SystemC library file is located[1].

Test benches are typically created in the *main.cpp* file.

It can be quite complicated to create a large test bench with many tasks and processing elements. Therefore it is advisable to make a diagram, where all the modules in the test bench are named, and use a suitable naming convention.

A test bench typically include the following modules.

- Input generators
- Tasks
- Resources (Processing elements with a specific scheduling algorithm)
- Buffers

[1] There is definitely an easier way, but this is not known by the author.

- Display devices
- Consumers

If a specific multimedia set-up is tested using different types of resources, it is advisable to partition the test bench into three parts.

Physical test bench Here the modules that are common for the tests are created. These include *input generators*, *tasks*, *buffers*, *display devices* and *Consumers*. Further the connections between tasks and buffers are done here.

Resource This part is created for each of the scheduling techniques which should be tested. The *resources* that should be used in the test are initialized, and the tasks in the test bench are assigned the resources. Further resource specific operations are performed. This could for example be assigning the correct trace files, and processor speeds. If the buffer tracing should be used the trace file is opened and the buffers are added to the trace file. Further all the output files for the *display* and *consumer* modules are given in this part.

Post processing This part contains all the operations, which should be performed after simulation. These include closing the vcd file. Further calling the functions *write_latencies* and *traceSummary* for the *display* and *Consumer* modules respectively.

Listings C.1 shows an example of a test bench, where the three partitions are included. The code in the listing correspond to the test bench in figure C.1.

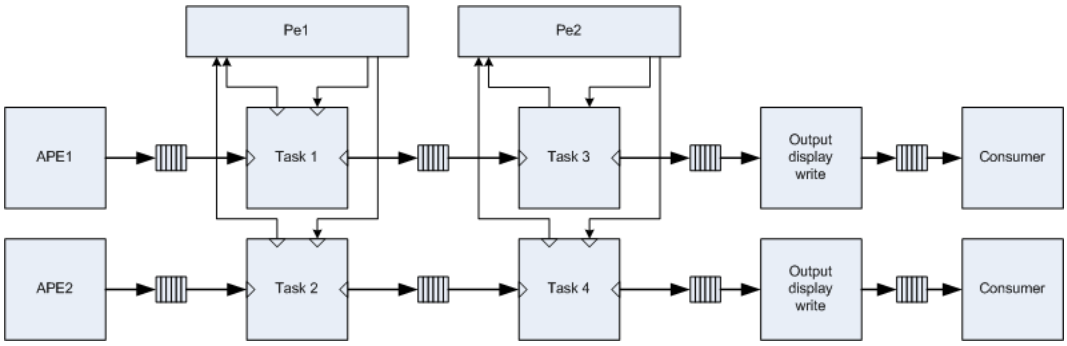

Figure C.1: *Example of a test bench with two processing elements and two continuous media streams.*

Listing C.1: *Example of a test bench*

```
1  #define MAXBUFFER 20000
2
3  bool verbose = false;
4
5  int sc_main (int argc , char *argv[]) {
6  //    initialize the random number generator
7  // srand(-24);
8
9  /* ================================================
10             Test bench for TC1
11    ================================================
```

```
12        2 processing elements
13        2 multimedia files
14     ============================================== */
15
16  // Token generators
17  // input_periodic ape1_generator("ape1_input", sc_time(25252, SC_NS), "
        TC1_ape1_input_tracer.tra");
18  // input_periodic ape2_generator("ape2_input", sc_time(133332, SC_NS), "
        TC1_ape2_input_tracer.tra");
19     input_from_trace ape1_generator("ape1_input", "TC1_ape1_input_tracer.tra",
           SC_PS); // ~30fps*1320mb /second
20     input_from_trace ape2_generator("ape2_input", "TC1_ape2_input_tracer.tra",
           SC_PS); // ~25fps*300mb / second
21
22  // Processing Element 1 with two tasks
23     task t1("ape1_vldIq");
24     task t2("ape2_vldIq");
25  // Processing Element 2 with two tasks
26     task t3("ape1_idctMp");
27     task t4("ape2_idctMp");
28
29  // Buffers
30     my_sc_fifo<event_token> t1_input(MAXBUFFER);
31     my_sc_fifo<event_token> t2_input(MAXBUFFER);
32     my_sc_fifo<event_token> t1_t3_input(MAXBUFFER);
33     my_sc_fifo<event_token> t2_t4_input(MAXBUFFER);
34     my_sc_fifo<event_token> t3_output(MAXBUFFER);
35     my_sc_fifo<event_token> t4_output(MAXBUFFER);
36     my_sc_fifo<event_token> ape1_consume(MAXBUFFER);
37     my_sc_fifo<event_token> ape2_consume(MAXBUFFER);
38
39  // Statistics
40     output_display_write ape1_display("ape1_0");
41     output_display_write ape2_display("ape2_0");
42
43  // Consumers
44     Consumer ape1_con("ape1_consumer", 1320, sc_time(33332, SC_US));
45     Consumer ape2_con("ape2_consumer", 300, sc_time(40, SC_MS));
46
47  // Buffer Mappings
48     // ape1 - Short period multimedia task
49     cout << "Connecting ape1 streams...";
50     ape1_generator.out(t1_input);
51     t1.in[0](t1_input);
52     t1.out[0](t1_t3_input);
53     t3.in[0](t1_t3_input);
54     t3.out[0](t3_output);
55     ape1_display.in(t3_output);
56     ape1_display.out(ape1_consume);
57     ape1_con.in(ape1_consume);
58     cout << "Success" << endl;
59
60     // ape2 - Long period multimedia task
61     cout << "Connecting ape2 streams...";
62     ape2_generator.out(t2_input);
63     t2.in[0](t2_input);
64     t2.out[0](t2_t4_input);
65     t4.in[0](t2_t4_input);
66     t4.out[0](t4_output);
67     ape2_display.in(t4_output);
68     ape2_display.out(ape2_consume);
69     ape2_con.in(ape2_consume);
70     cout << "Success" << endl;
71
72     timeStatus timer("TimeStatus", sc_time(1, SC_SEC));
73
74     time_t timeStart, timeEnd;
75     struct tm * timeInfo;
76
77  /* ============================================
```

```
78              Test 10 TC1.0
79      ================================================
80         Fixed Priority Scheduling
81         2 multimedia files
82      ================================================ */
83
84  // IDs for Pe1
85  #define T1_ID 0
86  #define T2_ID 1
87  // IDs for Pe2
88  #define T3_ID 0
89  #define T4_ID 1
90
91  // Fixed-Priority scheduling for Pe1
92      resource_n_tasks_fixed_priority_preemptive pe_1("CPU1", 2); // Two tasks
93      pe_1.setProcessorSpeed(270000000.0); // 270 MHz / 860 MHz
94      pe_1.assign_task(t1, T1_ID, SC_ZERO_TIME, 1);
95      pe_1.assign_task(t2, T2_ID, SC_ZERO_TIME, 2);
96
97      pe_1.setRunMode(T1_ID, 2); // Cycle time trace
98      pe_1.setInputTraceFile(T1_ID, "TC1_ape1_0_exect.tra", 1); // Trace data in
            second column
99      pe_1.setCycleMultiplier(T1_ID, 0.557); // ~inst to cycles
100     pe_1.setRunMode(T2_ID, 2); // Cycle time trace
101     pe_1.setInputTraceFile(T2_ID, "TC1_ape2_0_exect.tra", 1); // Trace data in
            second column
102     pe_1.setCycleMultiplier(T2_ID, 0.891); // ~inst to cycles
103
104
105 // Fixed-Priority scheduling for Pe2
106     resource_n_tasks_fixed_priority_preemptive pe_2("CPU2", 2); // Two tasks
107     pe_2.setProcessorSpeed(860000000.0); // 860 MHz / 1.5 GHz
108     pe_2.assign_task(t3, T3_ID, SC_ZERO_TIME, 1);
109     pe_2.assign_task(t4, T4_ID, SC_ZERO_TIME, 2);
110
111     pe_2.setRunMode(T3_ID, 2); // Cycle time trace
112     pe_2.setInputTraceFile(T3_ID, "TC1_ape2_1_exect.tra", 1); // Trace data in
            second column
113     pe_2.setCycleMultiplier(T3_ID, 0.139); // ~inst to cycles
114     pe_2.setRunMode(T4_ID, 2); // Cycle time trace
115     pe_2.setInputTraceFile(T4_ID, "TC1_ape2_1_exect.tra", 1); // Trace data in
            second column
116     pe_2.setCycleMultiplier(T4_ID, 0.244); // ~inst to cycles
117
118 // Trace file
119     sc_trace_file *tracefile;
120     tracefile = sc_create_vcd_trace_file("TC1_0_buffers");
121
122     t1_input.logger_trace(tracefile, "t1_input", "010");
123     t2_input.logger_trace(tracefile, "t2_input", "010");
124     t1_t3_input.logger_trace(tracefile, "t1_t3_input", "010");
125     t2_t4_input.logger_trace(tracefile, "t2_t4_input", "010");
126     t3_output.logger_trace(tracefile, "t3_output", "010");
127     t4_output.logger_trace(tracefile, "t4_output", "010");
128     ape1_consume.logger_trace(tracefile, "ape1_con", "010");
129     ape2_consume.logger_trace(tracefile, "ape2_con", "010");
130
131     ape1_display.setOutputTraceFile("TC1_0_ape1_display.tra");
132     ape2_display.setOutputTraceFile("TC1_0_ape2_display.tra");
133
134     ape1_con.setTraceFile("TC1_0_ape1_con.tra");
135     ape2_con.setTraceFile("TC1_0_ape2_con.tra");
136
137     time(&timeStart);
138     timeInfo = localtime(&timeStart);
139
140     cout << "Starting simulation @ " << asctime(timeInfo) << endl;
141     sc_start(15, SC_SEC);
142
143     ofstream sumFile("TC1_0_sum_270M_860M.txt");
```

```
144 // ofstream sumFile("TC1_0_sum_860M_1500M.txt");
145    sumFile << "Simulation summary for test case TC1_0" << endl;
146    sumFile << "Pe1: 270 MHz\t Pe2: 860 MHz" << endl << endl;
147 // sumFile << "Pe1: 860 MHz\t Pe2: 1.5 GHz" << endl << endl;
148
149
150 /* ================================================
151       Post processing
152    ================================================ */
153
154    time(&timeEnd);
155    timeInfo = localtime(&timeEnd);
156    cout  << "Simulation ended @ " << asctime(timeInfo) << endl;
157    cout  << "Simulation time was: "
158          << (difftime(timeEnd, timeStart) / 60) << " minutes" << endl;
159
160    sc_close_vcd_trace_file(tracefile);
161
162    ape1_display.write_latencies("ape1_input");
163    ape2_display.write_latencies("ape2_input");
164
165    cout  << "Lost frame for ape1: " << ape1_con.getFails() << " / "
166          << ape1_con.getSuccess() + ape1_con.getFails() << endl;
167    cout  << "Lost frame for ape2: " << ape2_con.getFails()
168          << " / " << ape2_con.getSuccess() + ape2_con.getFails() << endl;
169
170    ape1_con.traceSummary();
171    ape2_con.traceSummary();
172
173    cout << "count from ape1: " << ape1_generator.getCount() << endl;
174    cout << "count from ape2: " << ape2_generator.getCount() << endl;
175
176    cout << "Task 1: " << t1.input_count << "/" << t1.output_count << endl;
177    cout << "Task 2: " << t2.input_count << "/" << t2.output_count << endl;
178    cout << "Task 3: " << t3.input_count << "/" << t3.output_count << endl;
179    cout << "Task 4: " << t4.input_count << "/" << t4.output_count << endl;
180
181    pe_1.printCounts();
182    pe_2.printCounts();
183
184    sumFile << "Lost frames" << endl;
185    sumFile  << "Lost frame for ape1: " << ape1_con.getFails()
186             << " / " << ape1_con.getSuccess() + ape1_con.getFails() << endl;
187    sumFile  << "Lost frame for ape2: " << ape2_con.getFails()
188             << " / " << ape2_con.getSuccess() + ape2_con.getFails() << endl;
189
190    sumFile << endl << "Latencies" << endl;
191    sumFile  << "APE1: "
192             << "\tmin: " << ape1_display.getminlatency("ape1_input")
193             << "\tavg: " << ape1_display.getavglatency("ape1_input")
194             << "\tmax: " << ape1_display.getmaxlatency("ape1_input")
195             << endl;
196    sumFile  << "APE2: "
197             << "\tmin: " << ape2_display.getminlatency("ape2_input")
198             << "\tavg: " << ape2_display.getavglatency("ape2_input")
199             << "\tmax: " << ape2_display.getmaxlatency("ape2_input")
200             << endl;
201
202    sumFile << endl << "Token counts" << endl;
203    sumFile << "Task 1: " << t1.input_count << "/" << t1.output_count << endl;
204    sumFile << "Task 2: " << t2.input_count << "/" << t2.output_count << endl;
205    sumFile << "Task 3: " << t3.input_count << "/" << t3.output_count << endl;
206    sumFile << "Task 4: " << t4.input_count << "/" << t4.output_count << endl;
207
208    sumFile << endl << "Max. buffer backlogs" << endl;
209    sumFile << "t1_input: " << t1_input.getmaxbacklog() << endl;
210    sumFile << "t1_t3_input: " << t1_t3_input.getmaxbacklog() << endl;
211    sumFile << "t3_output: " << t3_output.getmaxbacklog() << endl;
212    sumFile << "ape1_con: " << ape1_consume.getmaxbacklog() << endl;
213    sumFile << "t2_input: " << t2_input.getmaxbacklog() << endl;
```

```
214     sumFile << "t2_t4_input: " << t2_t4_input.getmaxbacklog() << endl;
215     sumFile << "t4_output: " << t4_output.getmaxbacklog() << endl;
216     sumFile << "ape2_con: " << ape2_consume.getmaxbacklog() << endl;
217     sumFile.close();
218 
219     return 0;
220 }
```

Appendix D

Multimedia data files

D.1 MPEG-2 files

Presents a table with the most interesting characteristics of the various MPEG files used in this thesis.

File Name	H-Size	V-Size	Number of Macroblocks	Frame Rate	Profile	Level	Format	Number of Frames	Frame Types			Motion
									I	P	B	
100b_015	704	480	1320	29,97	4	8	4:2:0	450	38	113	229	No
100b_040	704	480	1320	29,97	4	8	4:2:0	450	38	113	299	No
100b_080	704	480	1320	29,97	4	8	4:2:0	450	38	113	299	No
100b_400	704	480	1320	29,97	133	8	4:2:2	450	226	0	224	No
bbc3_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	No
cact_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	Yes
flwr_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	Yes
mobl_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	Yes
mulb_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	No
pulb_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	No
susi_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	Yes
tens_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	Yes
time_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	Very little
v700_080	704	480	1320	29.97	4	8	4:2:0	450	38	113	299	No
smoke_low	320	240	300	25	4	8	4:2:0	399	23	376	0	Yes
smoke_low	640	480	1200	25	4	8	4:2:0	399	23	276	0	Yes
smoke_low	640	480	1200	30	4	8	4:2:0	479	27	452	0	Yes
smoke_medium	320	240	300	25	4	8	4:2:0	399	23	376	0	Yes
smoke_medium	640	480	1200	25	4	8	4:2:0	399	23	276	0	Yes
smoke_medium	640	480	1200	30	4	8	4:2:0	479	27	452	0	Yes
smoke_high	320	240	300	25	4	8	4:2:0	399	23	376	0	Yes
smoke_high	640	480	1200	25	4	8	4:2:0	399	23	276	0	Yes
smoke_high	640	480	1200	30	4	8	4:2:0	479	27	452	0	Yes

Table D.1: *Symmary of the characteristics of the MPEG video files, which were processed during the writing of this thesis. The file names for smoke_low -_ medium -_ high are shortened due to lack of space on the page. The frame rate and resolution are appended to the file names. The files* flwr_080 *and* smoke_high_25fps_320x480 *were used in the case studies.*

File Name	Decode Frame		VLD/IQ		IDCT/MP	
	Min	Max	Min	Max	Min	Max
100b_015	150.540.160	152.252.809	2.699	12.029	109.274	118.872
100b_040	150.589.969	152.817.213	2.699	12.029	109.274	120.884
100b_080	150.589.969	152.116.256	2.699	12.029	109.274	113.224
100b_400	200.233.129	202.063.089	3.397	16.023	145.082	148.288
bbc3_080	150.620.662	174.830.134	2.699	41.785	109.069	124.778
cact_080	156.418.246	172.078.665	2.700	43.129	109.464	128.112
flwr_080	159.331.317	173.085.536	2.699	46.205	109.381	128.158
moble_080	151.793.184	174.480.845	2.699	40.561	109.274	128.095
mulb_080	150.585.289	161.196.219	2.699	14.137	109.242	116.589
pulb_080	150.357.319	150.622.580	2.699	11.849	109.258	113.216
susi_080	160.527.650	177.306.073	2.703	41.292	109.421	128.119
tens_080	156.833.908	171.953.943	2.699	46.759	109.338	128.154
time_080	156.833.908	171.953.943	2.699	46.759	109.338	128.154
smoke_low_25fps_320x240	35.130.922	38.579.820	2.115	35.880	108.742	118.690
smoke_low_25fps_640x480	138.070.279	151.191.189	2.115	30.421	108.397	118.712
smoke_low_30fps_640x480	137.816.203	151.191.189	2.115	30.421	108.397	118.712
smoke_medium_25fps_320x240	36.040.717	38.643.062	2.115	40.813	109.253	118.652
smoke_medium_25fps_640x480	139.624.370	151.191.189	2.115	30.330	108.765	118.736
smoke_medium_30fps_640x480	138.873.698	151.191.189	2.115	30.330	108.397	118.746
smoke_high_25fps_320x240	36.968.720	39.910.160	2.115	46.255	109.250	118.649
smoke_high_25fps_640x480	142.084.292	152.259.565	2.115	30.579	108.952	118.717
smoke_high_30fps_640x480	141.443.695	151.191.189	2.115	30.330	108.493	118.723

Appendix E

Commands for generating traces

The commands below are used in the interactive debugger to configure the debugger to log three breakpoint sets, as described in section 6.

```
tracefile profile_block_xxx.txt
dbreak 0x0204b528 w
dbreak 0x0204b52c w
dbreak 0x0204b530 w
dbreak 0x0204b534 w
dbreak 0x0204b538 w
dbreak 0x0204b53c w
tracecomm trace_performed_19.01.09_by_Rolf_Kristensen
tracecomm breakpoint1:0x0204b528_global_start_decodePic
tracecomm breakpoint2:0x0204b52c_global_end_decodePic
tracecomm breakpoint3:0x0204b530_global_start_vldIq
tracecomm breakpoint4:0x0204b534_global_end_vldIq
tracecomm breakpoint5:0x0204b538_global_start_idctMp
tracecomm breakpoint6:0x0204b53c_global_end_idctMp
tracecomm bp1_and_bp2_encapsulates_decoding_of_one_frame.
tracecomm bp3_and_bp4_encapsulates_the_VLD/IQ_part_of_the_decoding_process
tracecomm bp5_and_bp6_encapsulates_the_IDCT/MP_part_of_the_decoding_process
tracecomm simulation_is_performed_using_the_sim-profile_simulator
tracecomm simulation_is_performed_on_the_file_xxx
quietTrace on
cont
```

Where xxx is the name of the multimedia file.

Appendix F

Profiling two MPEG video files

The profiling was performed using gprof on an Intel 2.8 GHz architecture. The PC CPU cycle count in the tables are obtained by multiplying the execution time with the processor rate. The ratio is calculated as the CPU cycle count divided by the summed instruction counts.

F.1 flwr_080

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 28.68      2.38     2.38  3564000     0.00     0.00  Fast_IDCT
 19.40      3.99     1.61      450     3.58    17.94  Decode_Picture
 10.60      4.87     0.88  4797882     0.00     0.00
     form_component_prediction
  8.68      5.59     0.72 228096000     0.00     0.00  putbyte
  7.47      6.21     0.62 46864367     0.00     0.00  Flush_Buffer
  5.48      6.67     0.46  1747561     0.00     0.00
     Decode_MPEG2_Non_Intra_Block
  4.94      7.08     0.41     1350     0.30     0.84  store_yuv1
  3.37      7.36     0.28  3564000     0.00     0.00  Clear_Block
  2.17      7.54     0.18                             conv420to422
  1.81      7.69     0.15   303534     0.00     0.00  Decode_MPEG2_Intra_Block
  1.57      7.82     0.13 24142950     0.00     0.00  Get_Bits
  1.33      7.93     0.11  1599294     0.00     0.00  form_prediction
  0.84      8.00     0.07 47483182     0.00     0.00  Show_Bits
  0.72      8.06     0.06   543411     0.00     0.00  form_predictions
  0.48      8.10     0.04  2111630     0.00     0.00  decode_motion_vector
  0.36      8.13     0.03  3147849     0.00     0.00  Get_Bits1
  0.36      8.16     0.03                             Decode_MPEG1_Intra_Block
  0.24      8.18     0.02  2111630     0.00     0.00  Get_motion_code
  0.24      8.20     0.02  1055815     0.00     0.00  motion_vector
  0.24      8.22     0.02   594000     0.00     0.00  macroblock_modes
  0.24      8.24     0.02      450     0.04     2.56  Write_Frame
  0.12      8.25     0.01   594000     0.00     0.00  Get_macroblock_type
  0.12      8.26     0.01   455663     0.00     0.00  Get_coded_block_pattern
  0.12      8.27     0.01   202356     0.00     0.00  Get_Luma_DC_dct_diff
  0.12      8.28     0.01     7349     0.00     0.00  Fill_Buffer
  0.12      8.29     0.01                             Spatial_Prediction
  0.06      8.29     0.01   799300     0.00     0.00  motion_vectors
  0.06      8.30     0.01   101178     0.00     0.00  Get_Chroma_DC_dct_diff
  0.06      8.30     0.01      452     0.01     0.01  Get_Hdr
  0.00      8.30     0.00   594000     0.00     0.00
     Get_macroblock_address_increment
  0.00      8.30     0.00    14858     0.00     0.00  next_start_code
```

```
  0.00      8.30     0.00    14405     0.00     0.00  Flush_Buffer32
  0.00      8.30     0.00    13500     0.00     0.00  slice_header
  0.00      8.30     0.00      454     0.00     0.00  Get_Bits32
  0.00      8.30     0.00      452     0.00     0.01  Headers
  0.00      8.30     0.00      452     0.00     0.00  extension_and_user_data
  0.00      8.30     0.00      450     0.00     0.00  extra_bit_information
  0.00      8.30     0.00      450     0.00     2.51  store_one
  0.00      8.30     0.00        3     0.00     0.00  Initialize_Buffer
  0.00      8.30     0.00        3     0.00     0.00  marker_bit
  0.00      8.30     0.00        1     0.00     0.00  Initialize_Fast_IDCT
  0.00      8.30     0.00        1     0.00     2.56
      Output_Last_Frame_of_Sequence
```

Block	Description	value
VLD/IQ	Sum instruction counts	$4.52521e^9$
	PC execution time	$0.9s$
	PC CPU cycles	$2.52e^9$
	Ratio	0.557
IDCT/MP	Sum instruction counts	$6.94239e^{10}$
	PC execution time	$3.45s$
	PC CPU cycles	$9.66e^9$
	Ratio	0.139

Table F.1: *Summary of profiling calculations for flwr_ 080*

F.2 high_25fps_320x240

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 32.64      0.94     0.94   718200     0.00     0.00  Fast_IDCT
 14.58      1.36     0.42   633492     0.00     0.00
     Decode_MPEG2_Non_Intra_Block
 11.11      1.68     0.32      399     0.80     6.92  Decode_Picture
 10.77      1.99     0.31 22715738     0.00     0.00  Flush_Buffer
  7.64      2.21     0.22   666834     0.00     0.00
     form_component_prediction
  4.86      2.35     0.14 45964800     0.00     0.00  putbyte
  3.47      2.45     0.10     1197     0.08     0.20  store_yuv1
  3.13      2.54     0.09                             conv420to422
  2.43      2.61     0.07 11020727     0.00     0.00  Get_Bits
  2.43      2.68     0.07   718200     0.00     0.00  Clear_Block
  2.08      2.74     0.06 22845348     0.00     0.00  Show_Bits
  1.39      2.78     0.04   222278     0.00     0.00  form_prediction
  1.04      2.81     0.03    51366     0.00     0.00  Decode_MPEG2_Intra_Block
  0.69      2.83     0.02   203584     0.00     0.00  Get_motion_code
  0.69      2.85     0.02     4236     0.00     0.00  Fill_Buffer
  0.69      2.87     0.02                             Decode_MPEG1_Intra_Block
  0.35      2.88     0.01      445     0.02     0.02  extension_and_user_data
  0.00      2.88     0.00   312616     0.00     0.00  Get_Bits1
  0.00      2.88     0.00   203584     0.00     0.00  decode_motion_vector
  0.00      2.88     0.00   119645     0.00     0.00
     Get_macroblock_address_increment
  0.00      2.88     0.00   119645     0.00     0.00  Get_macroblock_type
  0.00      2.88     0.00   119645     0.00     0.00  macroblock_modes
  0.00      2.88     0.00   111139     0.00     0.00  form_predictions
  0.00      2.88     0.00   111030     0.00     0.00  Get_coded_block_pattern
  0.00      2.88     0.00   101792     0.00     0.00  motion_vector
  0.00      2.88     0.00   101792     0.00     0.00  motion_vectors
  0.00      2.88     0.00    34244     0.00     0.00  Get_Luma_DC_dct_diff
  0.00      2.88     0.00    17122     0.00     0.00  Get_Chroma_DC_dct_diff
```

```
  0.00      2.88      0.00      7300      0.00      0.00  next_start_code
  0.00      2.88      0.00      6854      0.00      0.00  Flush_Buffer32
  0.00      2.88      0.00      5985      0.00      0.00  slice_header
  0.00      2.88      0.00       447      0.00      0.00  Get_Bits32
  0.00      2.88      0.00       401      0.00      0.03  Get_Hdr
  0.00      2.88      0.00       401      0.00      0.03  Headers
  0.00      2.88      0.00       399      0.00      0.60  Write_Frame
  0.00      2.88      0.00       399      0.00      0.00  extra_bit_information
  0.00      2.88      0.00       399      0.00      0.60  store_one
  0.00      2.88      0.00        69      0.00      0.00  marker_bit
  0.00      2.88      0.00         3      0.00      0.00  Initialize_Buffer
  0.00      2.88      0.00         1      0.00      0.00  Initialize_Fast_IDCT
  0.00      2.88      0.00         1      0.00      0.60
      Output_Last_Frame_of_Sequence
```

Block	Description	value
VLD/IQ	Sum instruction counts	$1.79033e^9$
	PC execution time	$0.57s$
	PC CPU cycles	$1.596e^9$
	Ratio	0.891
IDCT/MP	Sum instruction counts	$1.37487e^{10}$
	PC execution time	$1.2s$
	PC CPU cycles	$3.36e^9$
	Ratio	0.244

Table F.2: *Summary of profiling calculations for high_25fps_320x240*

Appendix G

CD-rom contents

Appendix H

Case Study plots

H.1 Variability in the MPEG video files

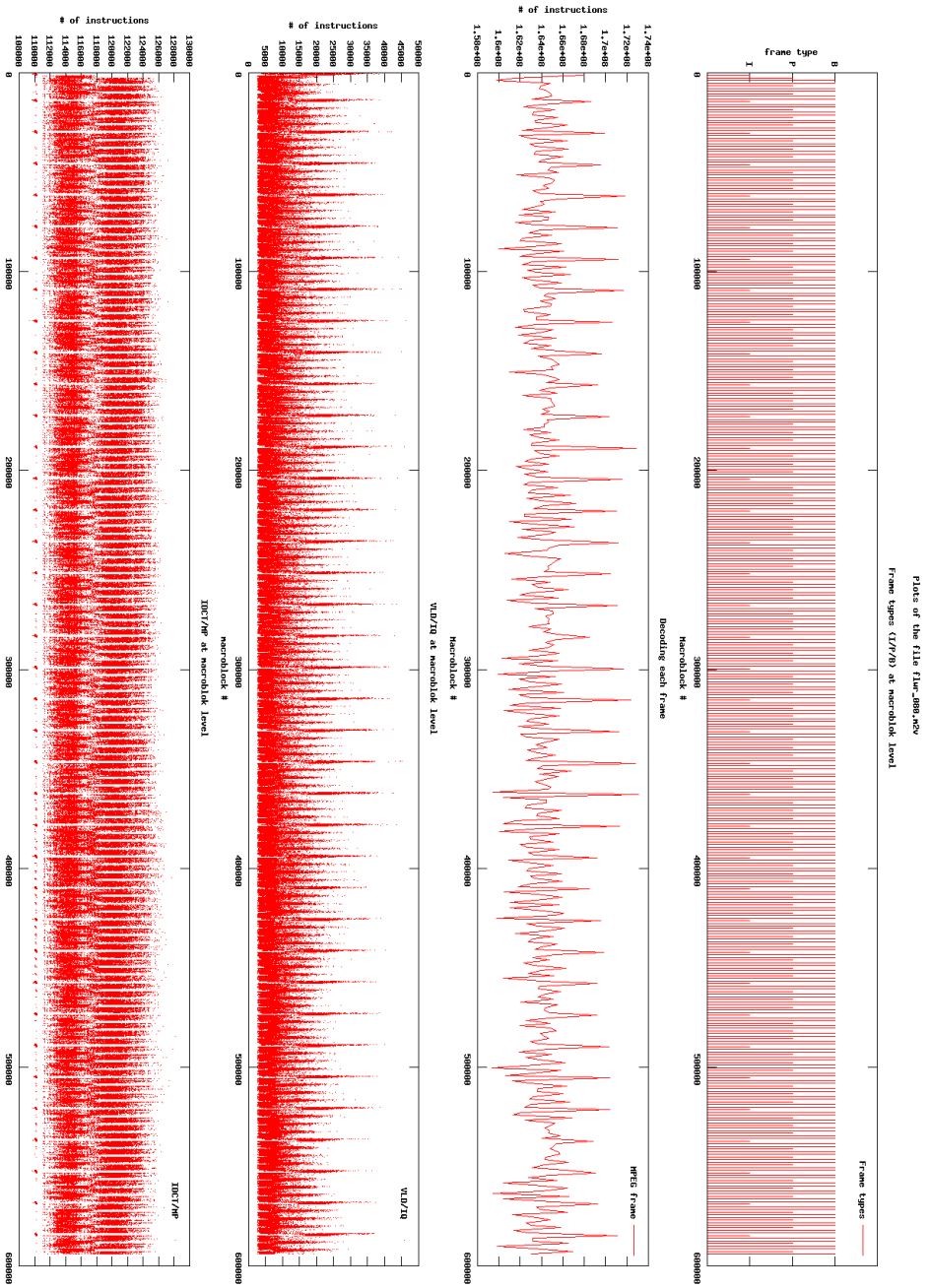

Figure H.1: *The variability in the APE$_1$ stream: flwr_080.m2v*

Figure H.2: *The variability in the APE_2 stream: high_25fps_320x480.m2v*

H.2 Case study 1

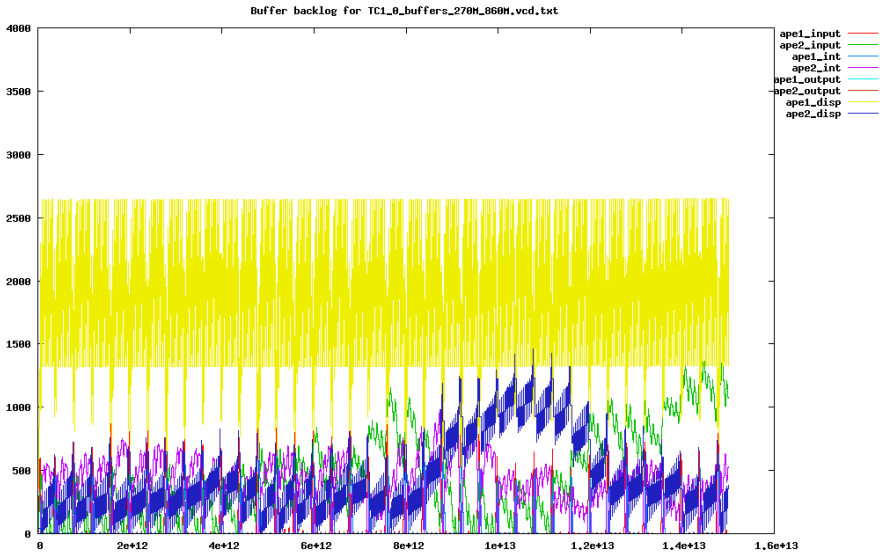

Figure H.3: *Buffer backlog for TC1.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

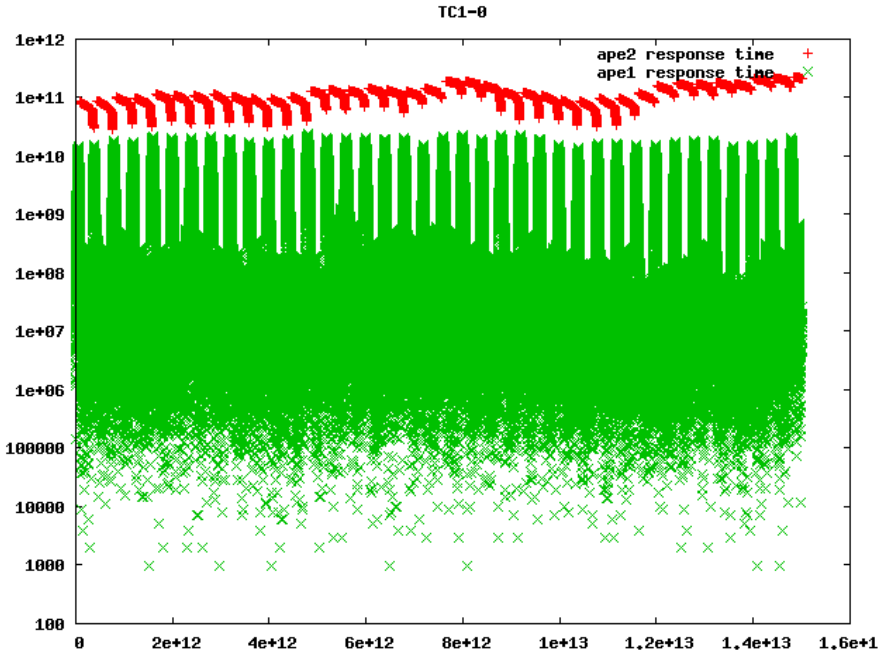

Figure H.4: *Idle response times for TC1.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

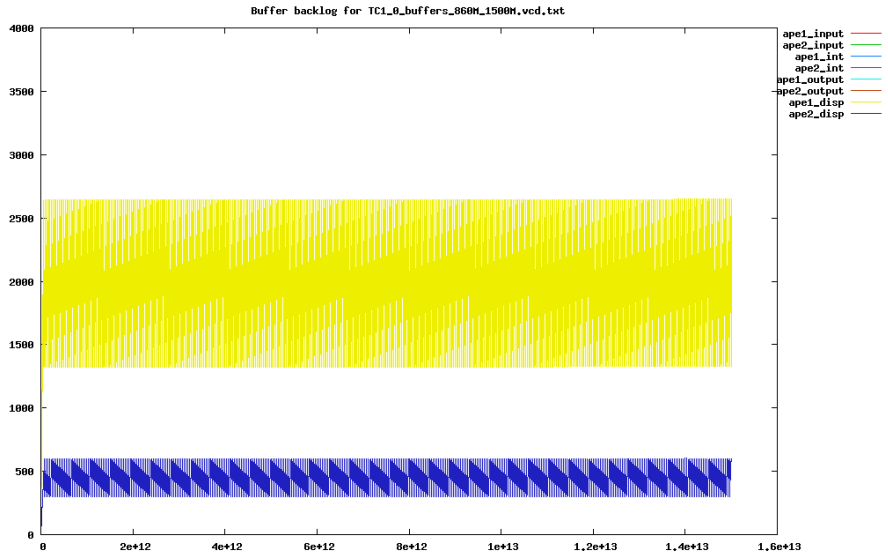

Figure H.5: *Buffer backlog for TC1.0 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

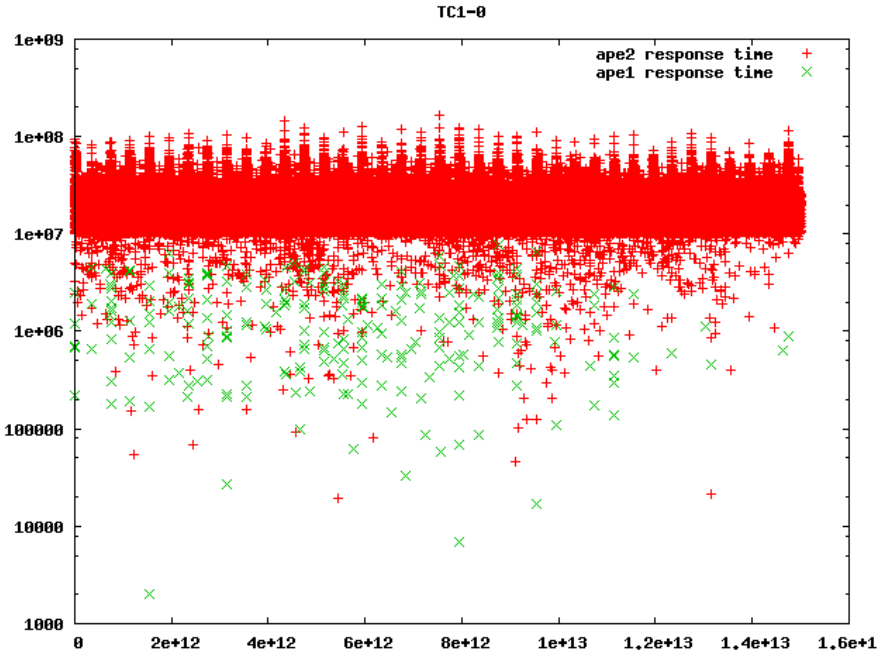

Figure H.6: *Idle response times for TC1.0 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

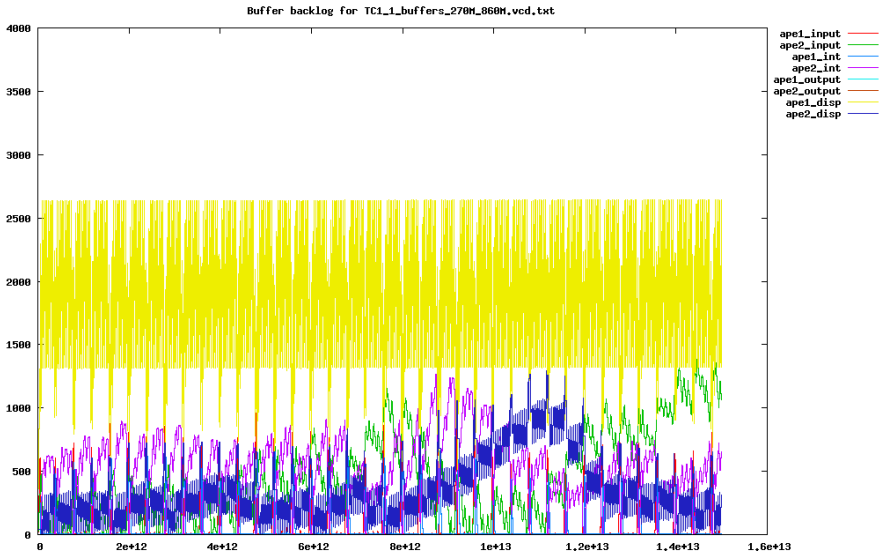

Figure H.7: *Buffer backlog for TC1.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

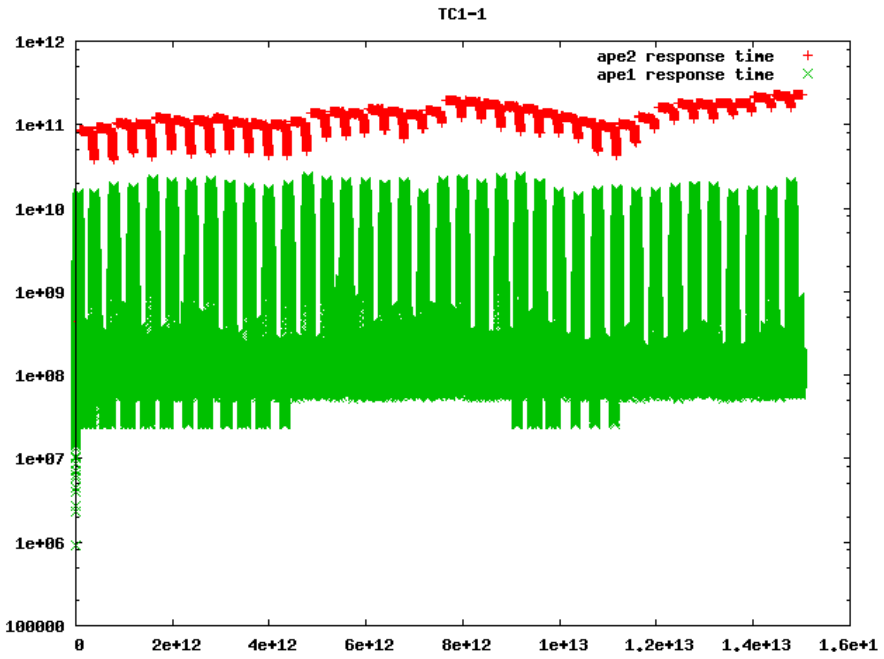

Figure H.8: *Idle response times for TC1.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

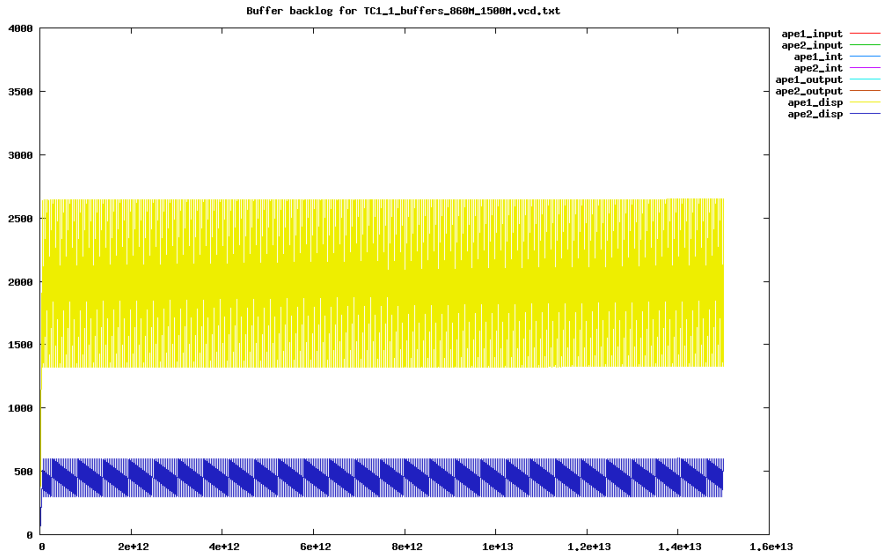

Figure H.9: *Buffer backlog for TC1.1 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

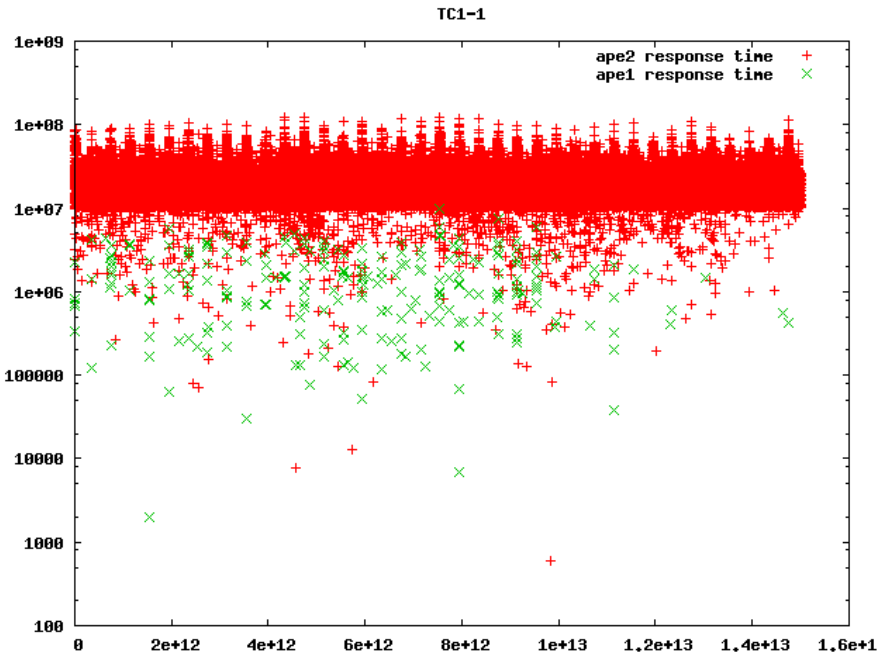

Figure H.10: *Idle response times for TC1.1 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

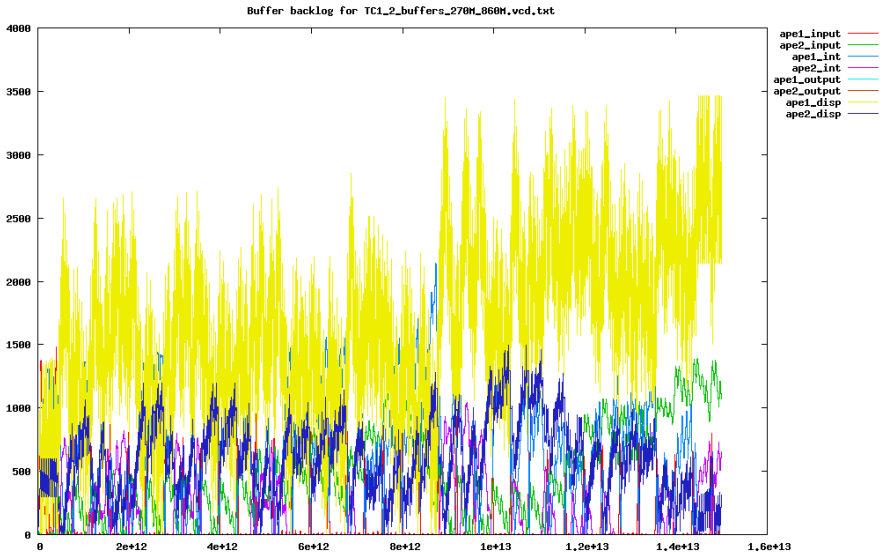

Figure H.11: *Buffer backlog for TC1.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

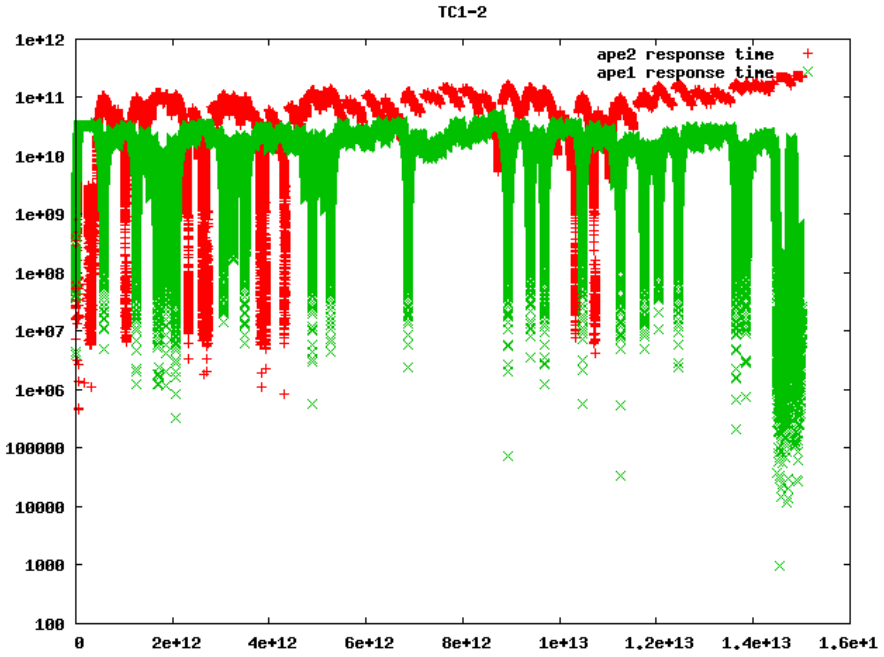

Figure H.12: *Idle response times for TC1.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

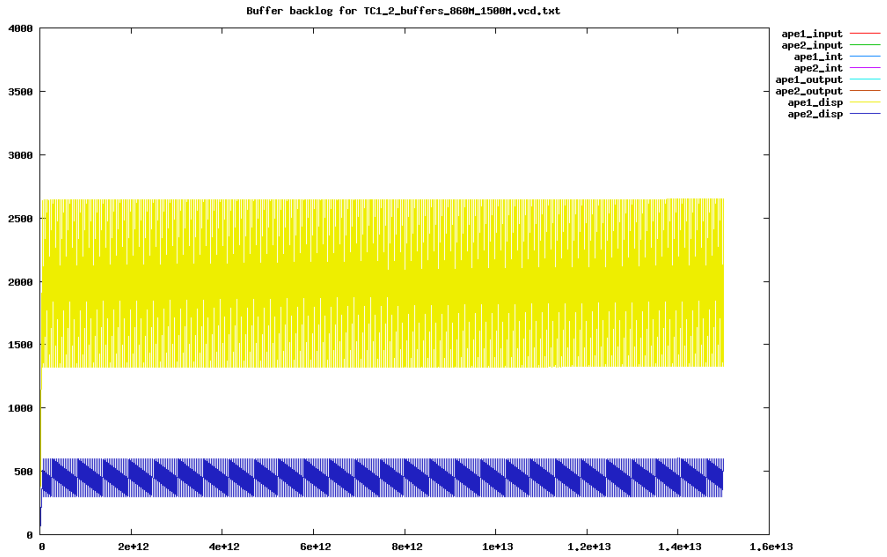

Figure H.13: *Buffer backlog for TC1.2 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

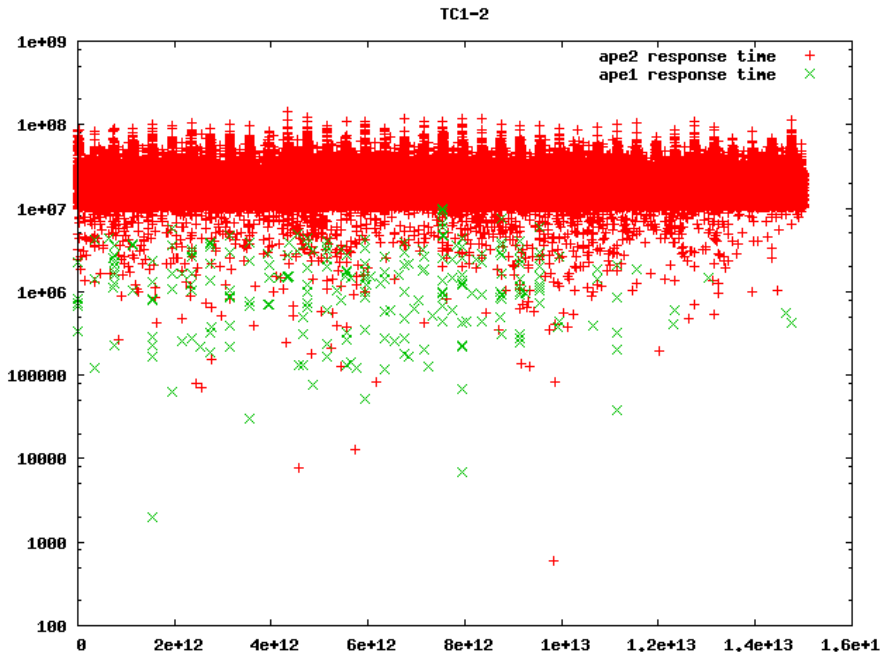

Figure H.14: *Idle response times for TC1.2 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

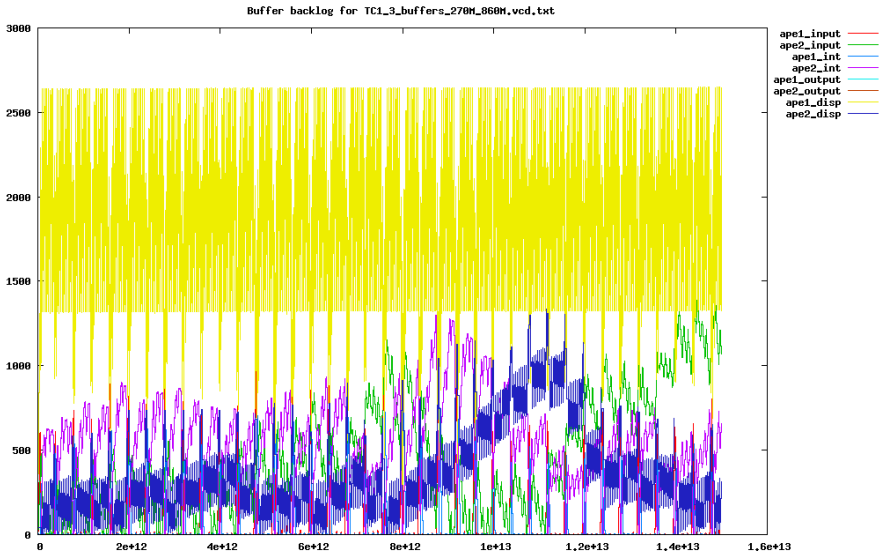

Figure H.15: *Buffer backlog for TC1.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

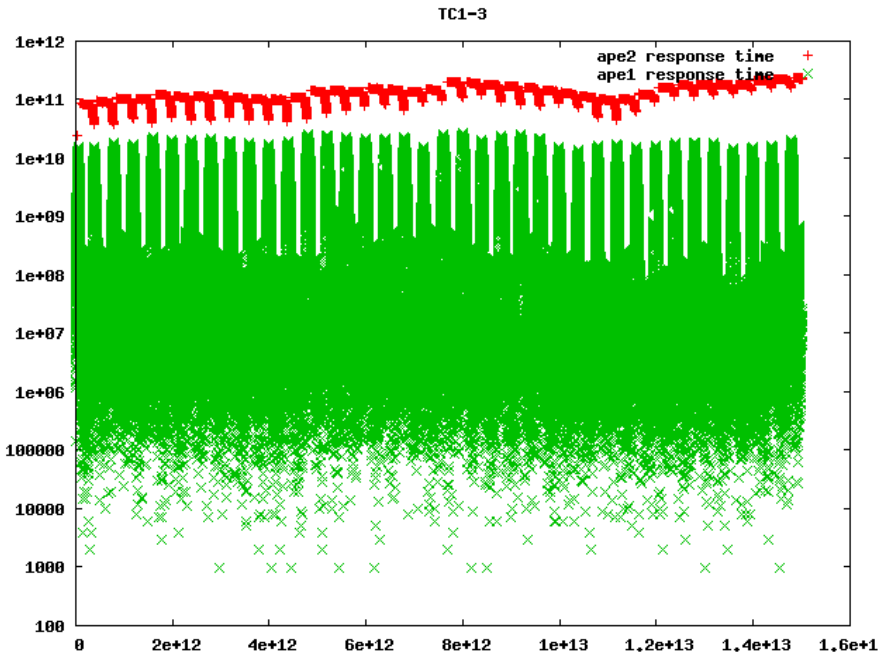

Figure H.16: *Idle response times for TC1.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

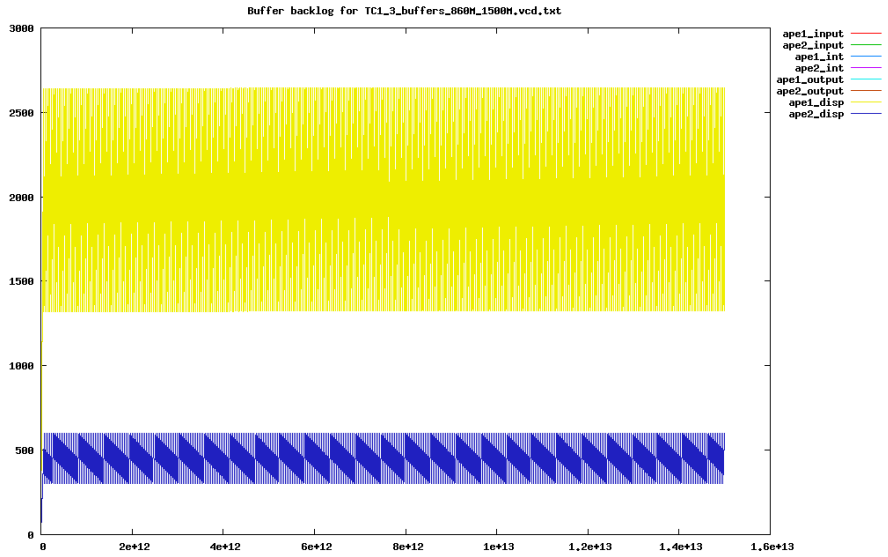

Figure H.17: *Buffer backlog for TC1.3 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

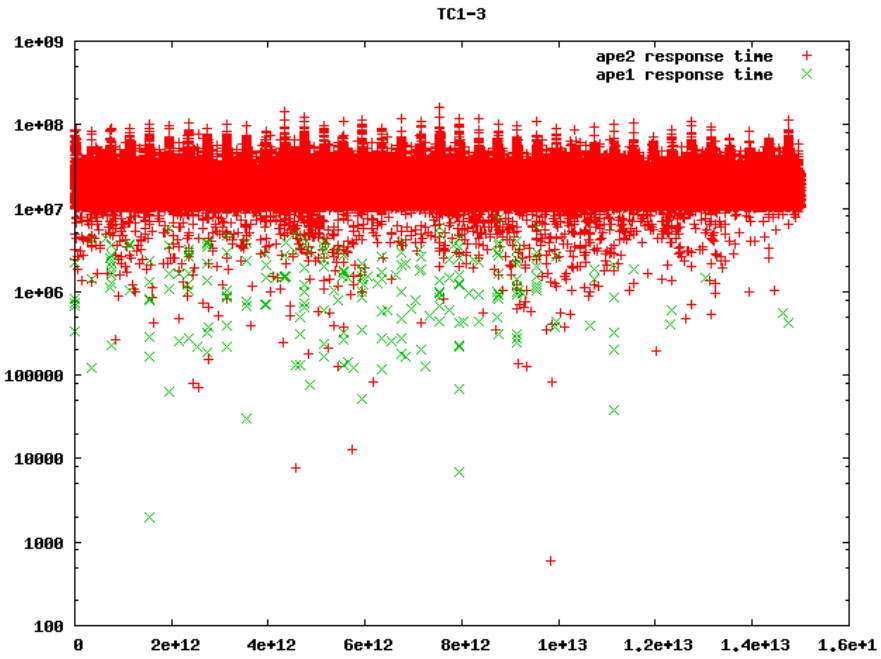

Figure H.18: *Idle response times for TC1.3 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.3 Case study 2

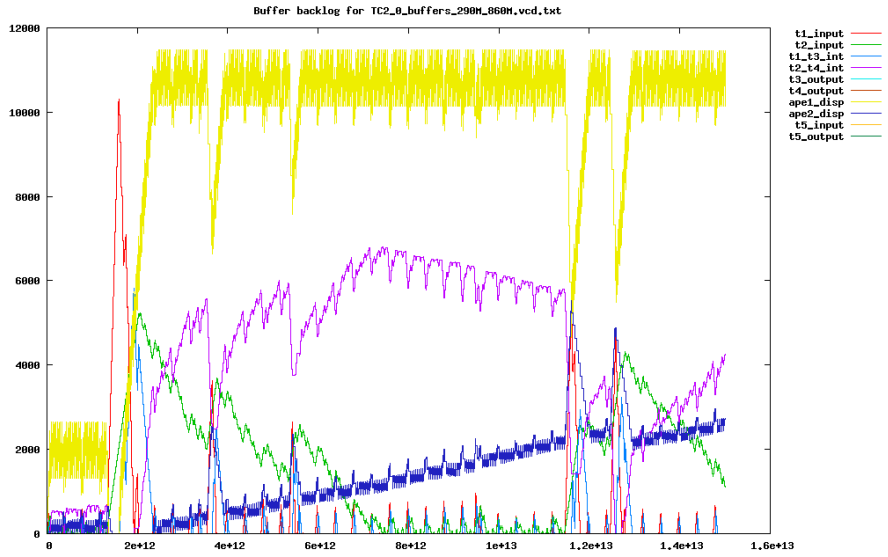

Figure H.19: *Buffer backlog for TC2.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

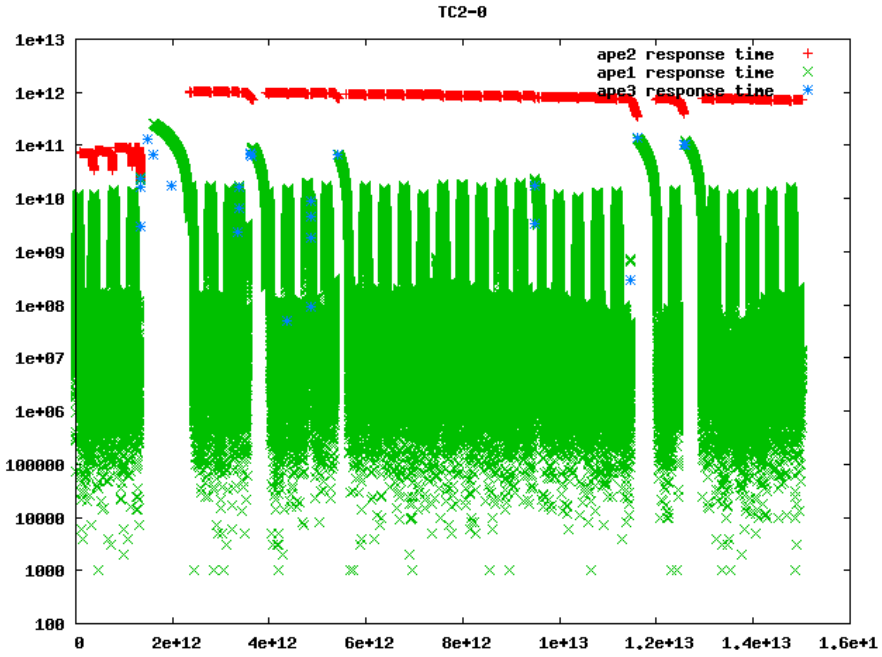

Figure H.20: *Idle response times for TC2.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

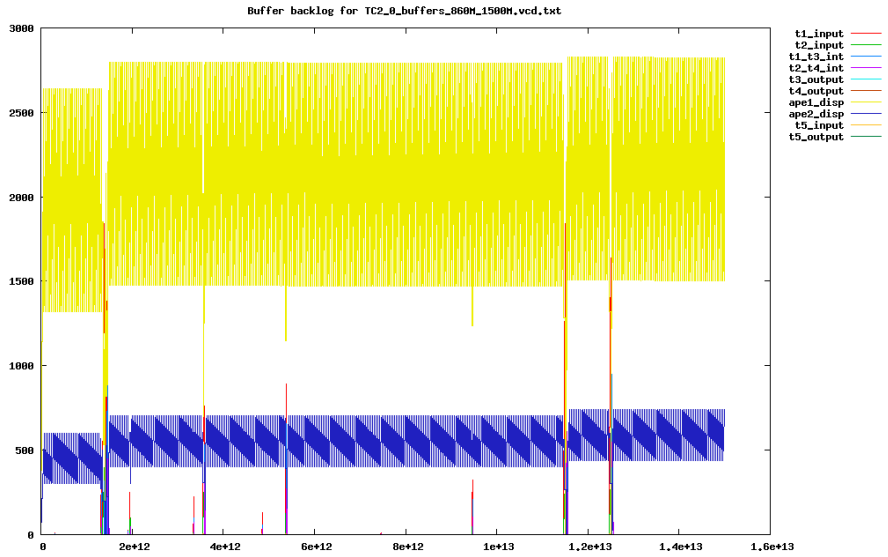

Figure H.21: *Buffer backlog for TC2.0 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

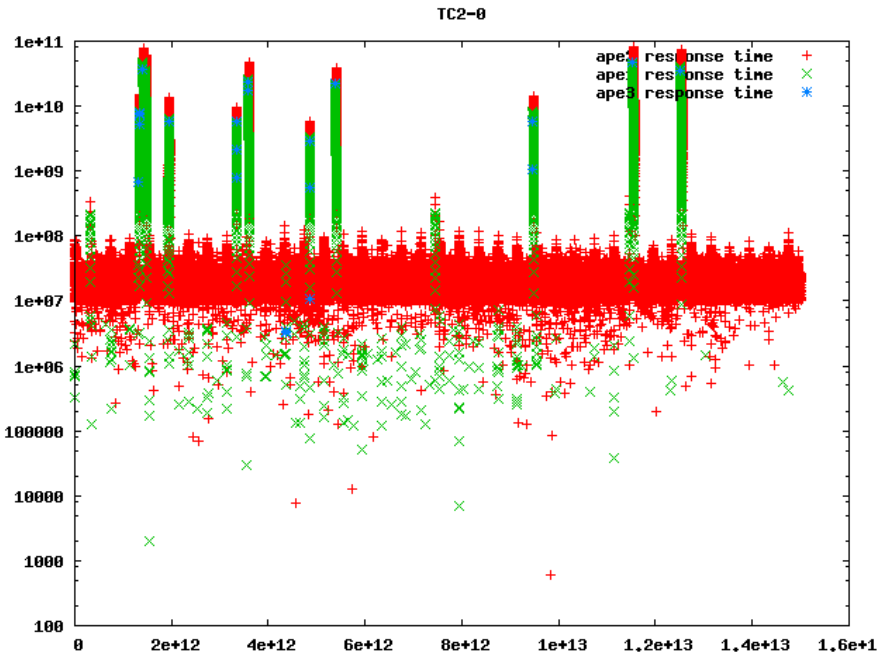

Figure H.22: *Idle response times for TC2.0 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

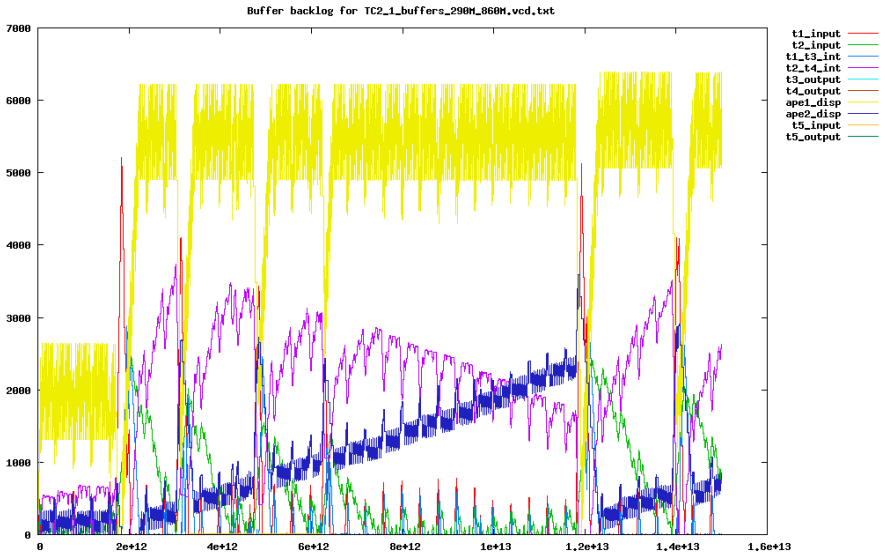

Figure H.23: *Buffer backlog for TC2.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

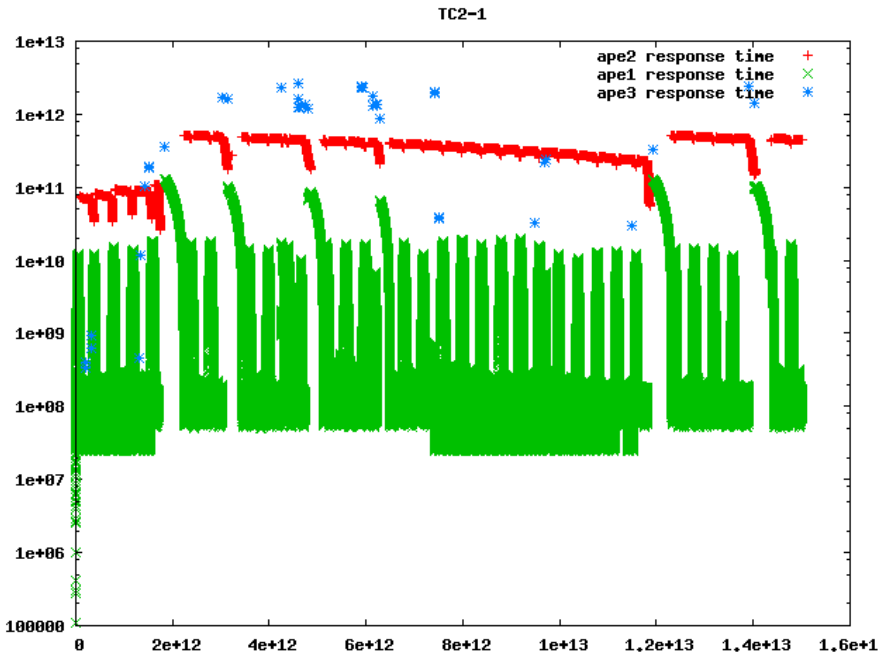

Figure H.24: *Idle response times for TC2.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

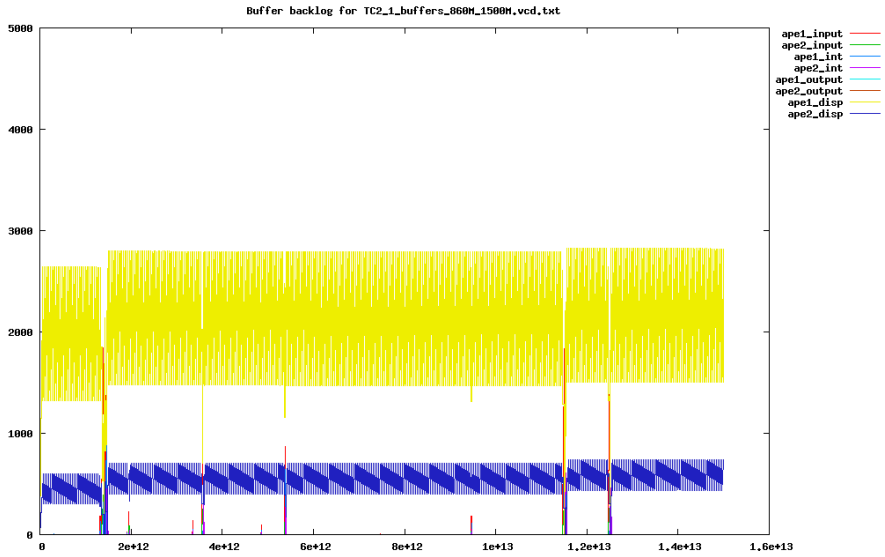

Figure H.25: *Buffer backlog for TC2.1 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

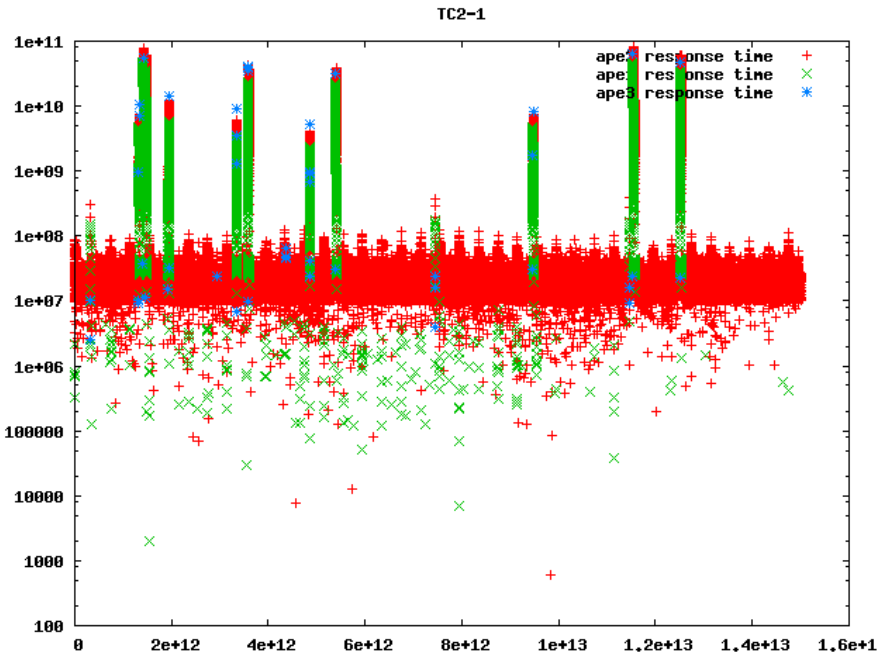

Figure H.26: *Idle response times for TC2.1 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

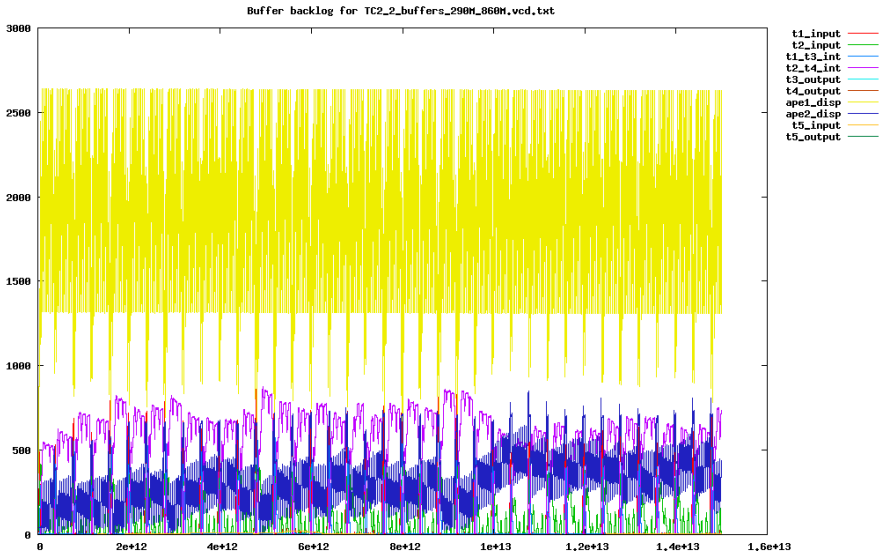

Figure H.27: *Buffer backlog for TC2.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

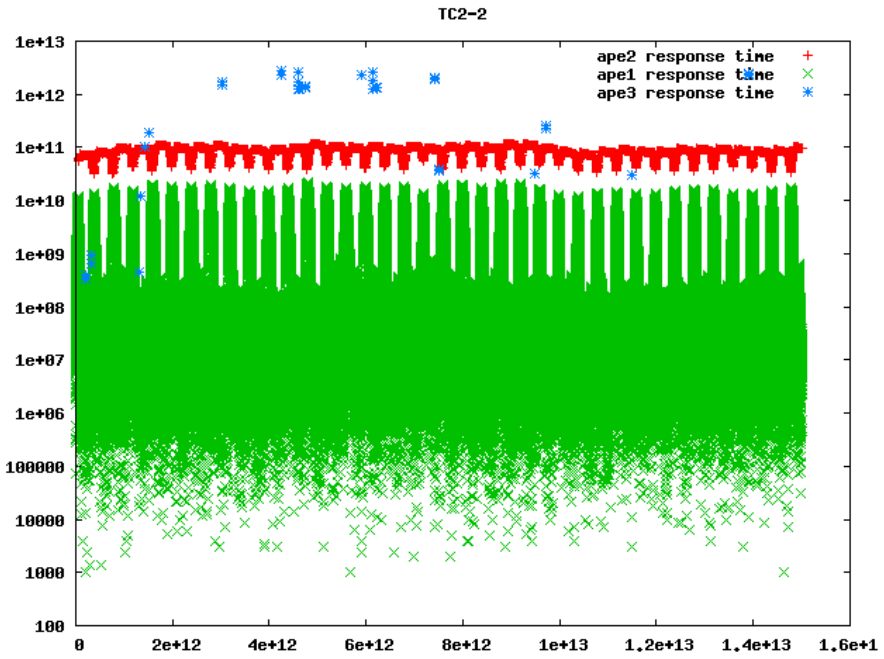

Figure H.28: *Idle response times for TC2.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

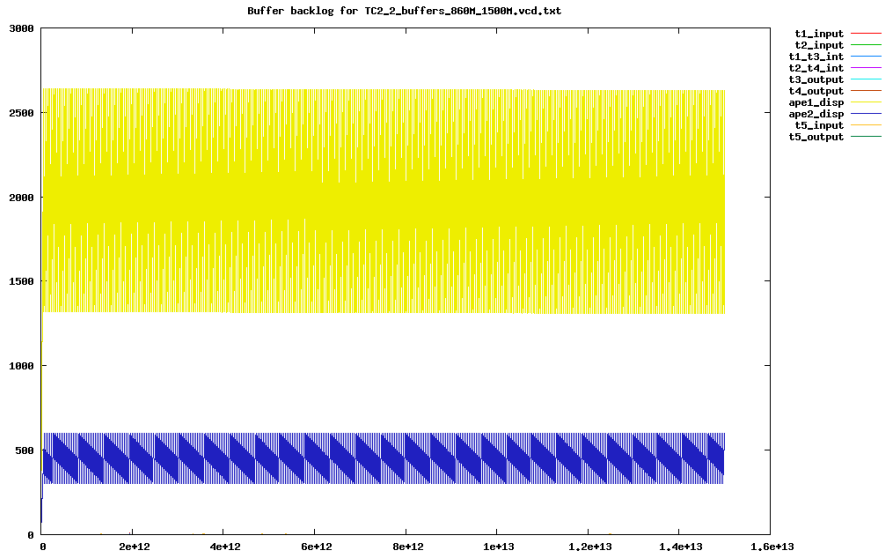

Figure H.29: *Buffer backlog for TC2.2 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

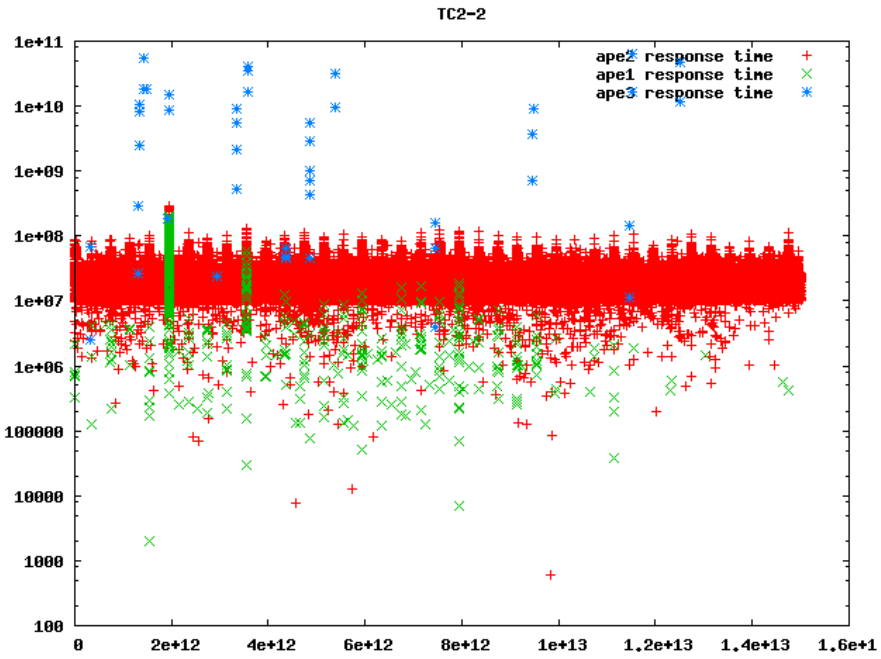

Figure H.30: *Idle response times for TC2.2 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

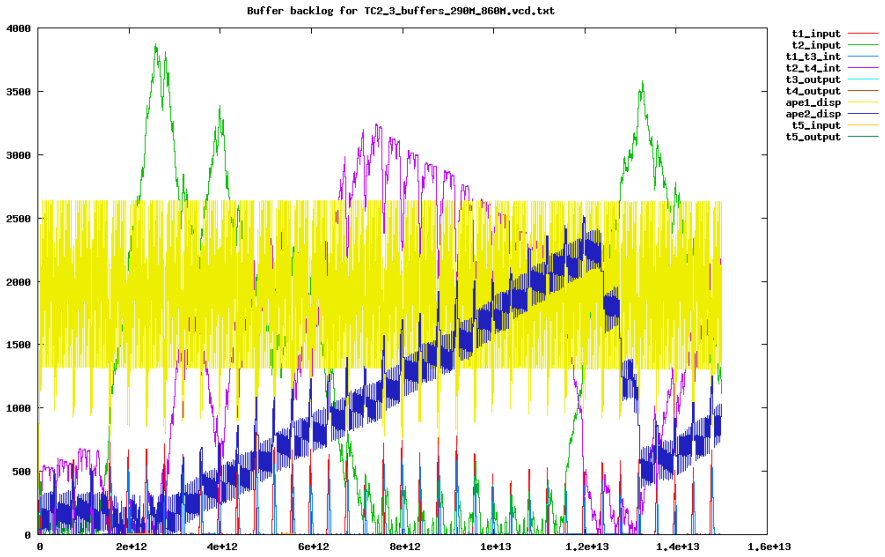

Figure H.31: *Buffer backlog for TC2.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

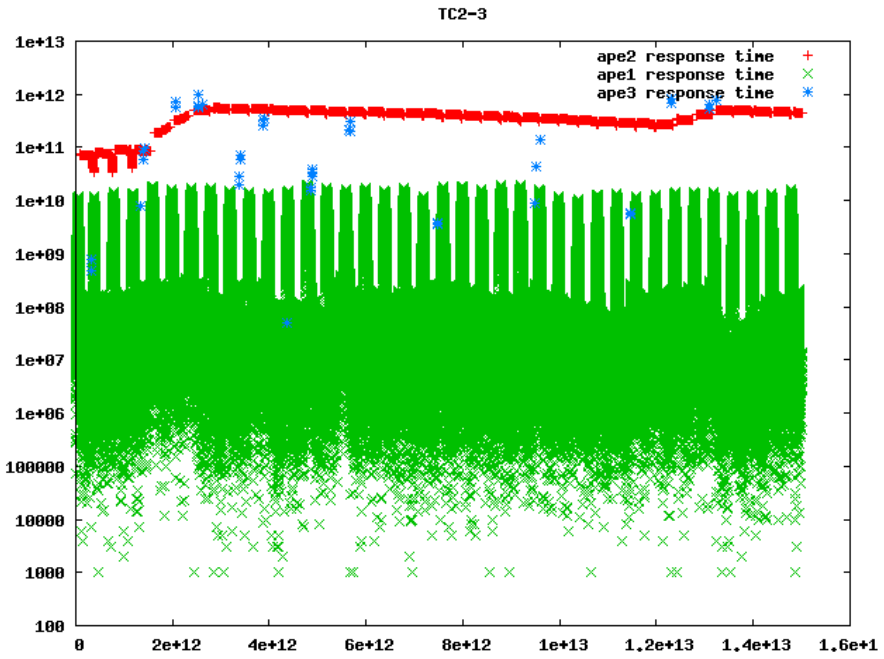

Figure H.32: *Idle response times for TC2.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

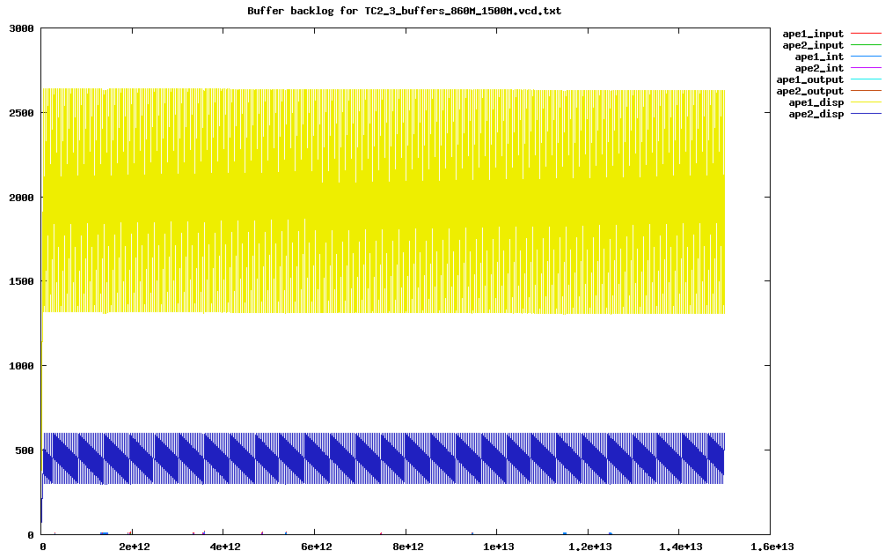

Figure H.33: *Buffer backlog for TC2.3 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

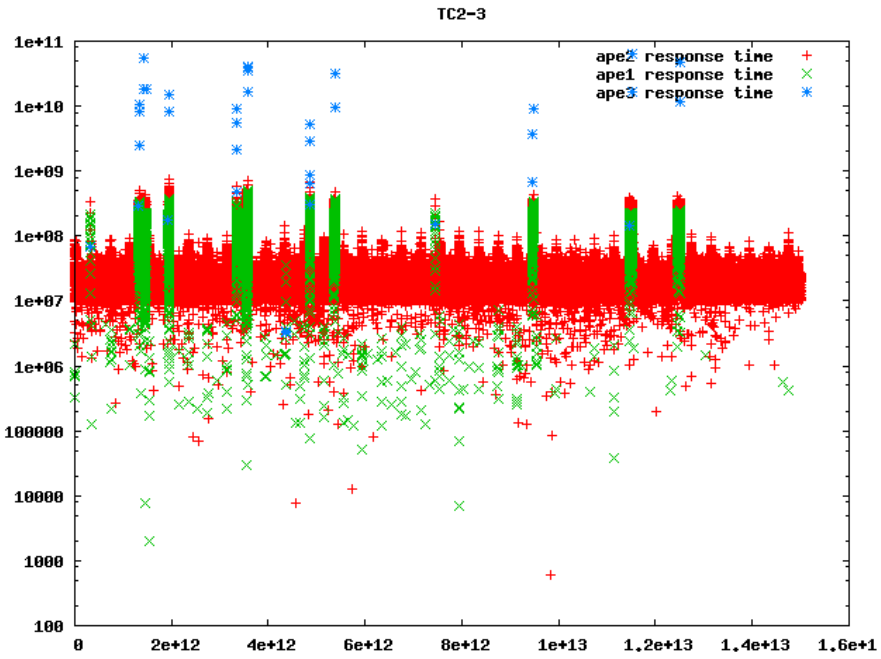

Figure H.34: *Idle response times for TC2.3 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.4 Case study 3

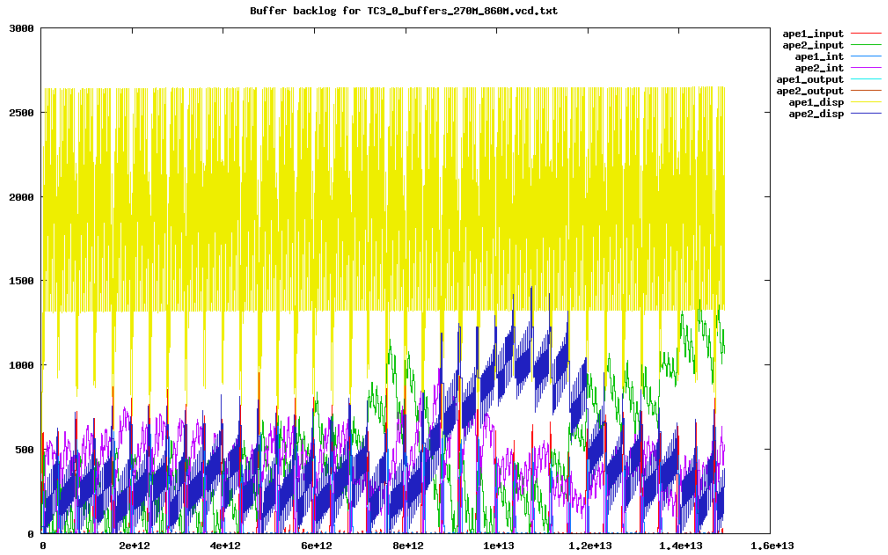

Figure H.35: *Buffer backlog for TC3.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

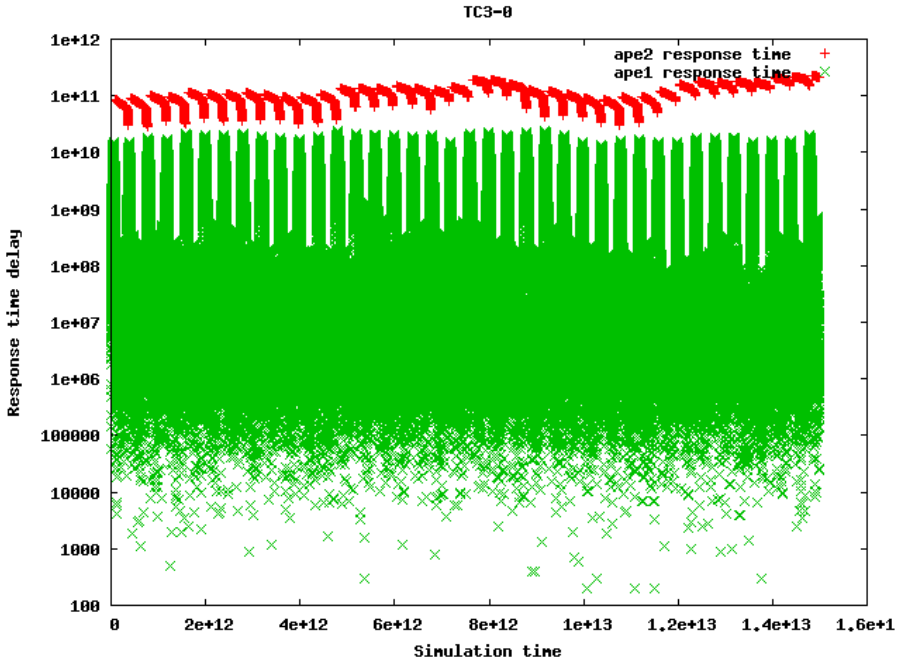

Figure H.36: *Idle response times for TC3.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

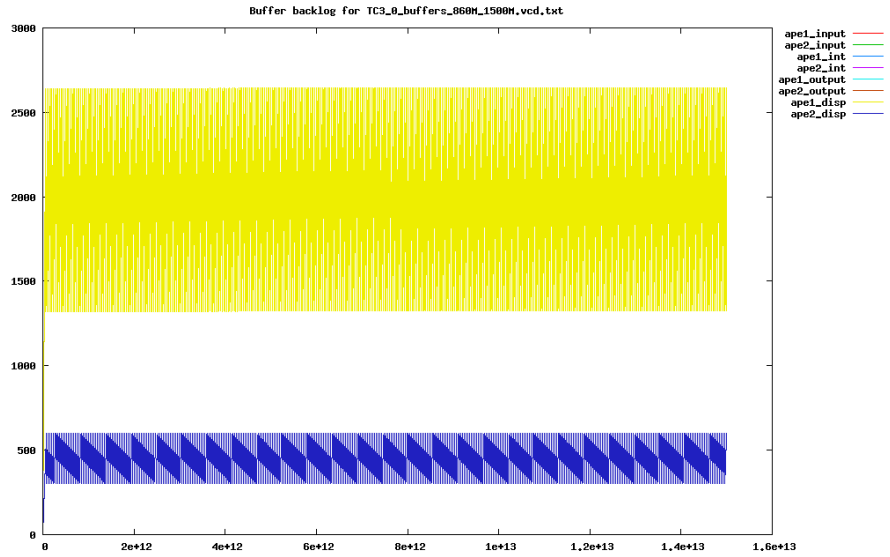

Figure H.37: *Buffer backlog for TC3.0 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

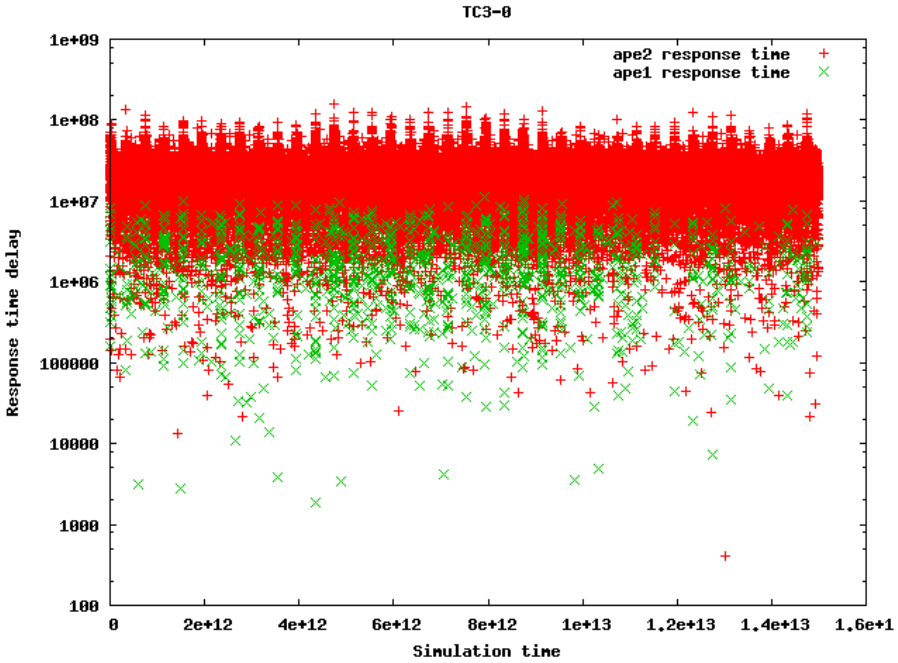

Figure H.38: *Idle response times for TC3.0 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

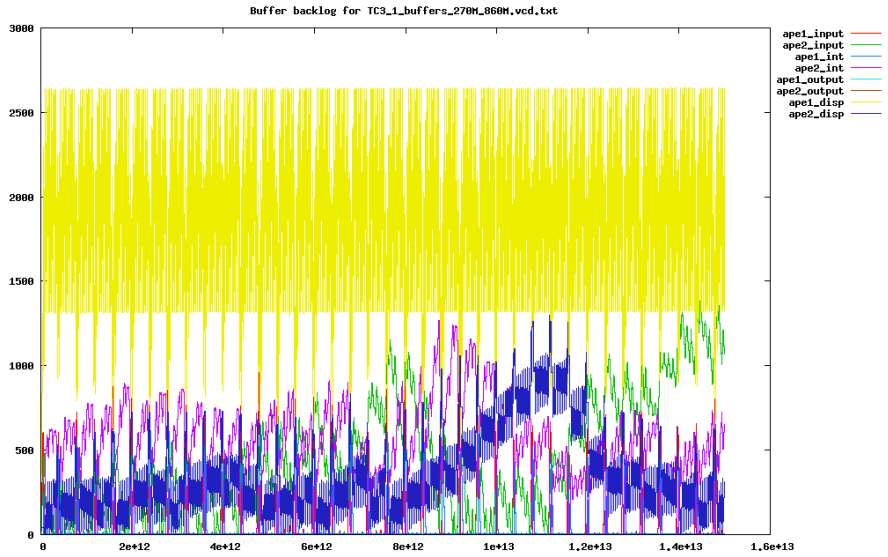

Figure H.39: *Buffer backlog for TC3.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

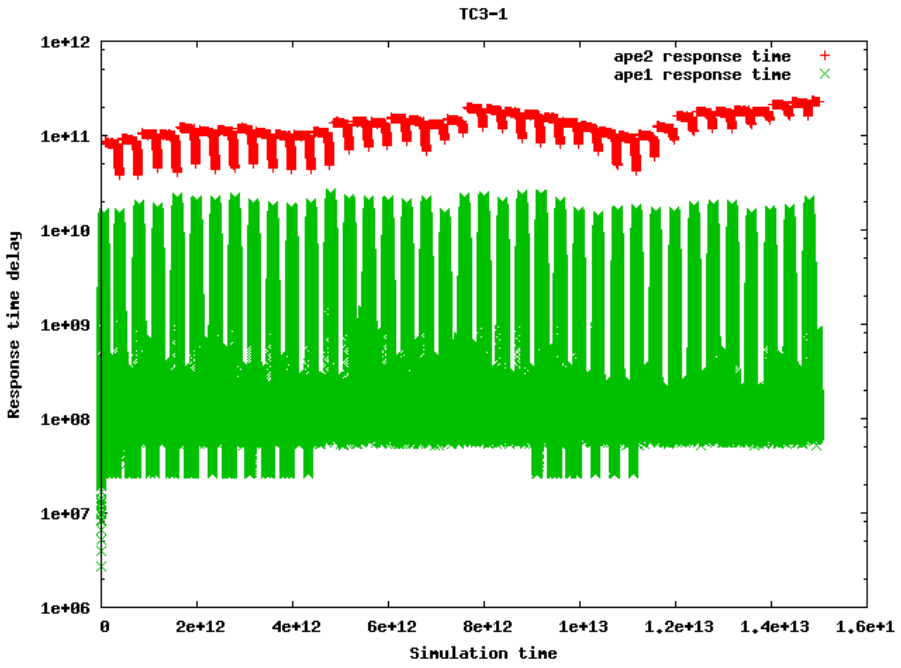

Figure H.40: *Idle response times for TC3.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

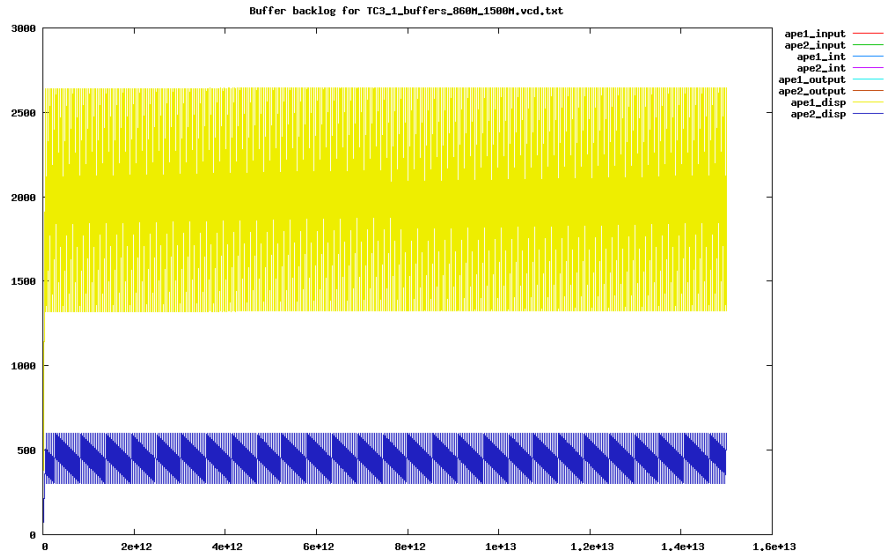

Figure H.41: *Buffer backlog for TC3.1 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

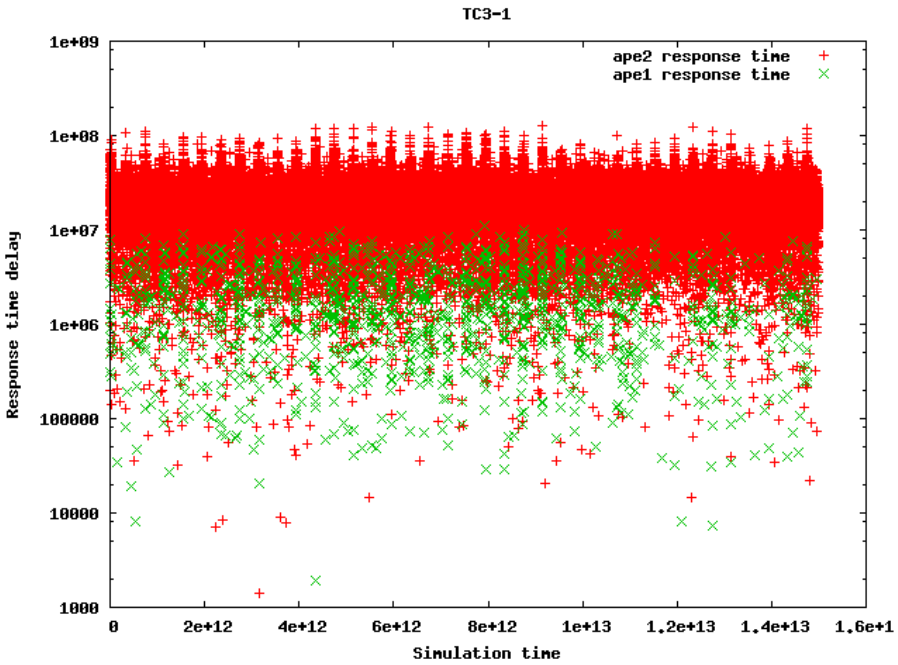

Figure H.42: *Idle response times for TC3.1 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

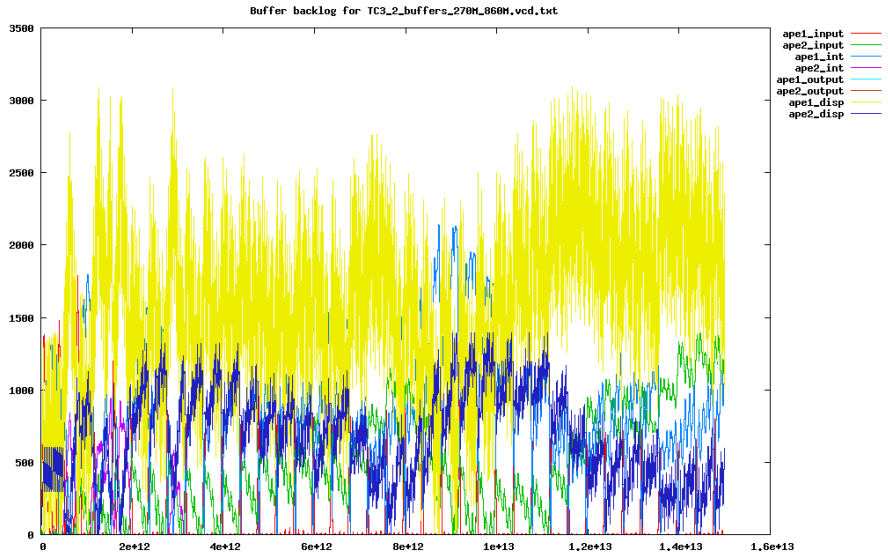

Figure H.43: *Buffer backlog for TC3.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

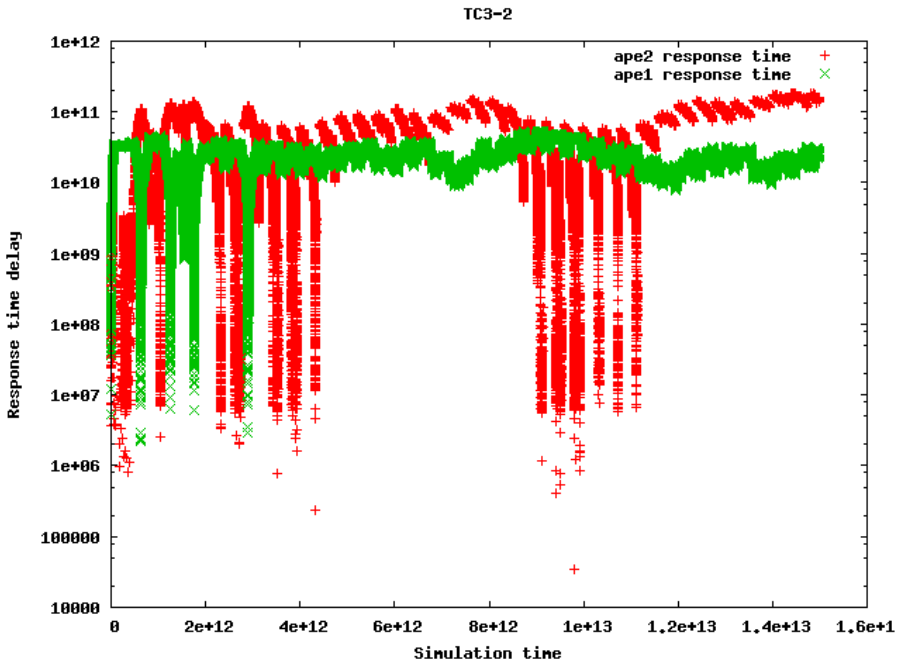

Figure H.44: *Idle response times for TC3.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

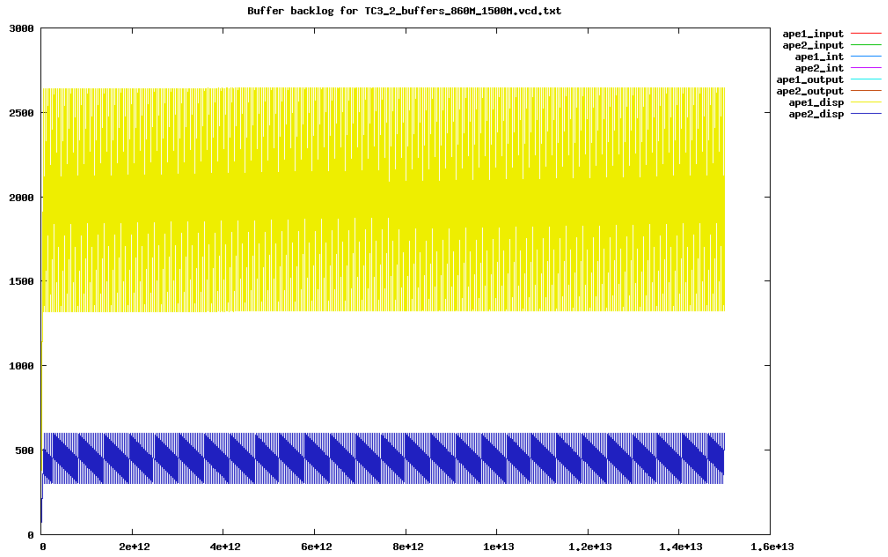

Figure H.45: *Buffer backlog for TC3.2 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

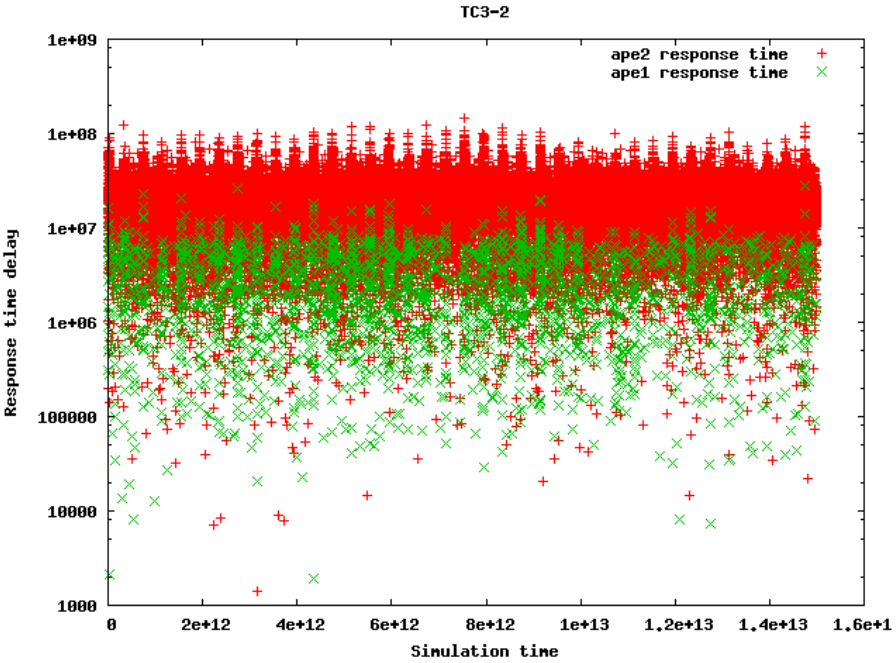

Figure H.46: *Idle response times for TC3.2 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

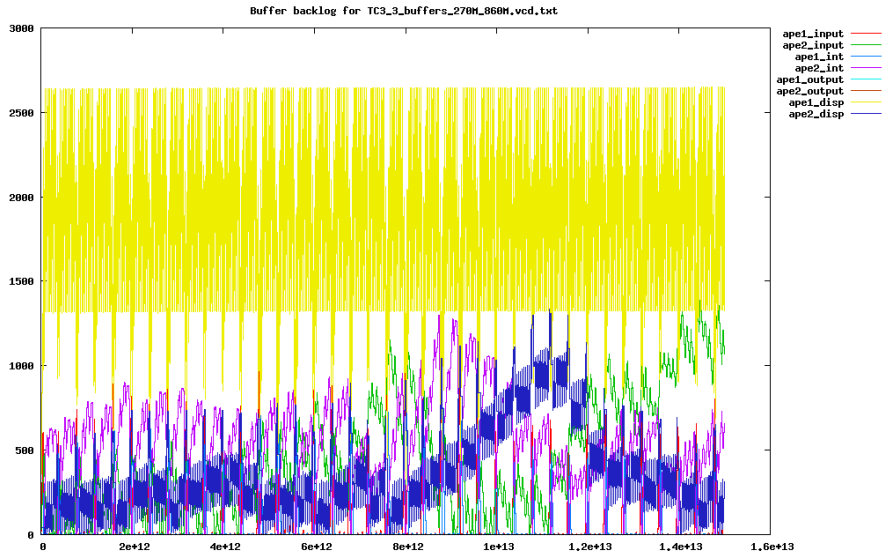

Figure H.47: *Buffer backlog for TC3.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

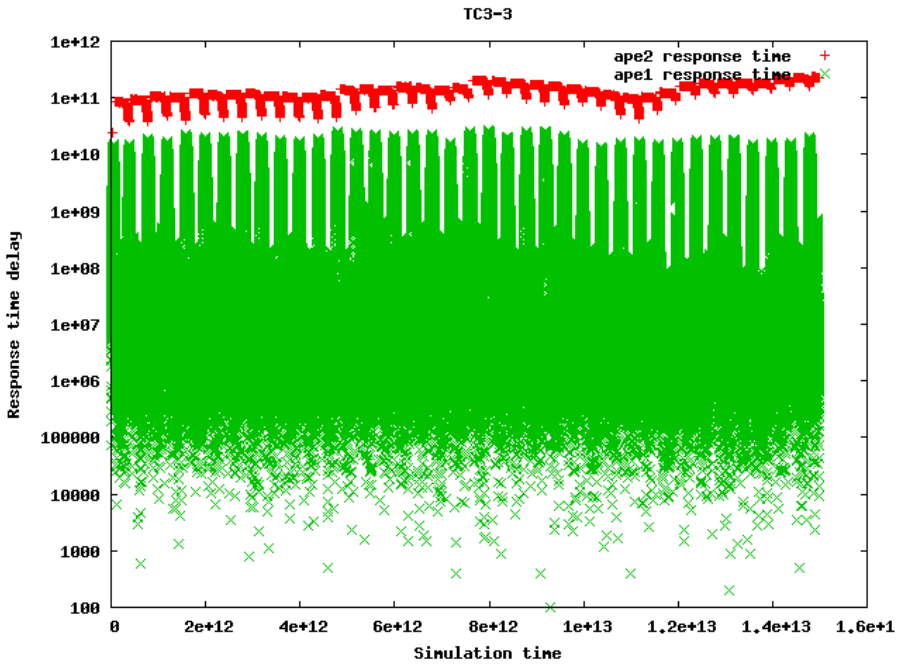

Figure H.48: *Idle response times for TC3.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

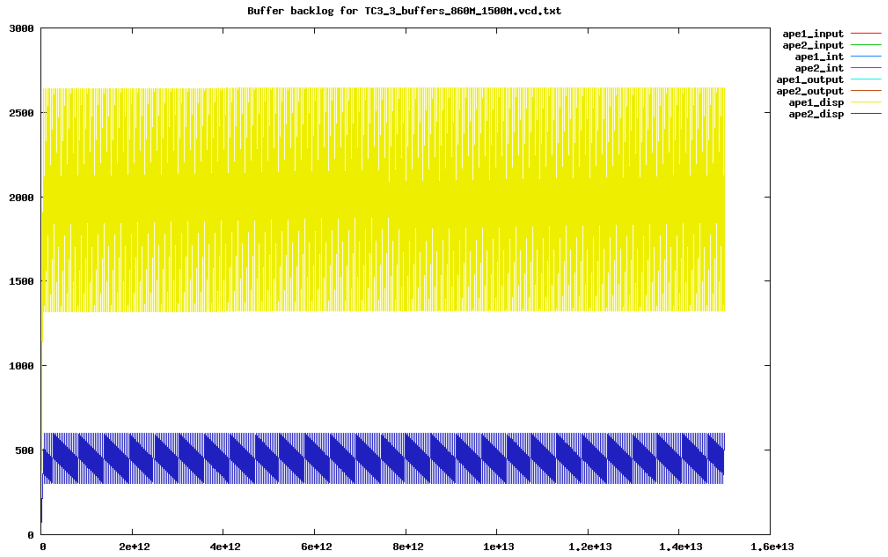

Figure H.49: *Buffer backlog for TC3.3 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

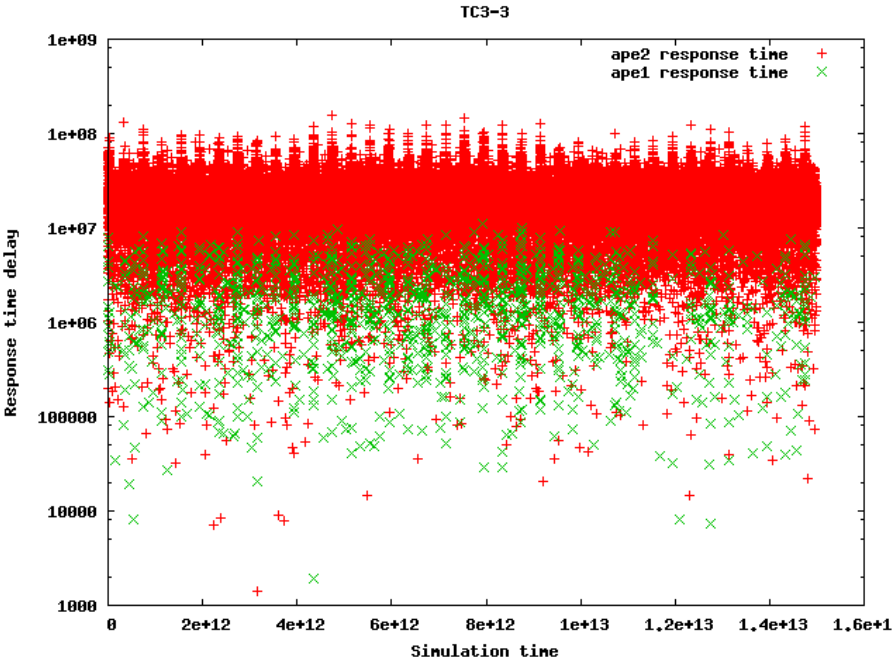

Figure H.50: *Idle response times for TC3.3 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.5 Case study 4

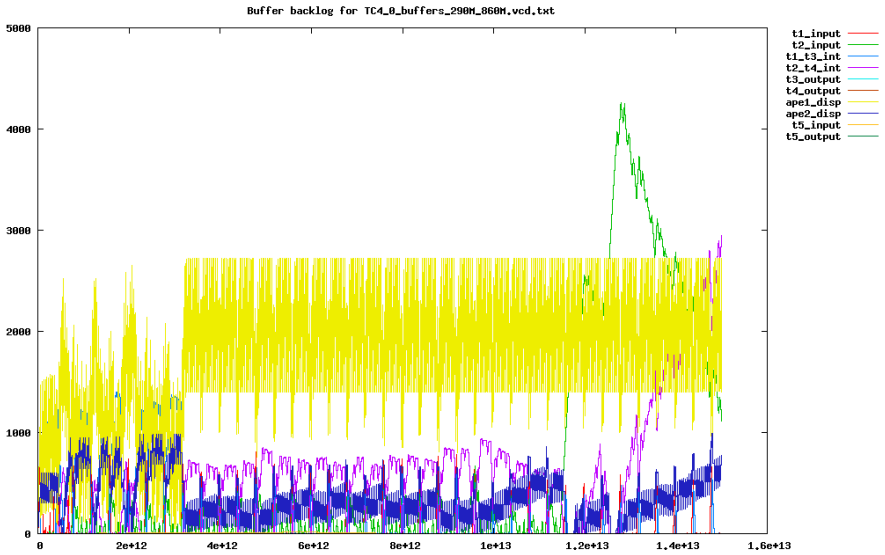

Figure H.51: *Buffer backlog for TC4.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

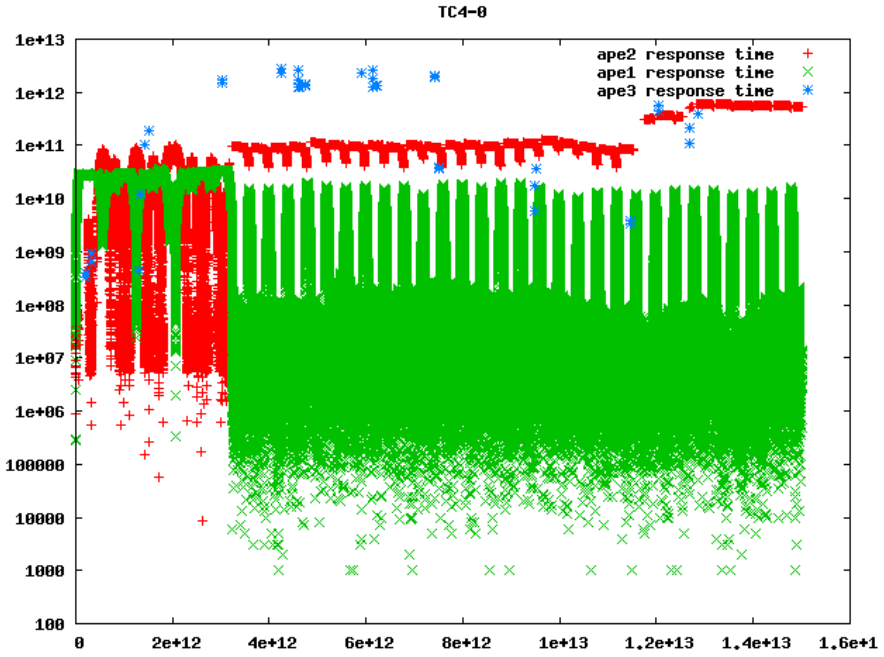

Figure H.52: *Idle response times for TC4.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

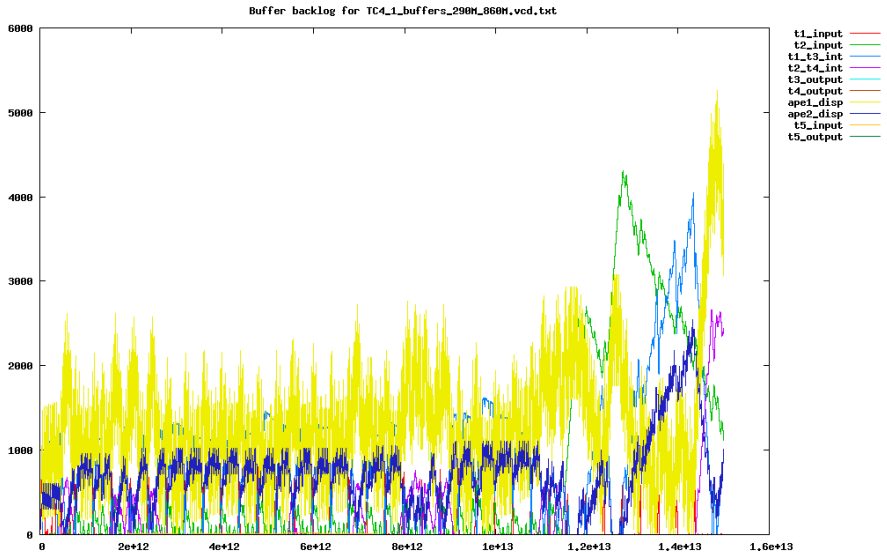

Figure H.53: *Buffer backlog for TC4.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

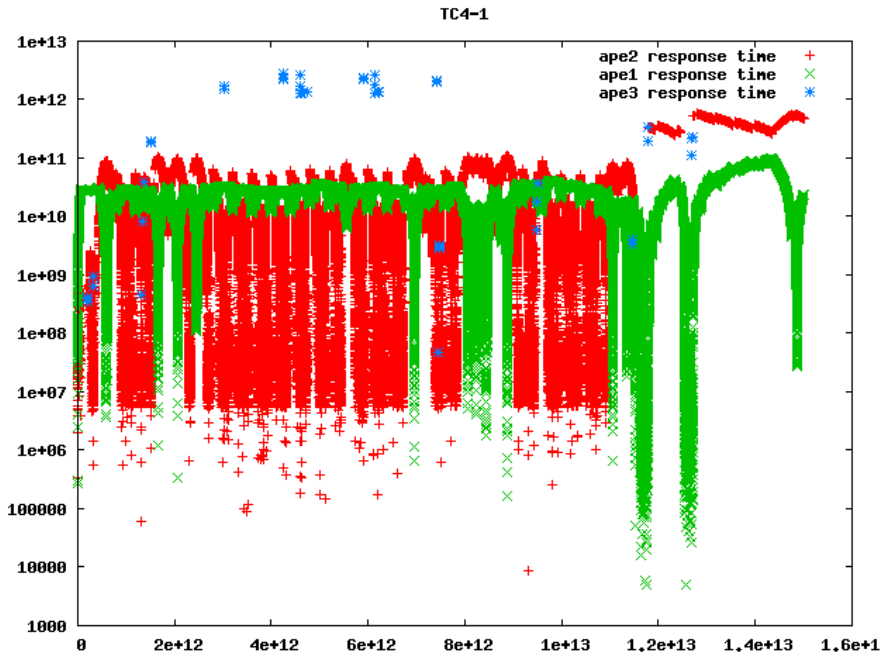

Figure H.54: *Idle response times for TC4.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

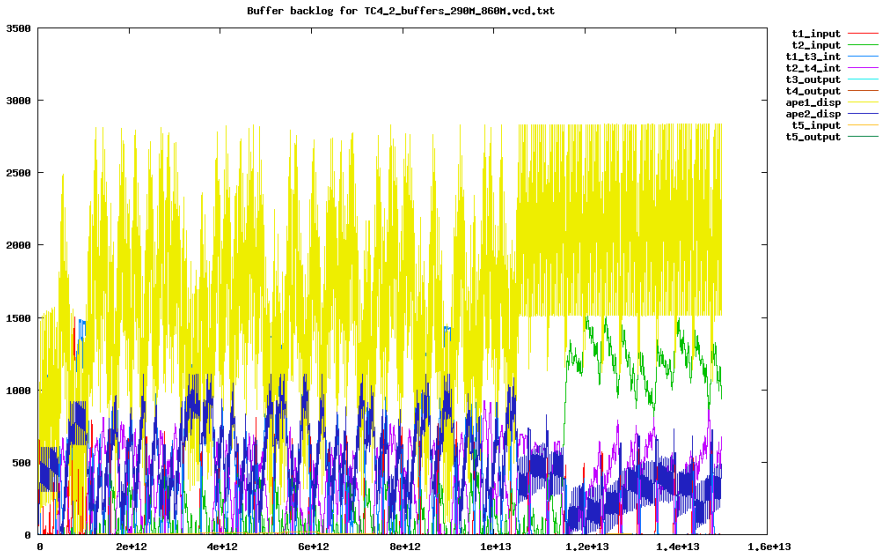

Figure H.55: *Buffer backlog for TC4.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

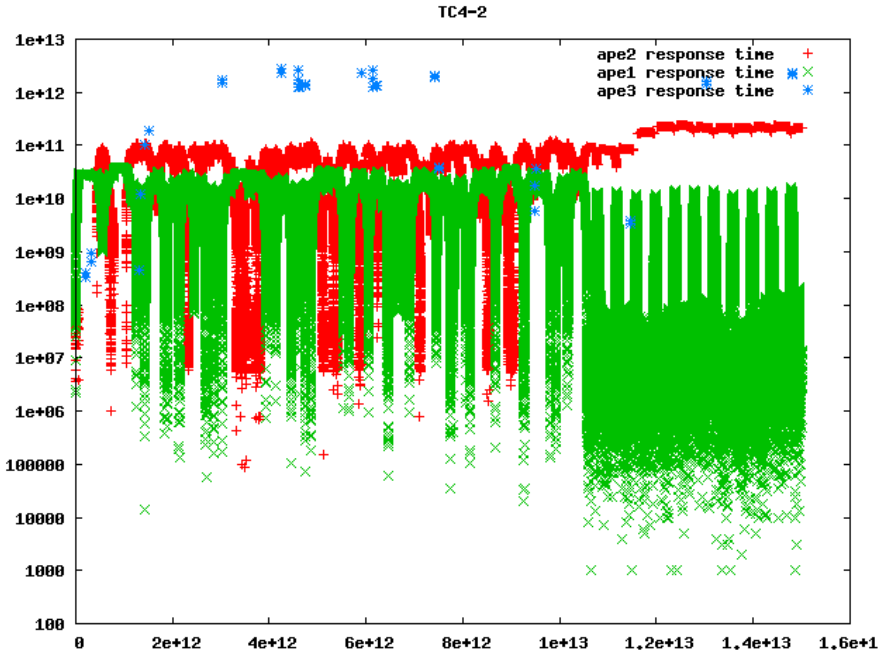

Figure H.56: *Idle response times for TC4.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

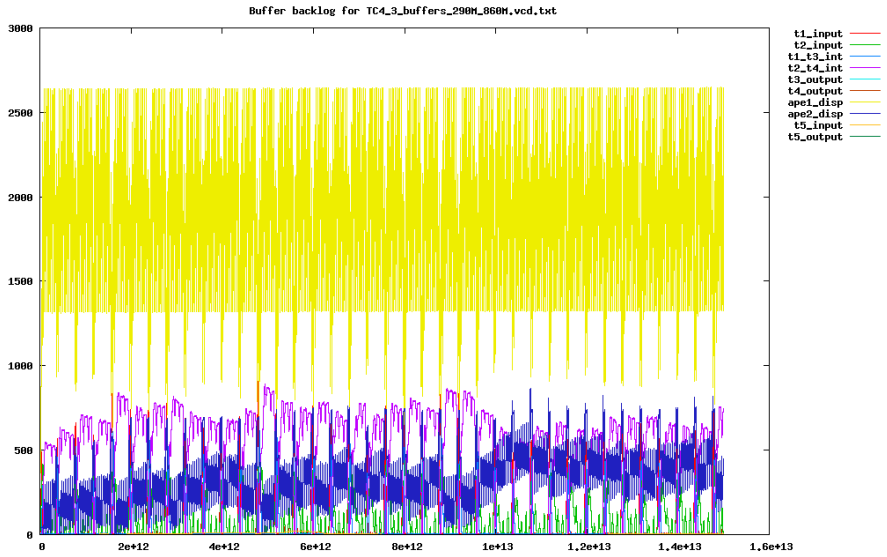

Figure H.57: *Buffer backlog for TC4.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

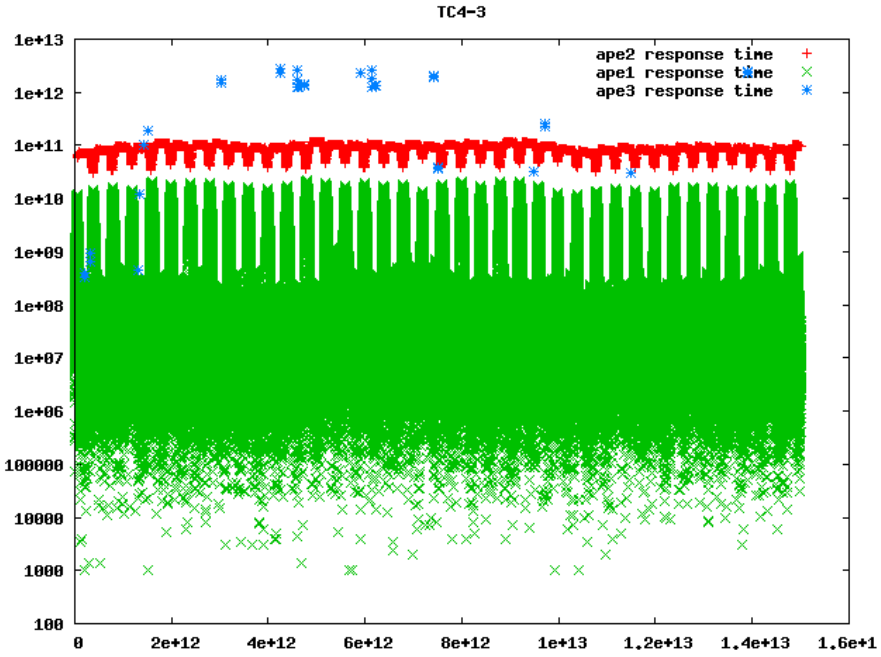

Figure H.58: *Idle response times for TC4.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

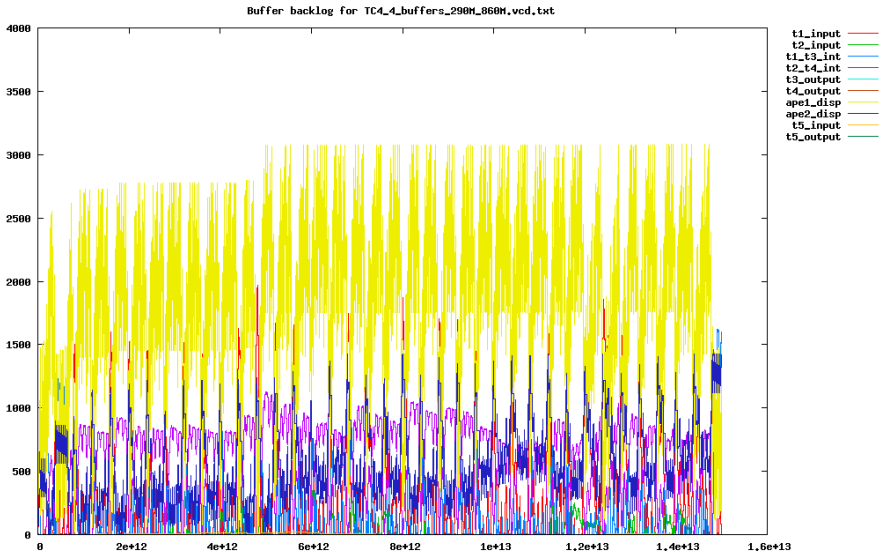

Figure H.59: *Buffer backlog for TC4.4 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

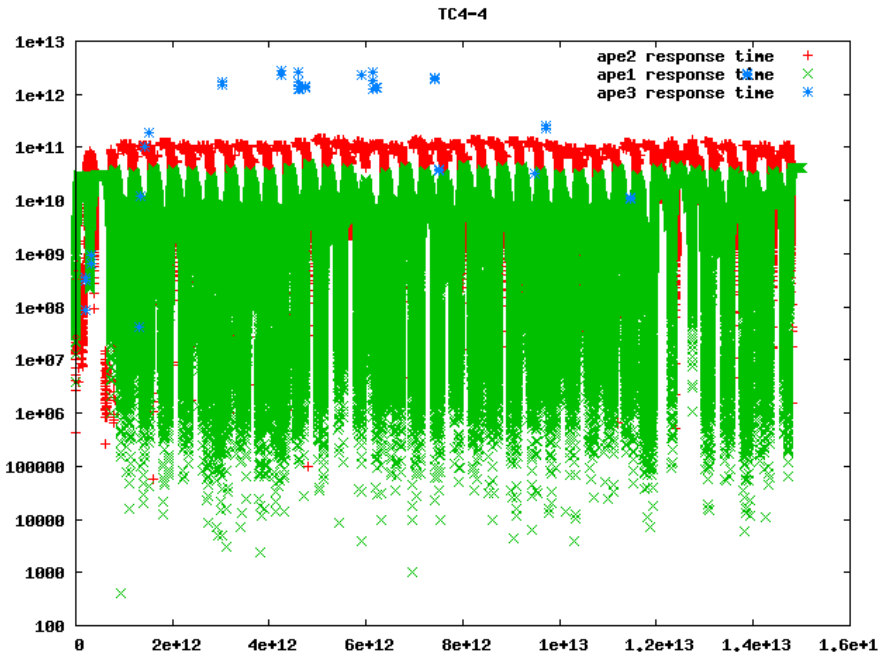

Figure H.60: *Idle response times for TC4.4 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.6 Case study 5

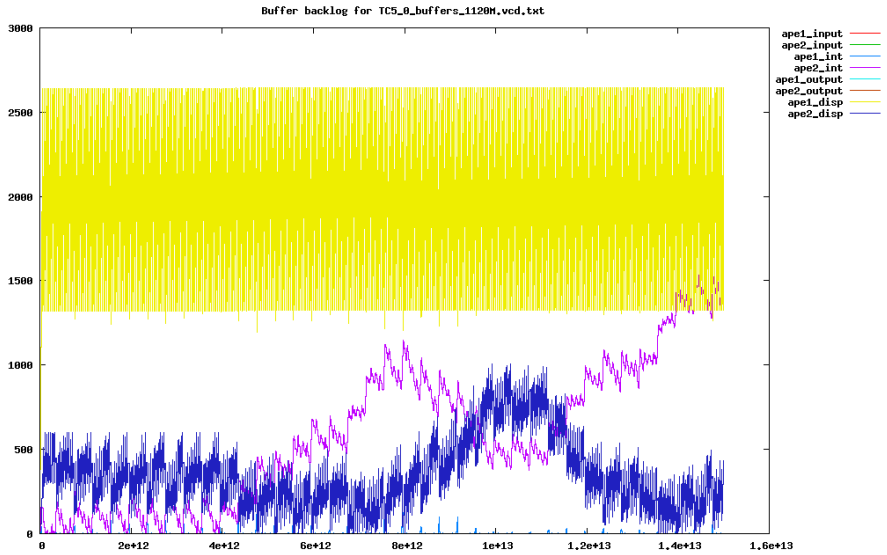

Figure H.61: *Buffer backlog for TC5.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

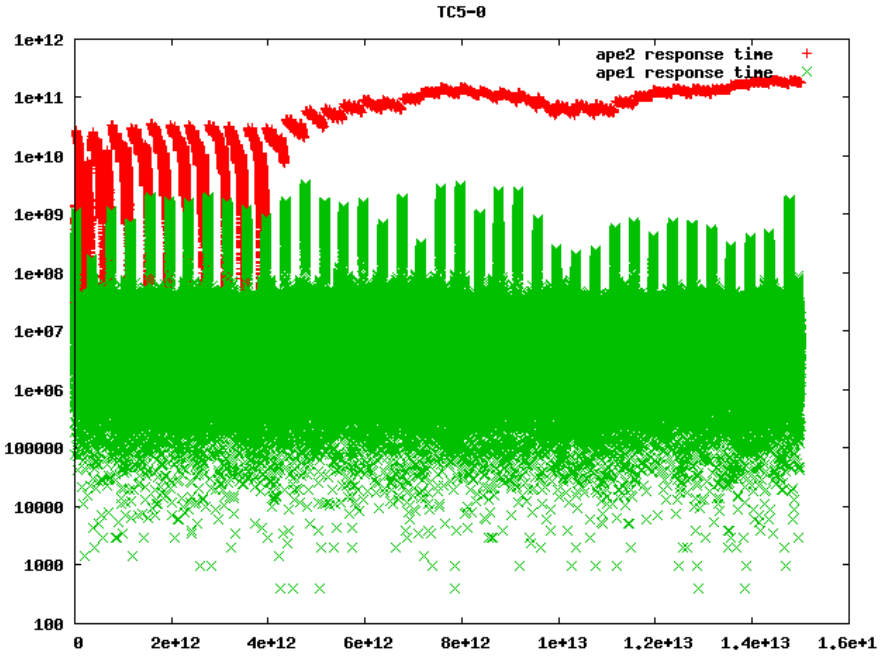

Figure H.62: *Idle response times for TC5.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

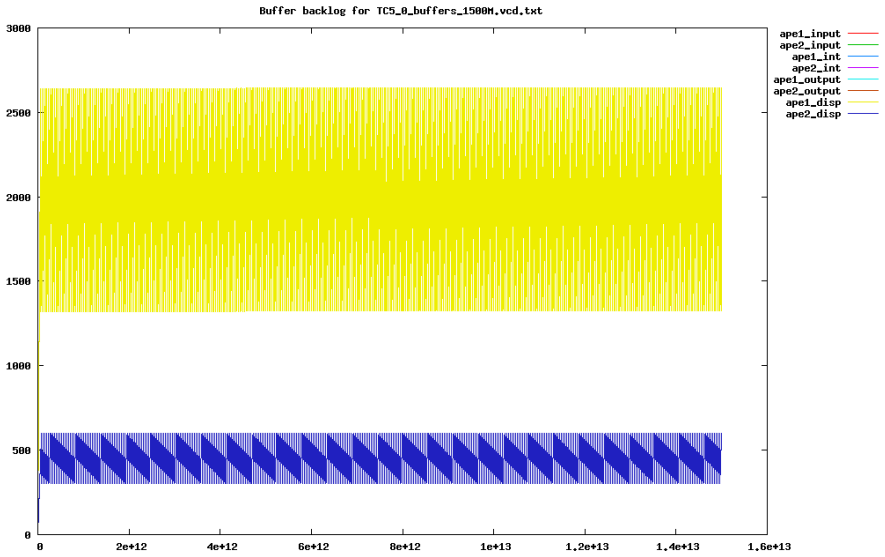

Figure H.63: *Buffer backlog for TC5.0 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

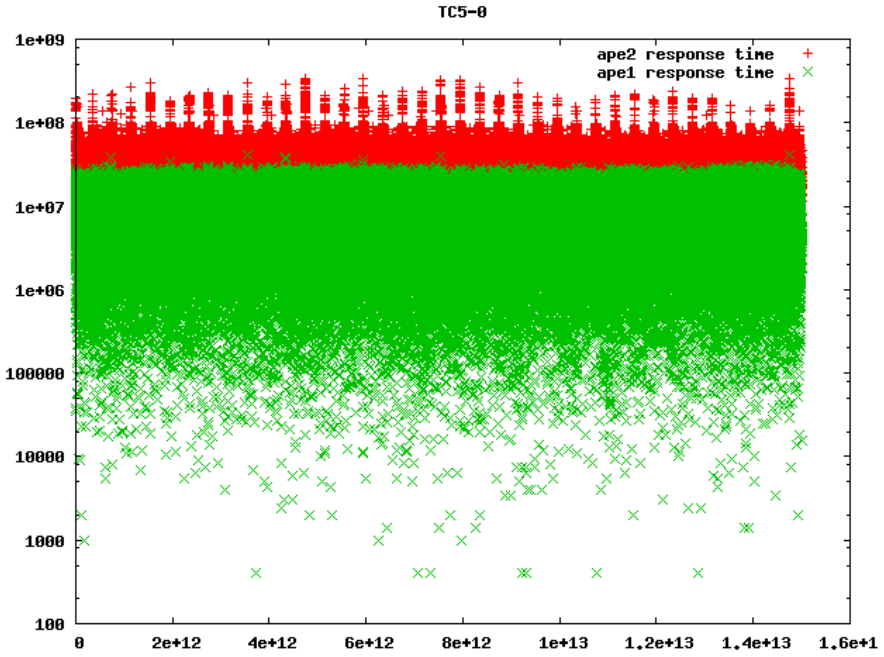

Figure H.64: *Idle response times for TC5.0 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

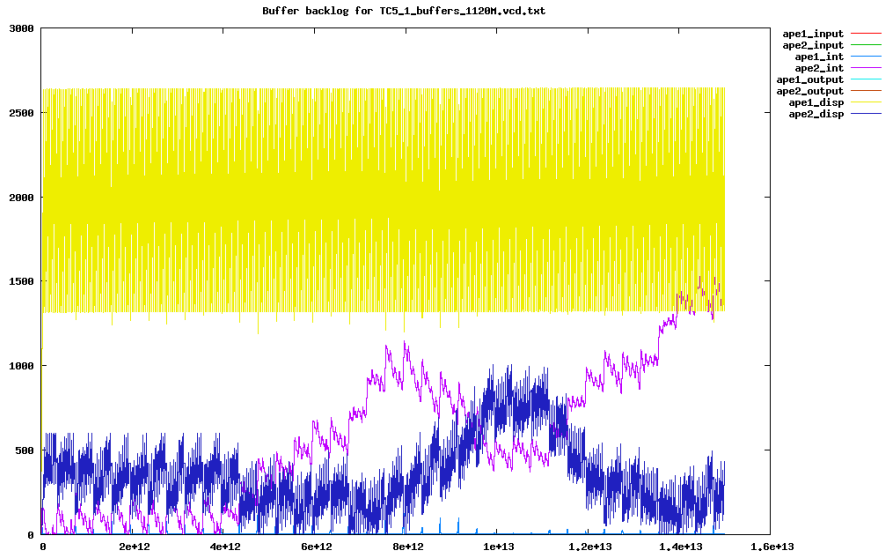

Figure H.65: *Buffer backlog for TC5.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

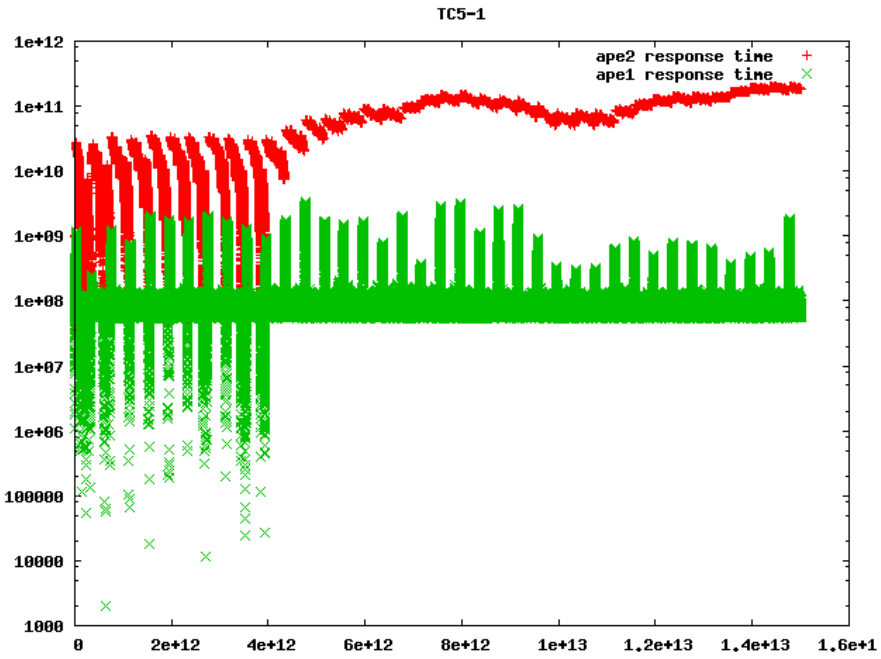

Figure H.66: *Idle response times for TC5.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

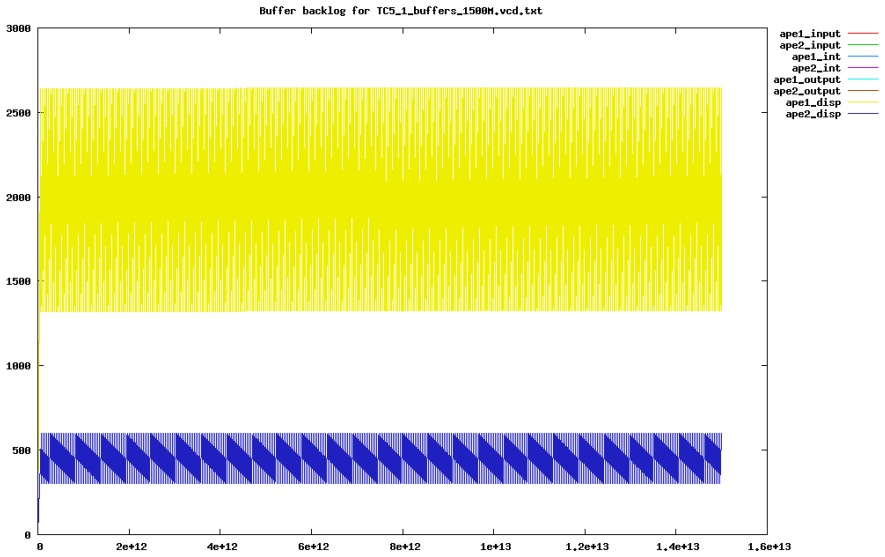

Figure H.67: *Buffer backlog for TC5.1 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

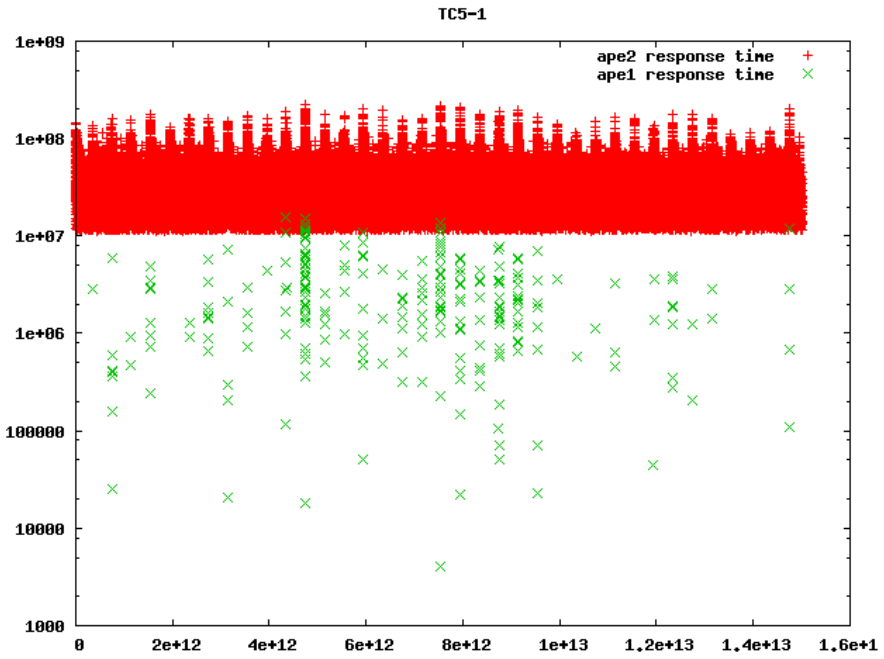

Figure H.68: *Idle response times for TC5.1 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

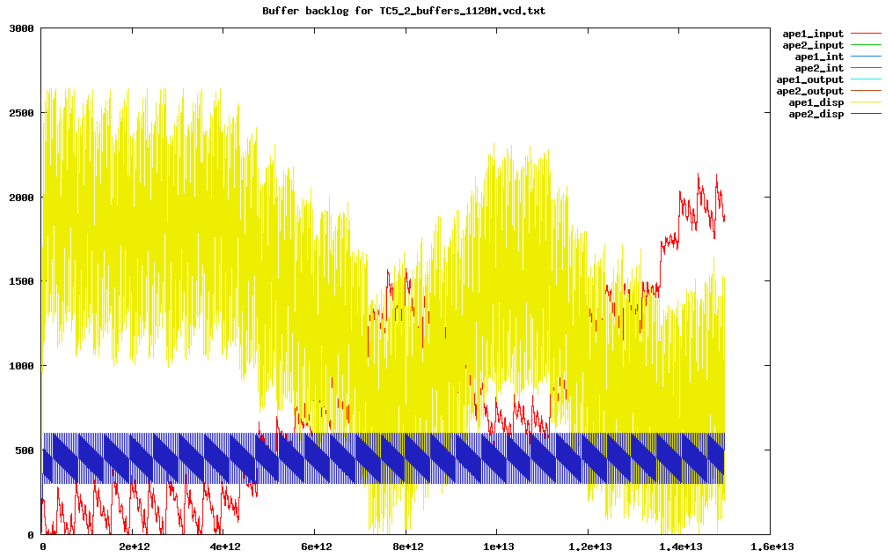

Figure H.69: *Buffer backlog for TC5.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

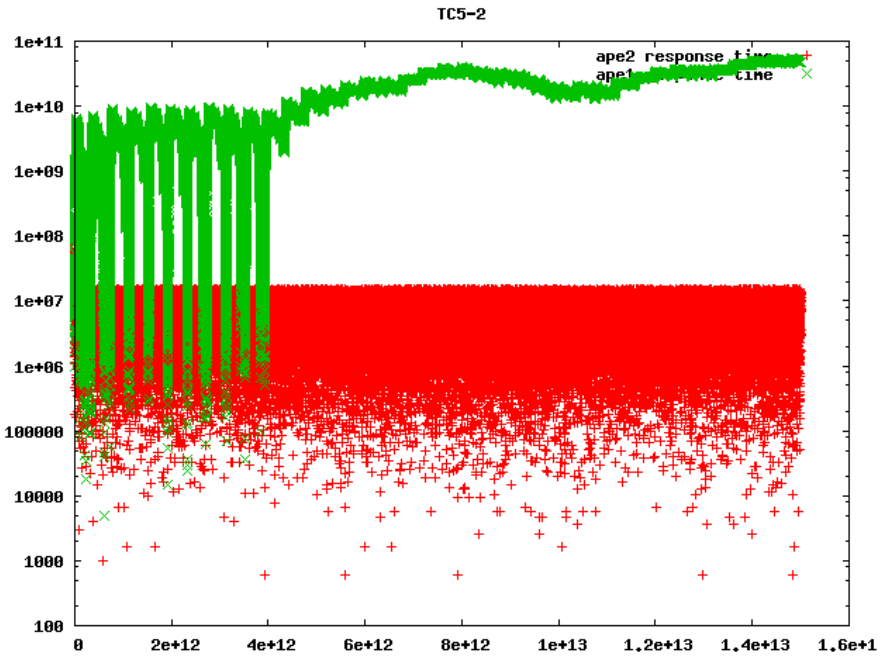

Figure H.70: *Idle response times for TC5.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

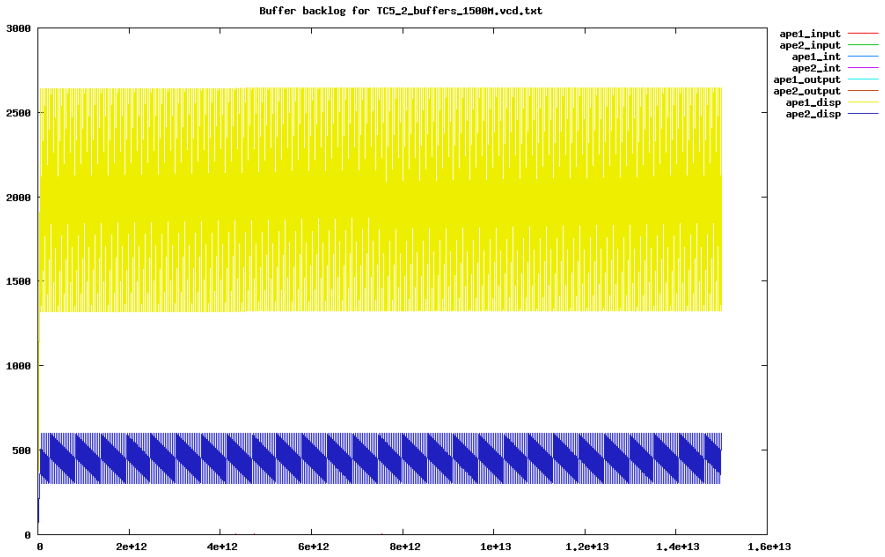

Figure H.71: *Buffer backlog for TC5.2 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

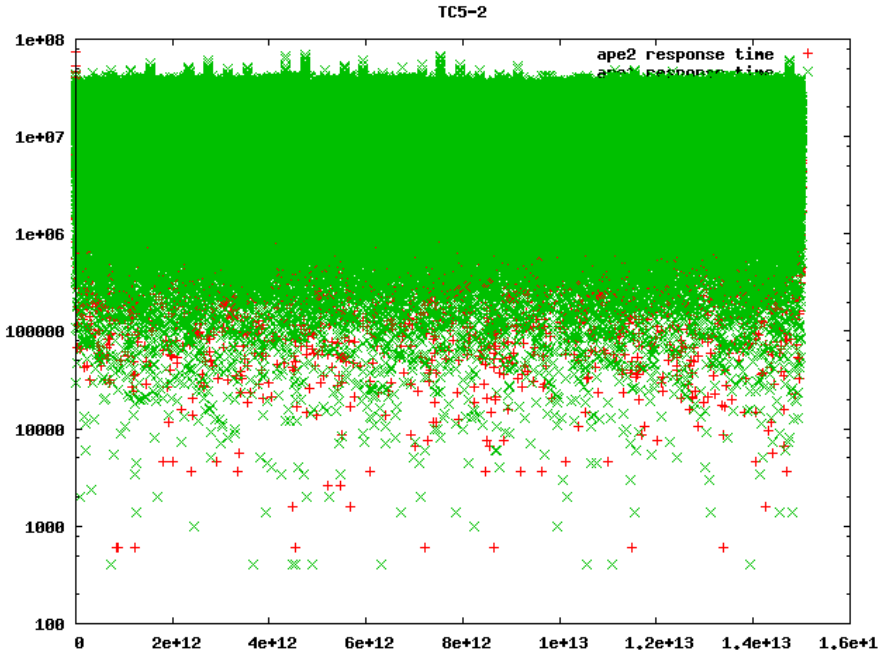

Figure H.72: *Idle response times for TC5.2 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

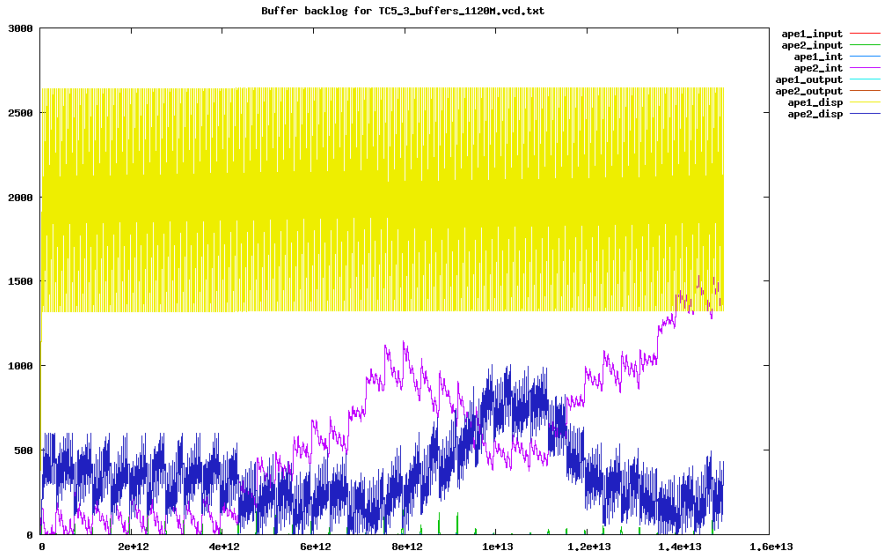

Figure H.73: *Buffer backlog for TC5.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

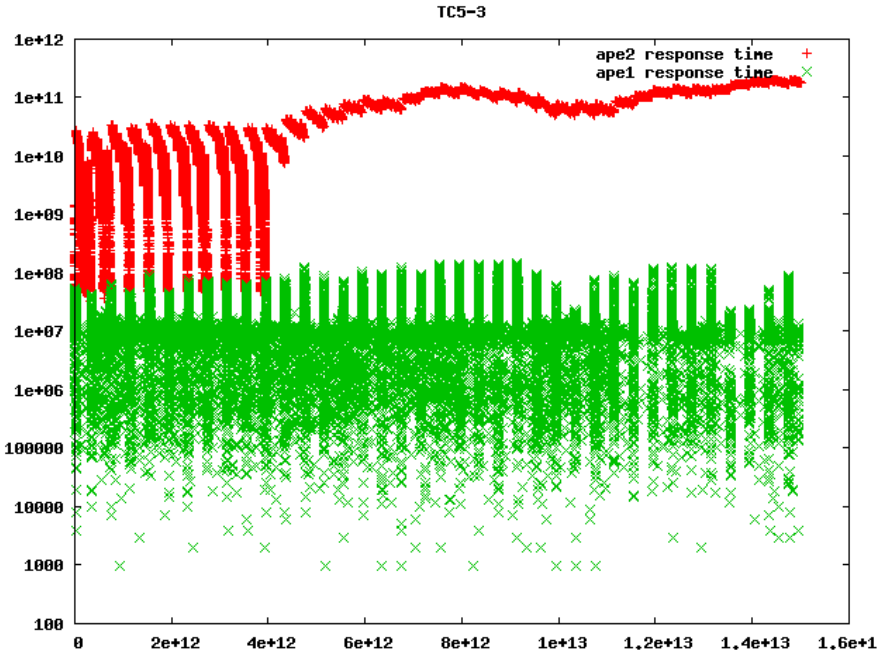

Figure H.74: *Idle response times for TC5.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

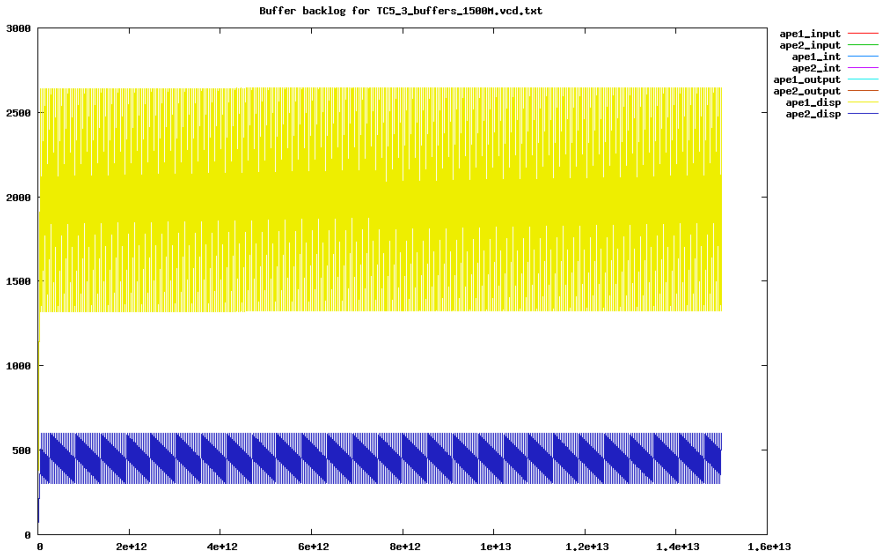

Figure H.75: *Buffer backlog for TC5.3 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

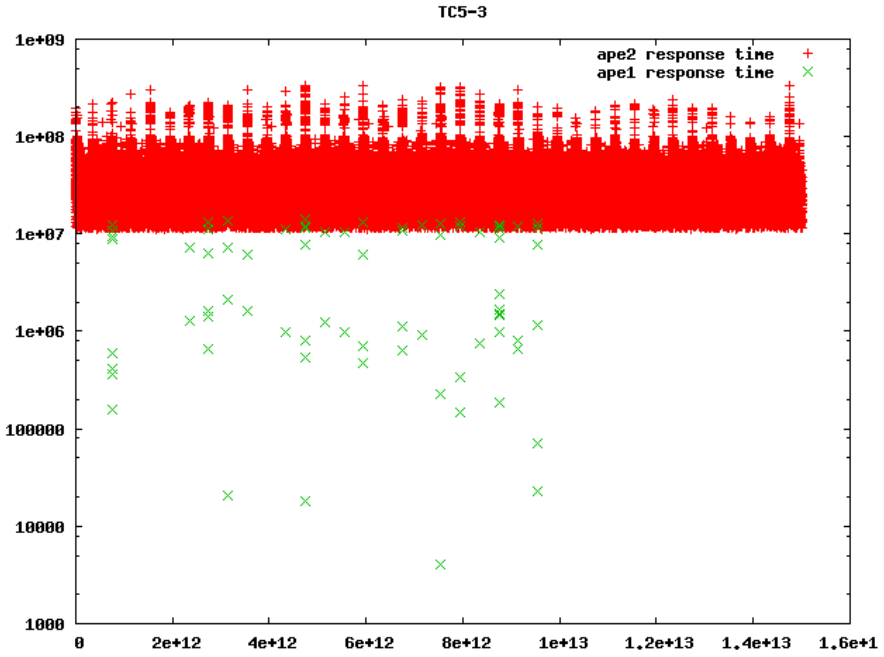

Figure H.76: *Idle response times for TC5.3 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.7 Case study 6

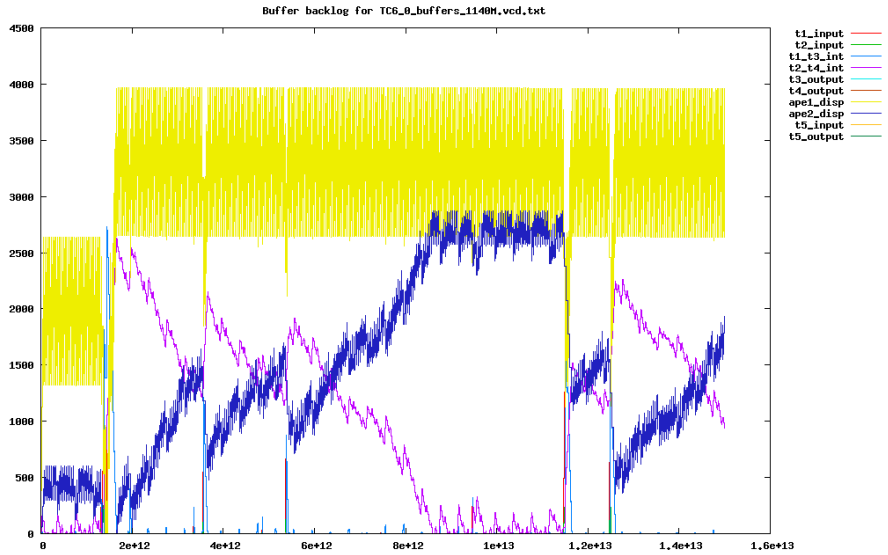

Figure H.77: *Buffer backlog for TC6.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

Figure H.78: *Idle response times for TC6.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

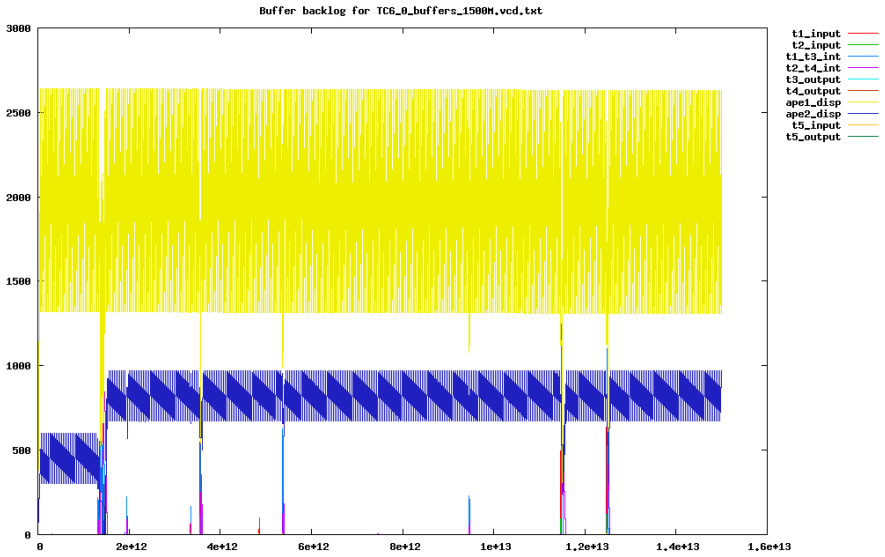

Figure H.79: *Buffer backlog for TC6.0 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

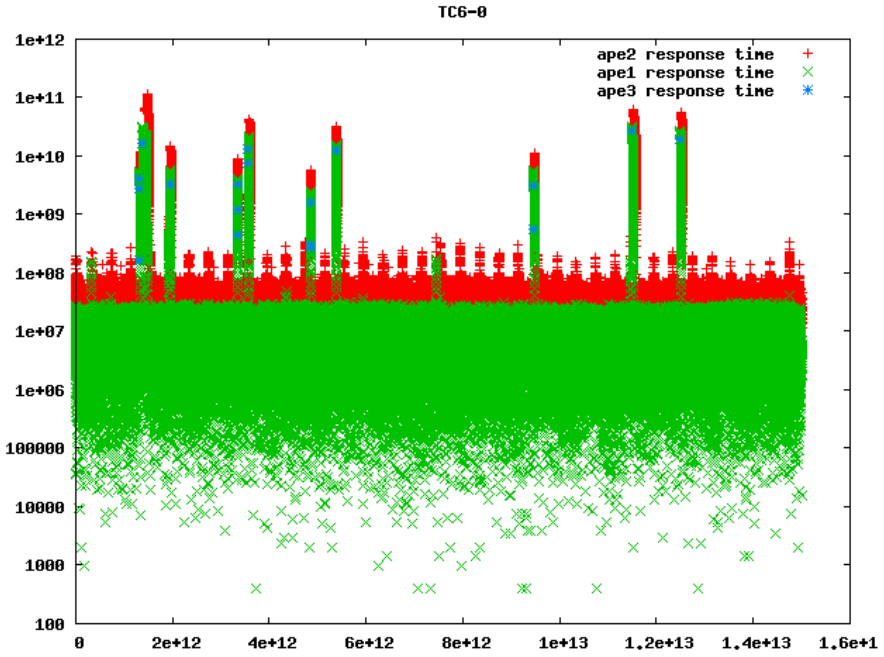

Figure H.80: *Idle response times for TC6.0 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

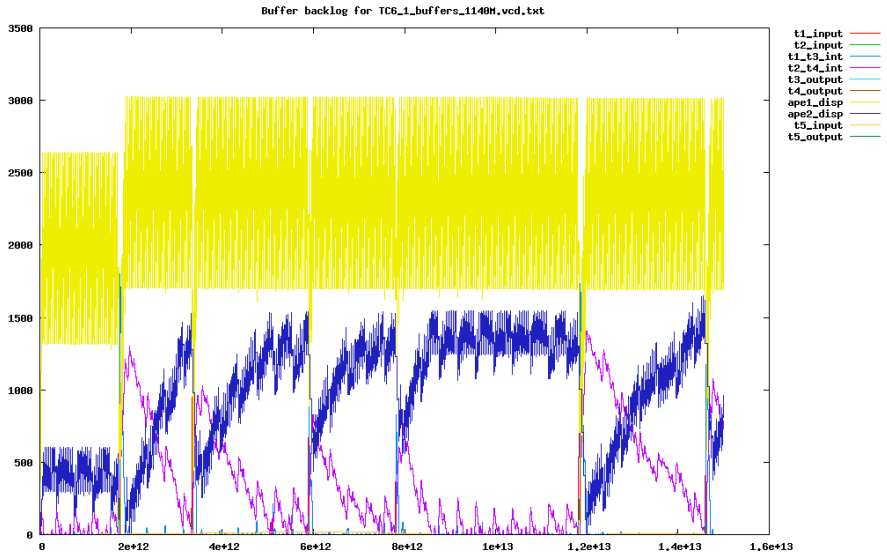

Figure H.81: *Buffer backlog for TC6.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

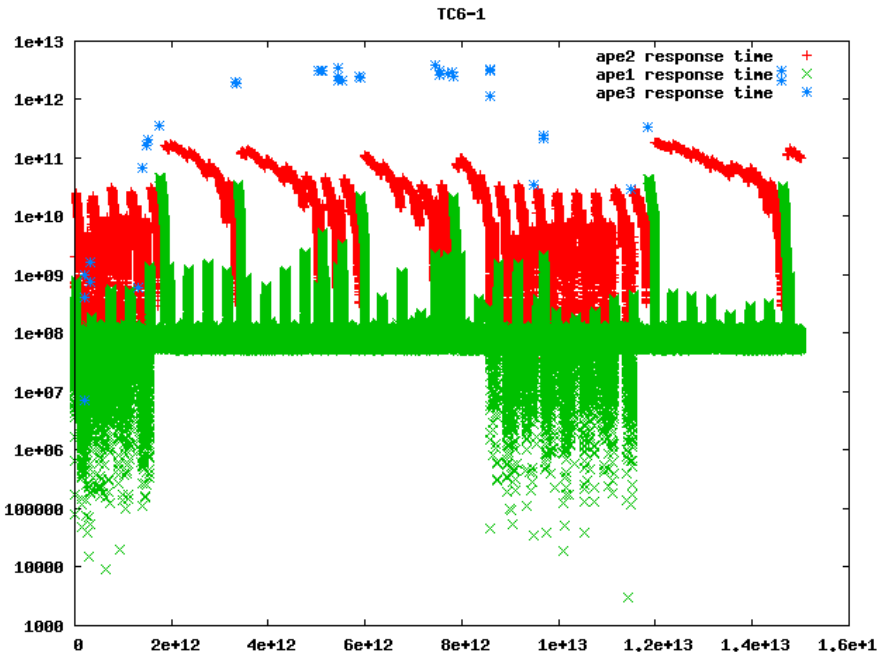

Figure H.82: *Idle response times for TC6.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

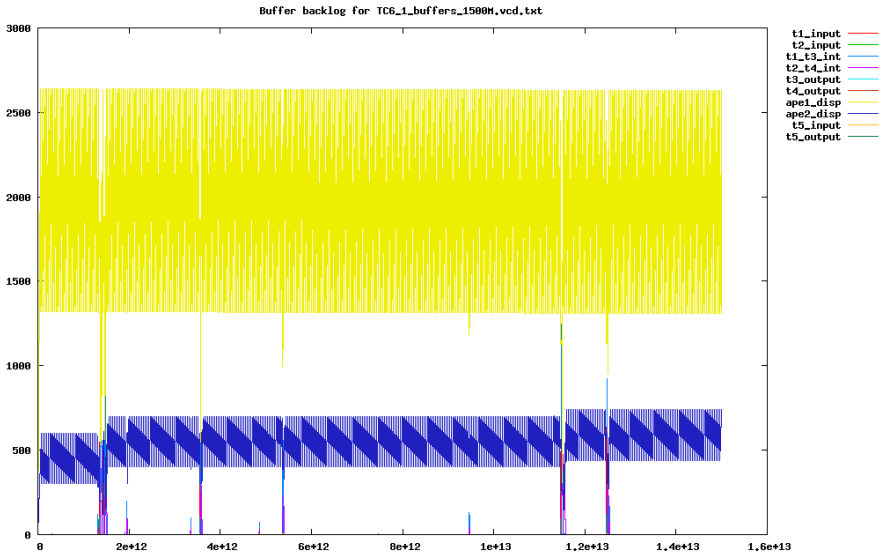

Figure H.83: *Buffer backlog for TC6.1 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

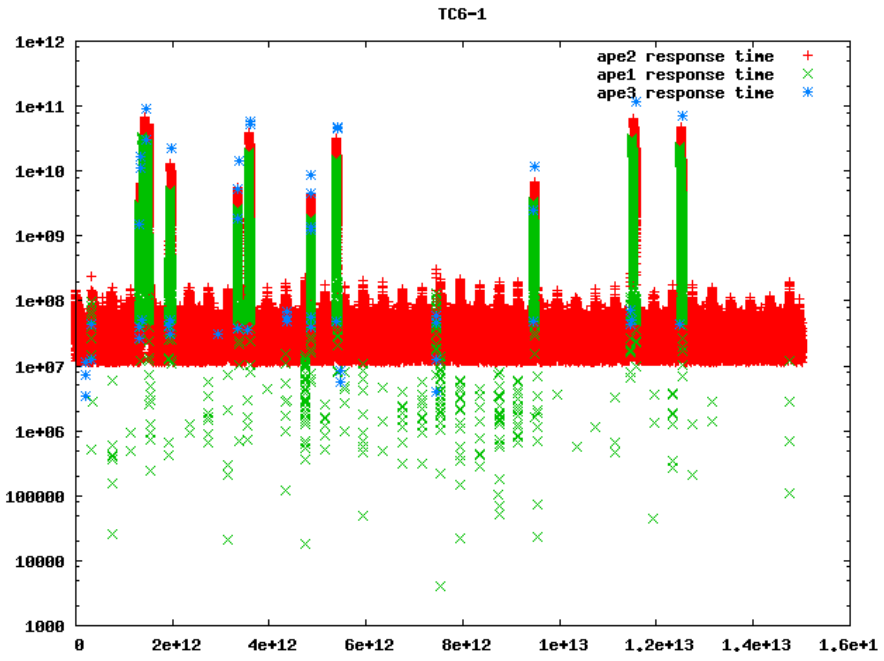

Figure H.84: *Idle response times for TC6.1 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

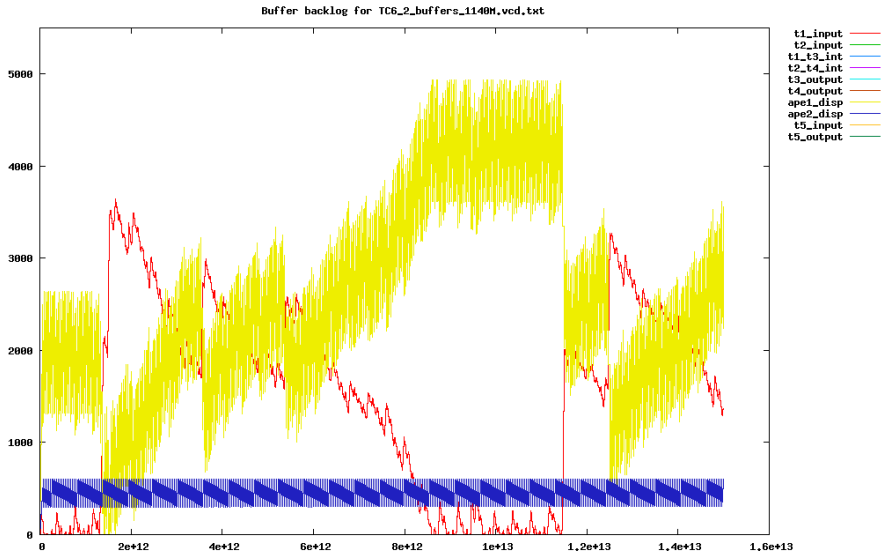

Figure H.85: *Buffer backlog for TC6.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

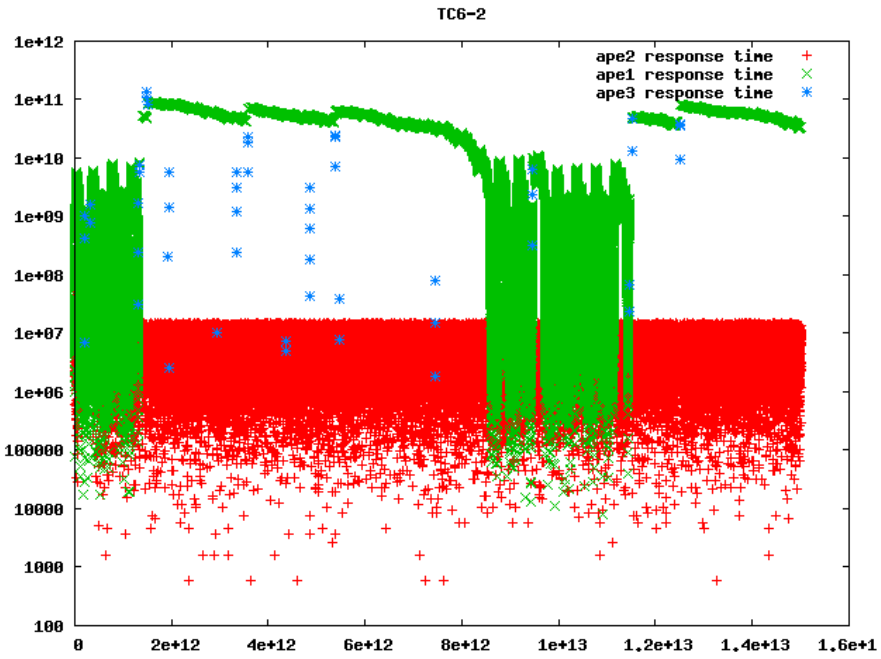

Figure H.86: *Idle response times for TC6.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

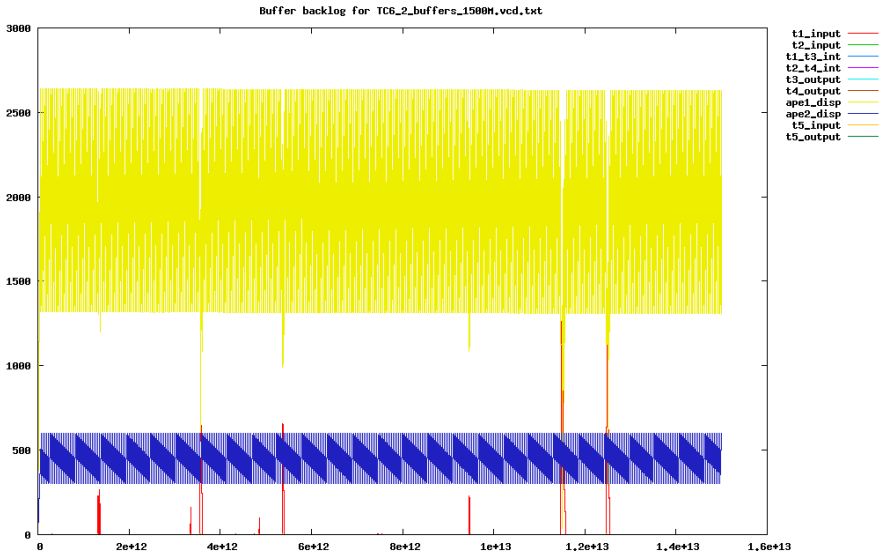

Figure H.87: *Buffer backlog for TC6.2 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

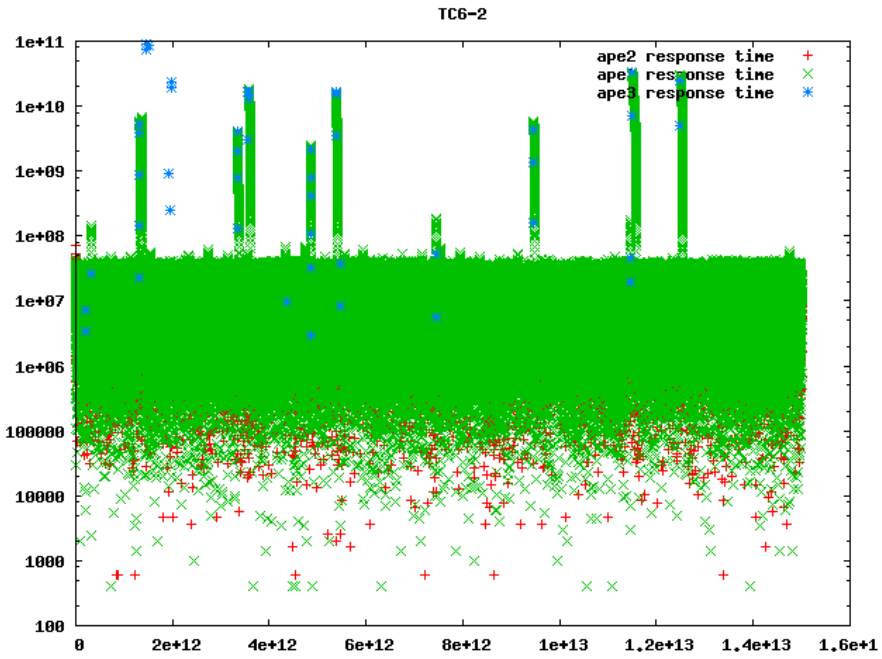

Figure H.88: *Idle response times for TC6.2 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

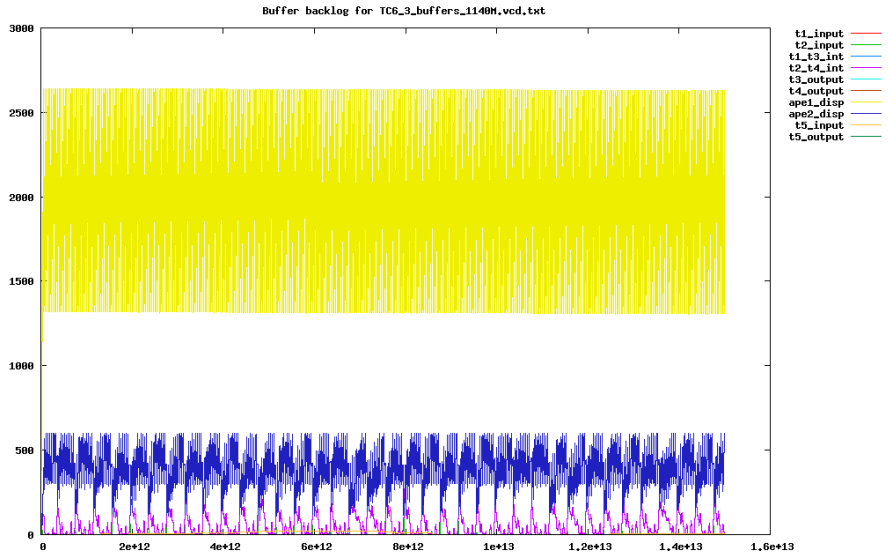

Figure H.89: *Buffer backlog for TC6.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

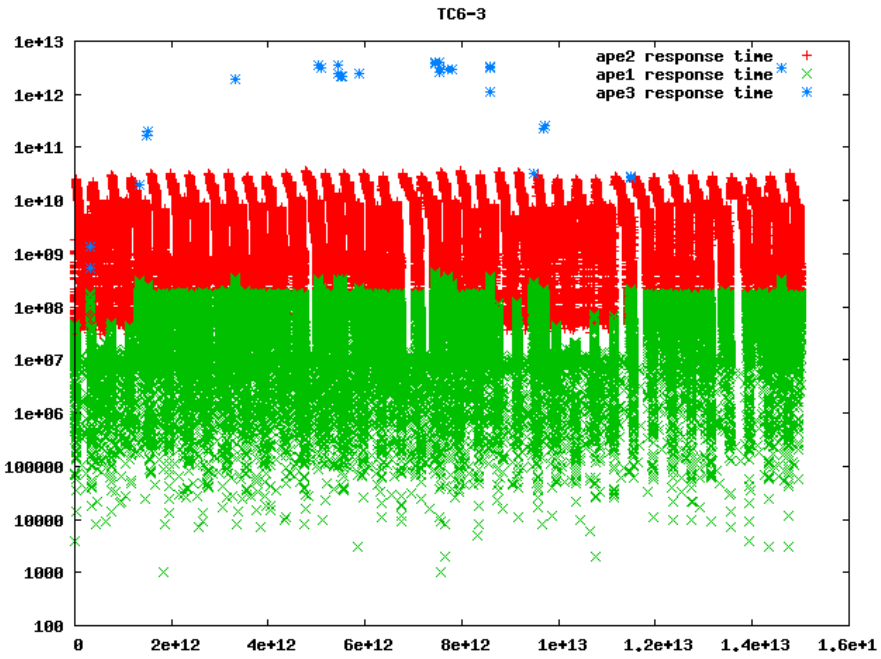

Figure H.90: *Idle response times for TC6.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

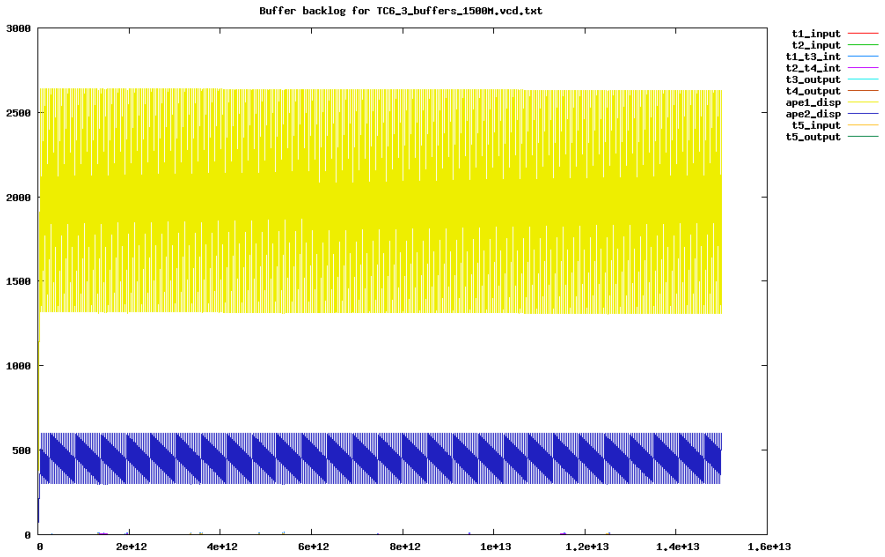

Figure H.91: *Buffer backlog for TC6.3 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

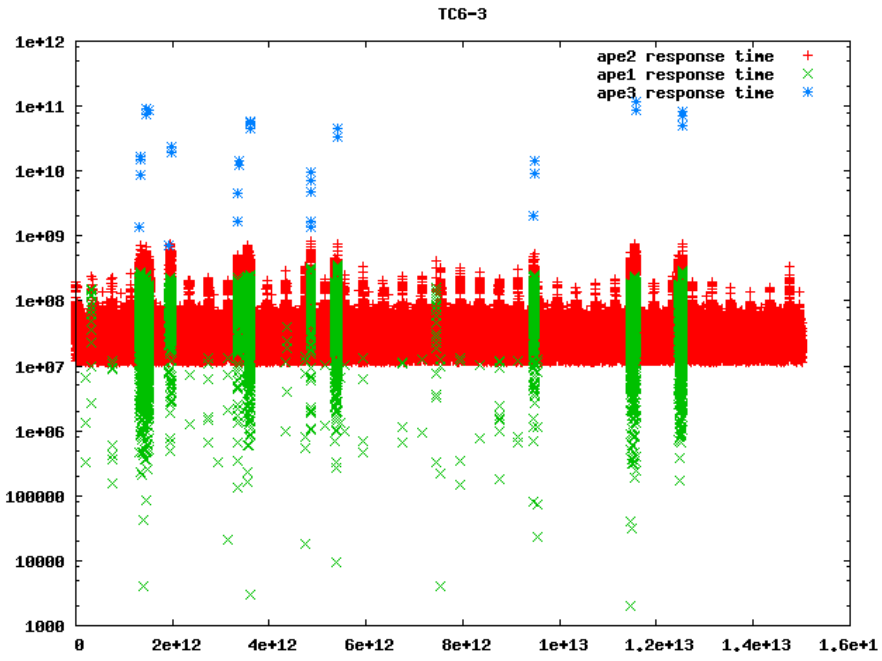

Figure H.92: *Idle response times for TC6.3 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.8 Case study 7

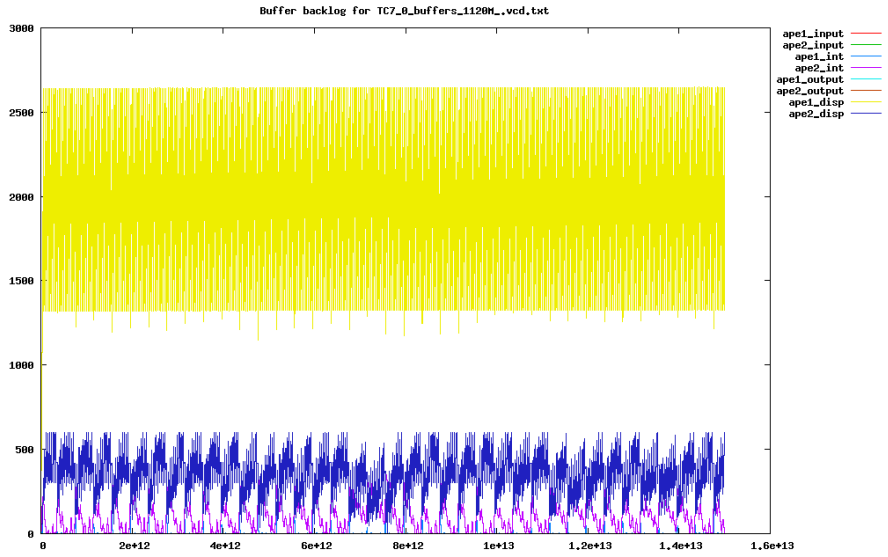

Figure H.93: *Buffer backlog for TC7.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

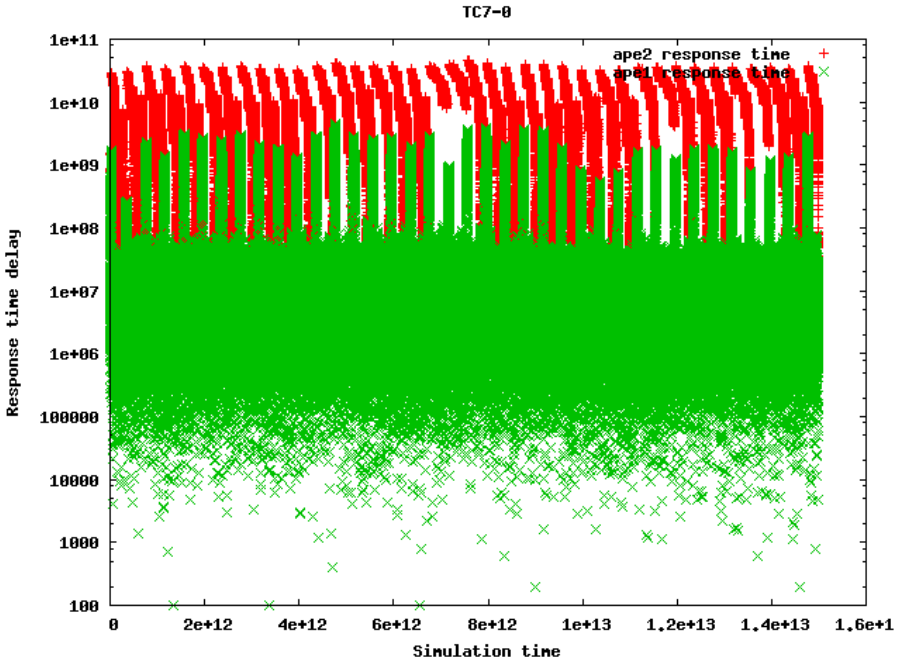

Figure H.94: *Idle response times for TC7.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

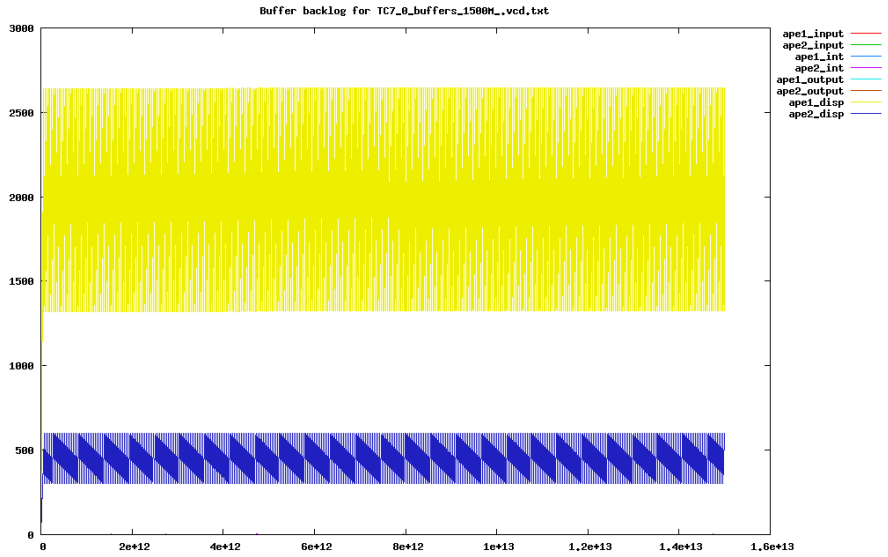

Figure H.95: *Buffer backlog for TC7.0 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

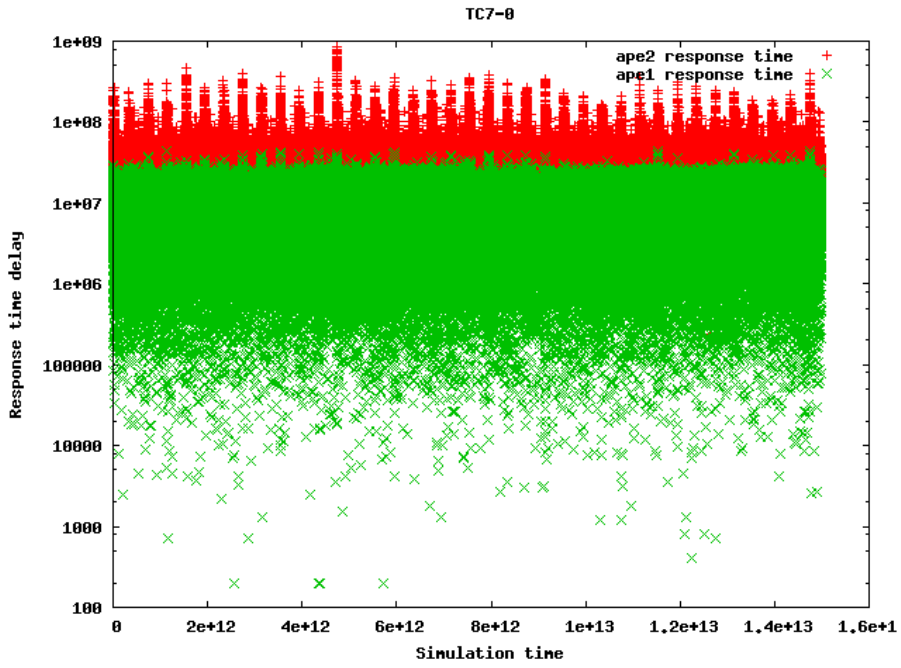

Figure H.96: *Idle response times for TC7.0 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

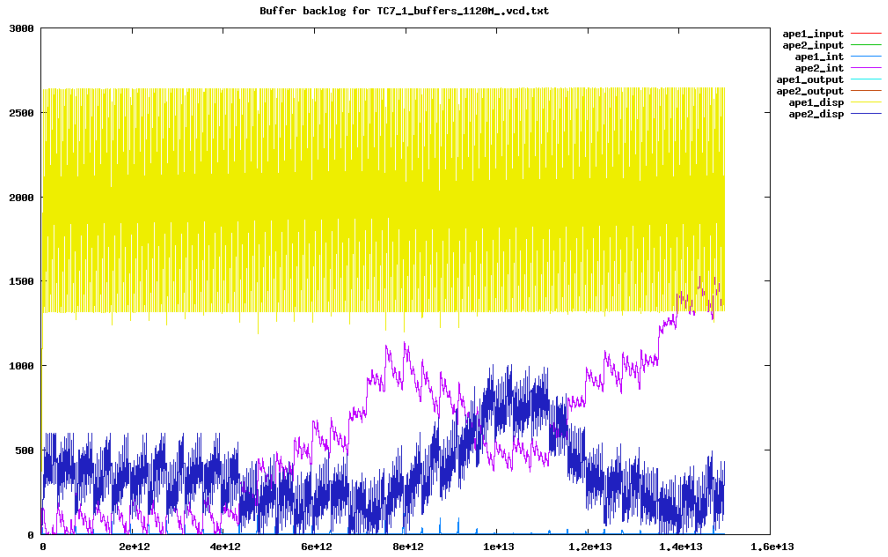

Figure H.97: *Buffer backlog for TC7.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

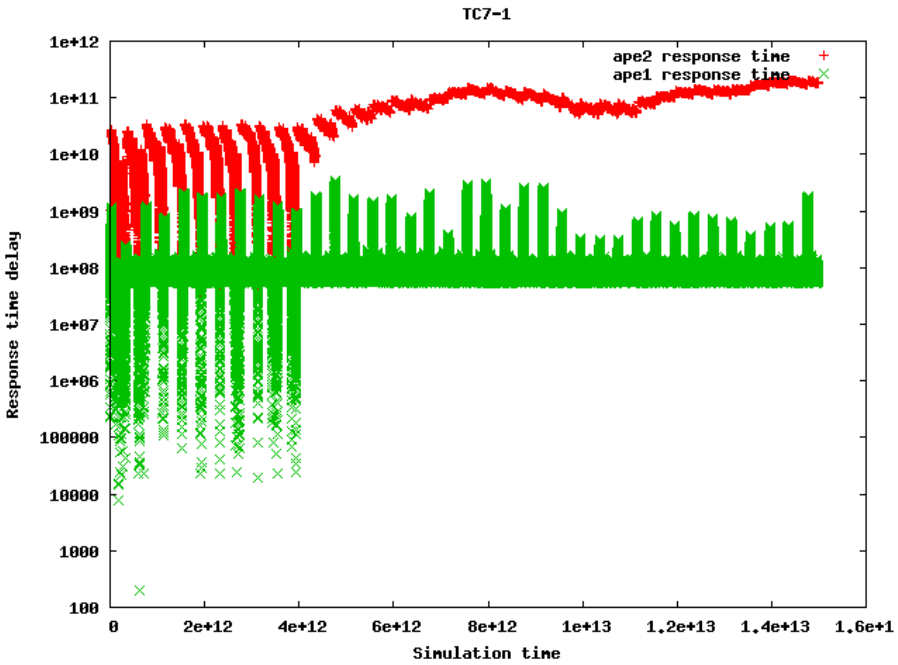

Figure H.98: *Idle response times for TC7.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

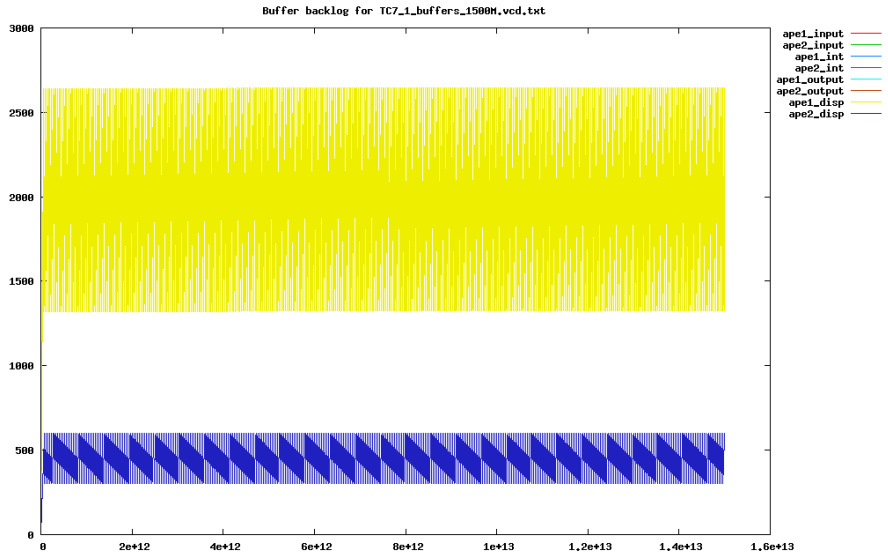

Figure H.99: *Buffer backlog for TC7.1 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

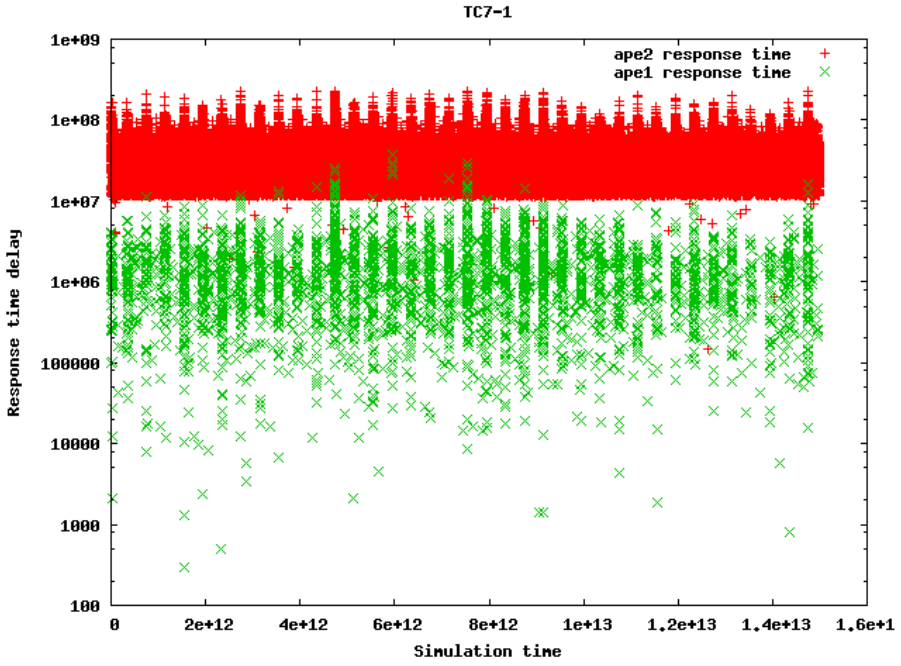

Figure H.100: *Idle response times for TC7.1 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

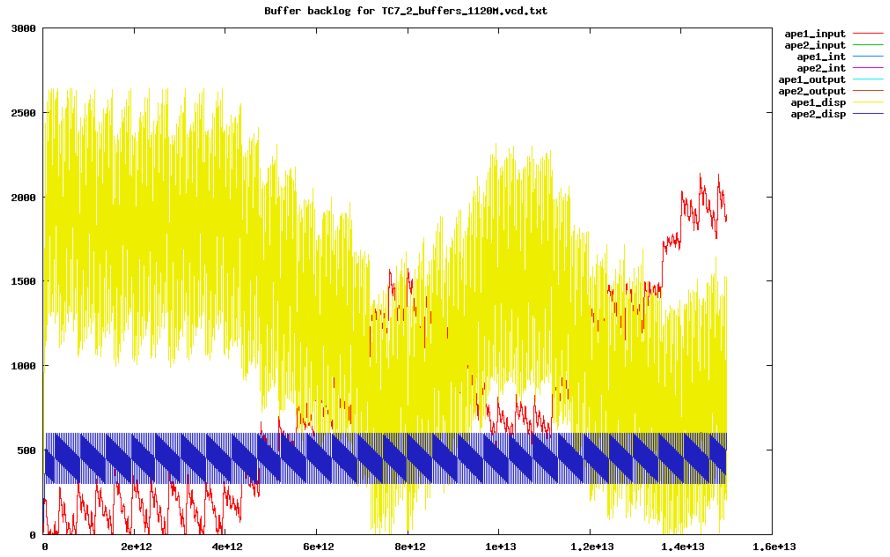

Figure H.101: *Buffer backlog for TC7.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

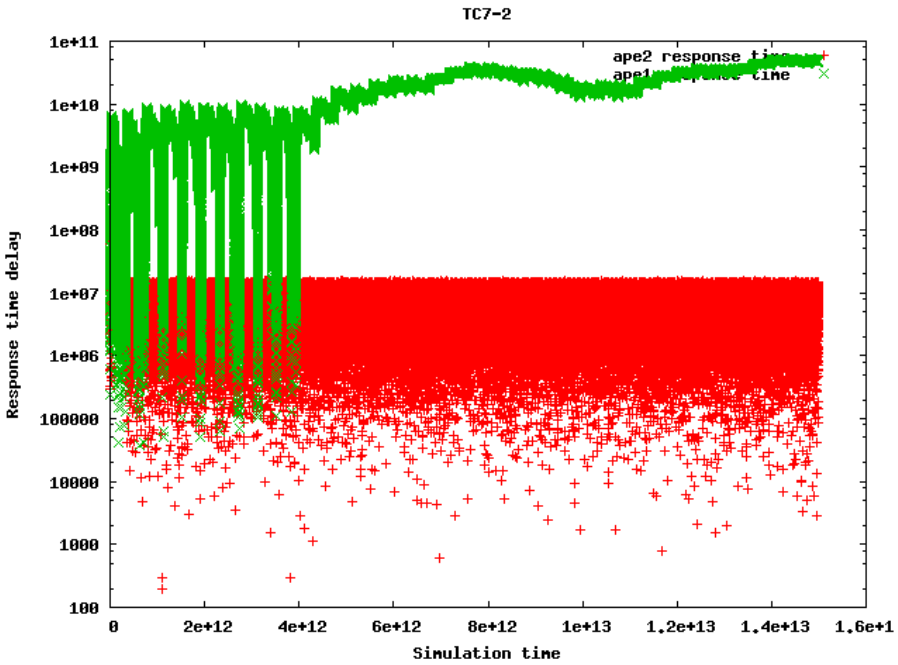

Figure H.102: *Idle response times for TC7.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

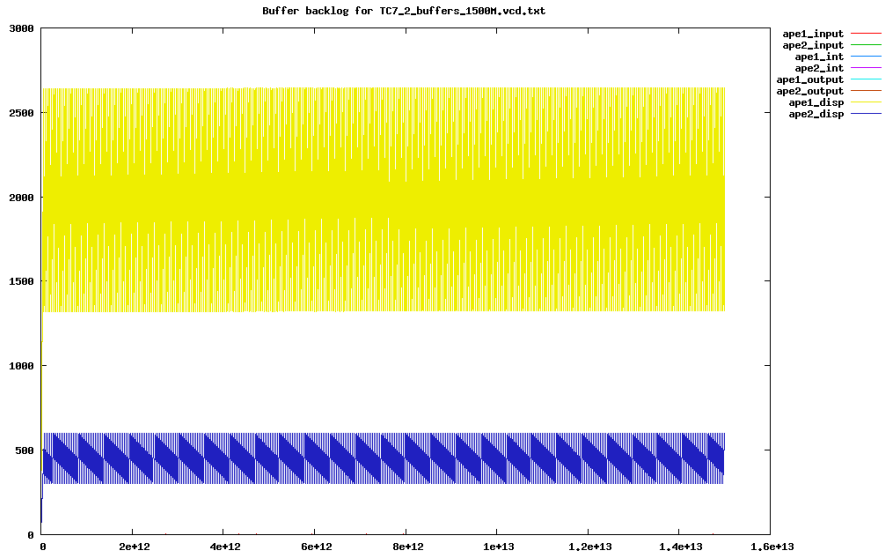

Figure H.103: *Buffer backlog for TC7.2 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

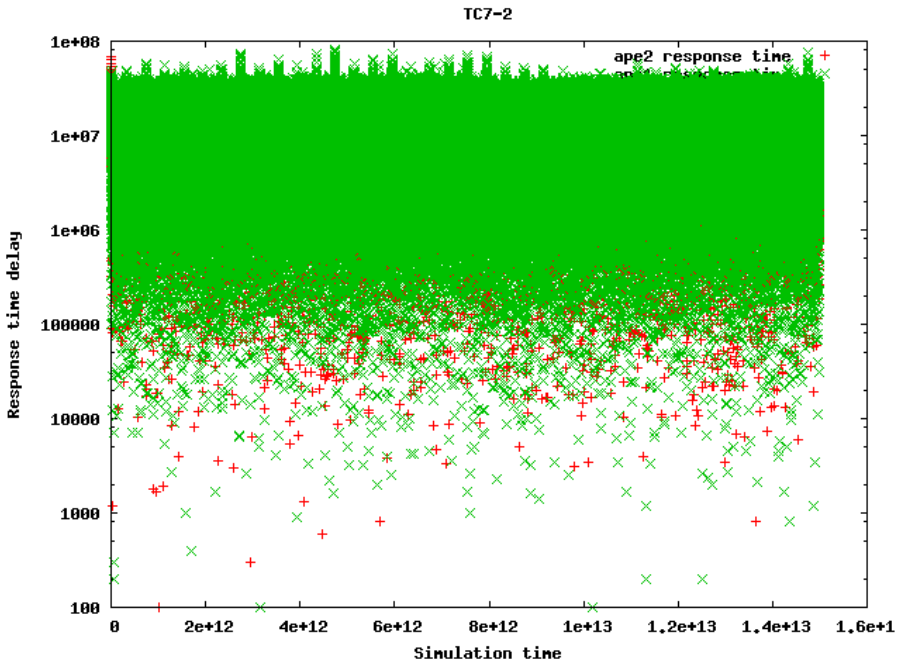

Figure H.104: *Idle response times for TC7.2 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

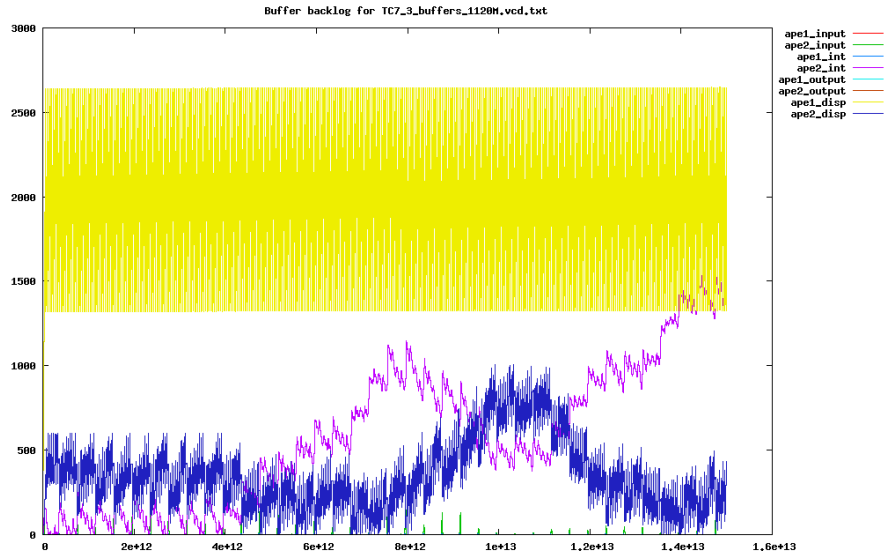

Figure H.105: *Buffer backlog for TC7.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

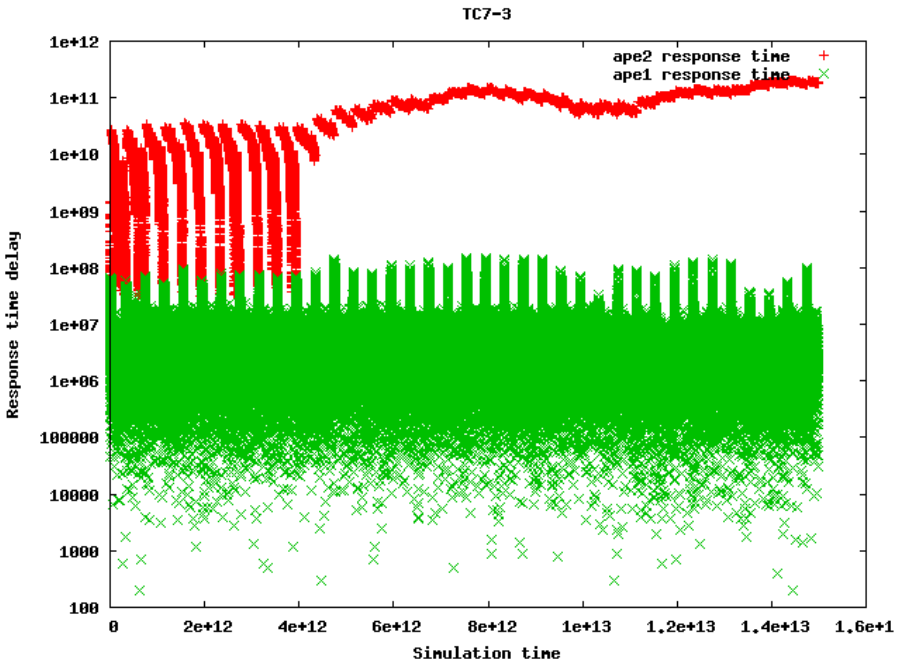

Figure H.106: *Idle response times for TC7.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

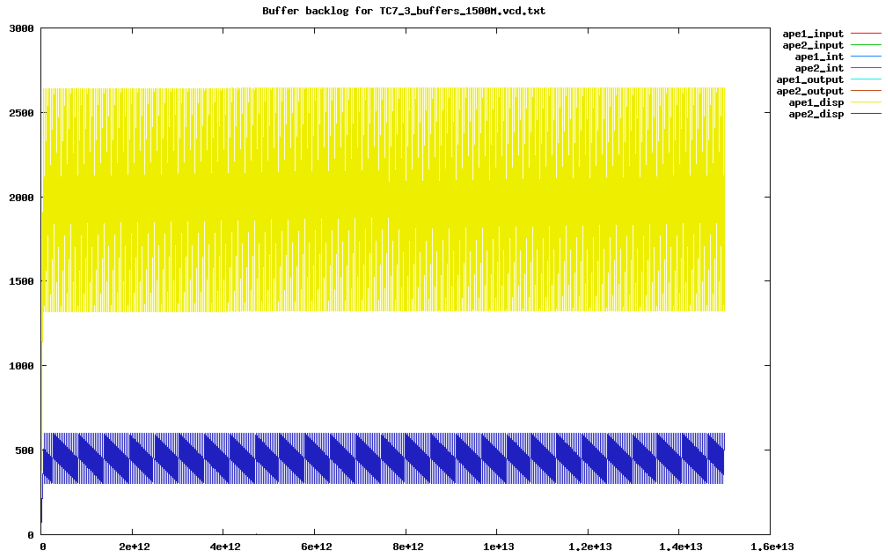

Figure H.107: *Buffer backlog for TC7.3 with the fast processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

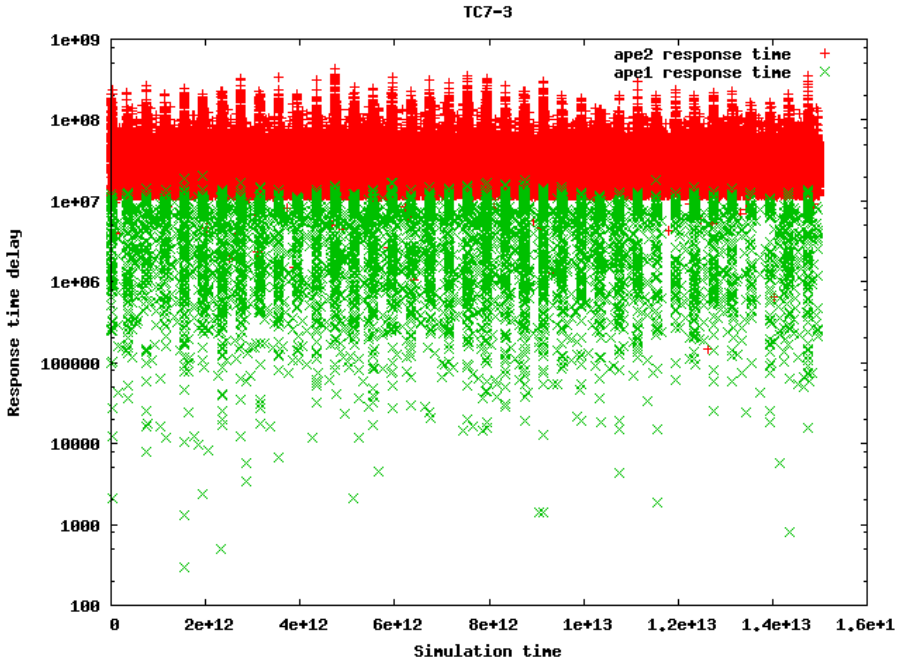

Figure H.108: *Idle response times for TC7.3 with the fast processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.9 Case study 8

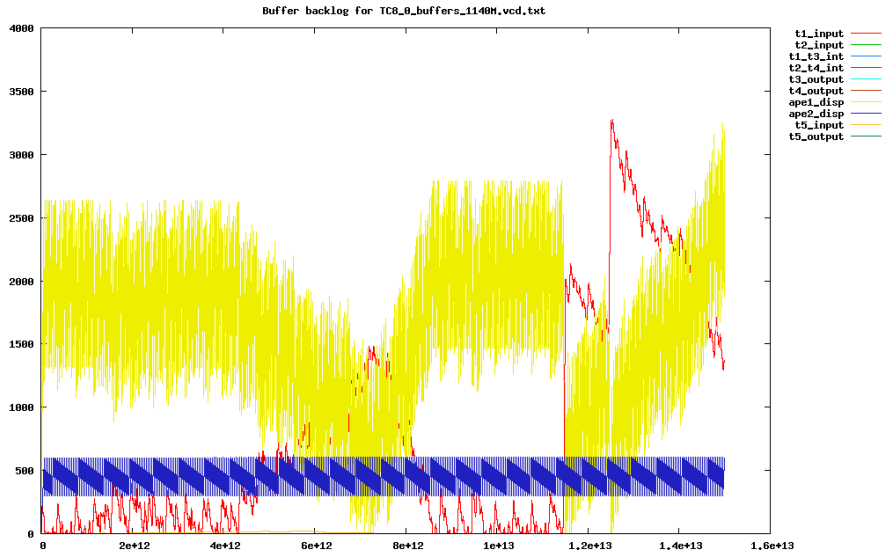

Figure H.109: *Buffer backlog for TC8.0 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

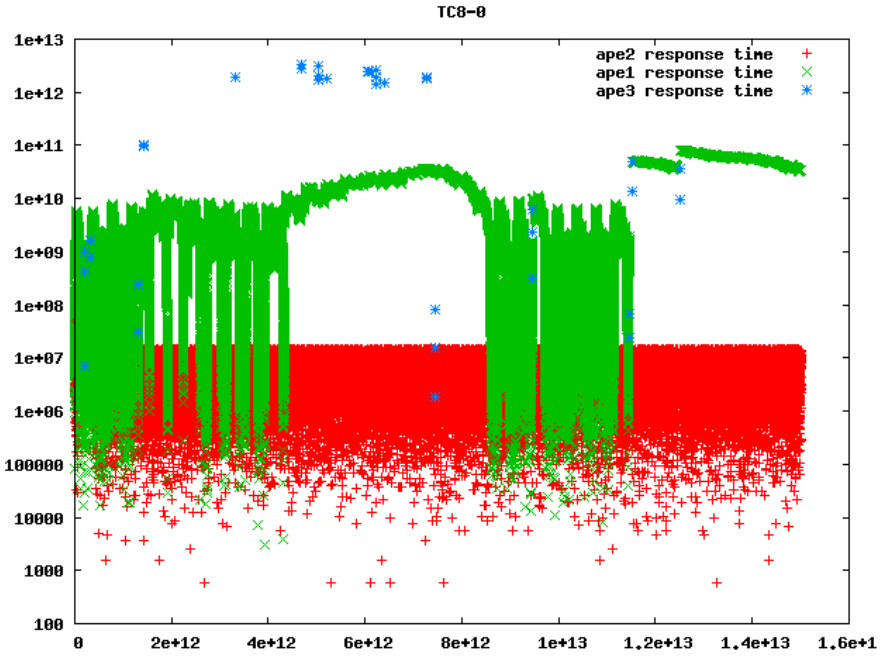

Figure H.110: *Idle response times for TC8.0 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

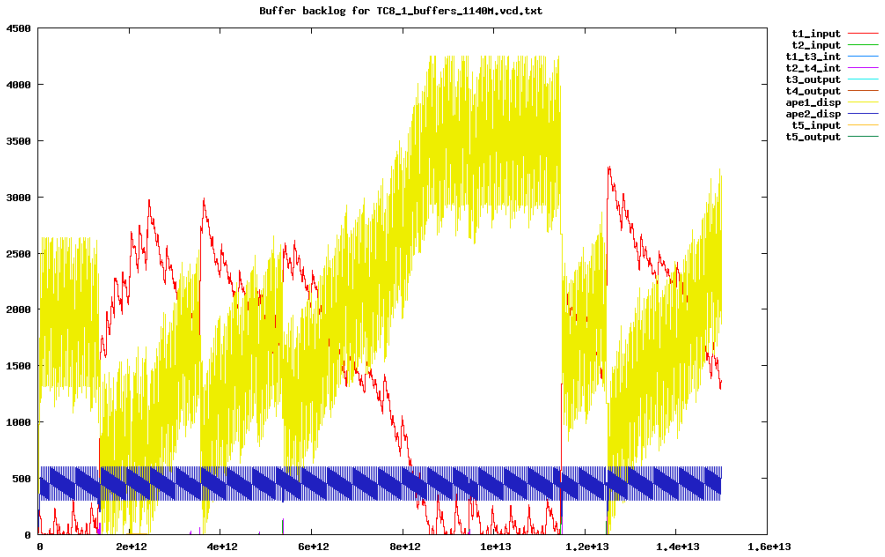

Figure H.111: *Buffer backlog for TC8.1 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

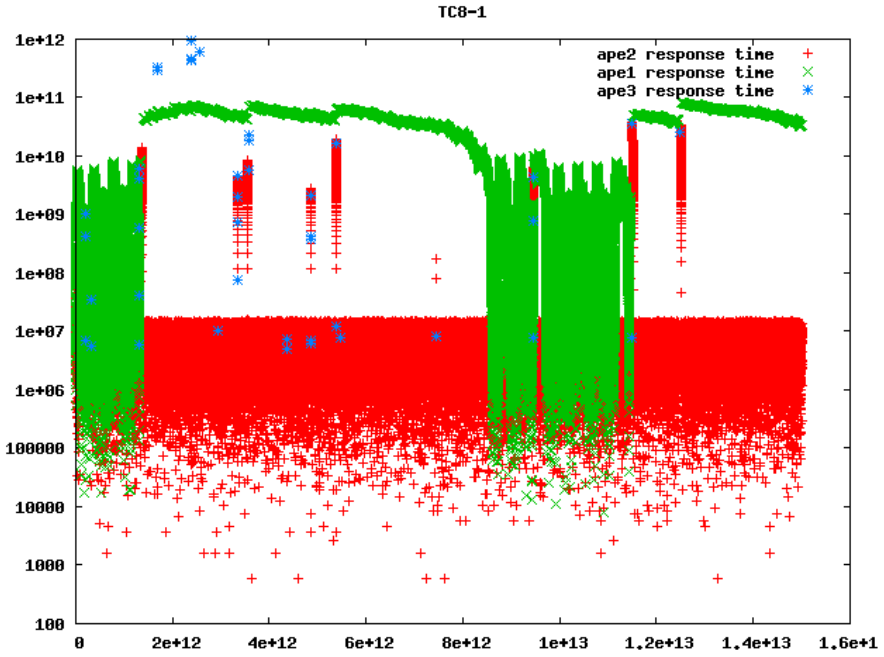

Figure H.112: *Idle response times for TC8.1 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

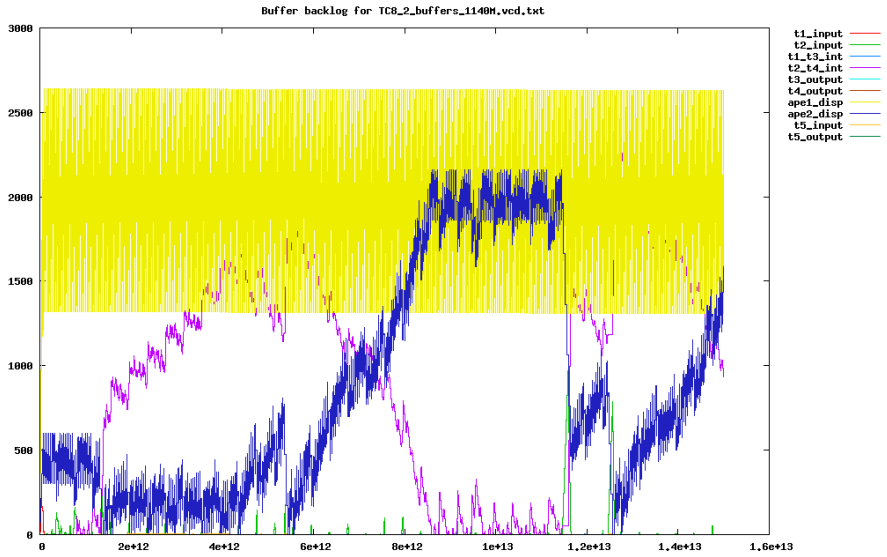

Figure H.113: *Buffer backlog for TC8.2 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

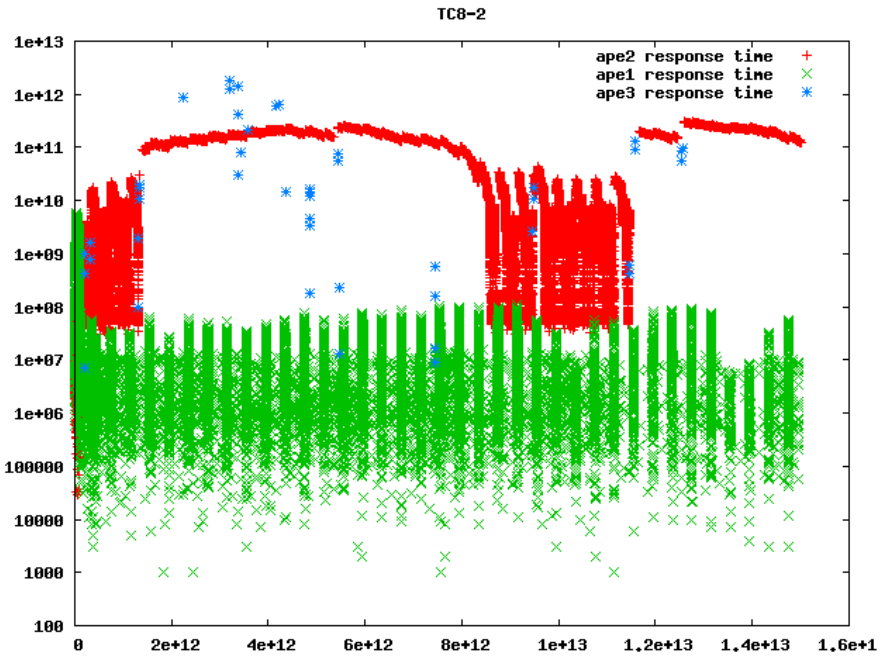

Figure H.114: *Idle response times for TC8.2 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

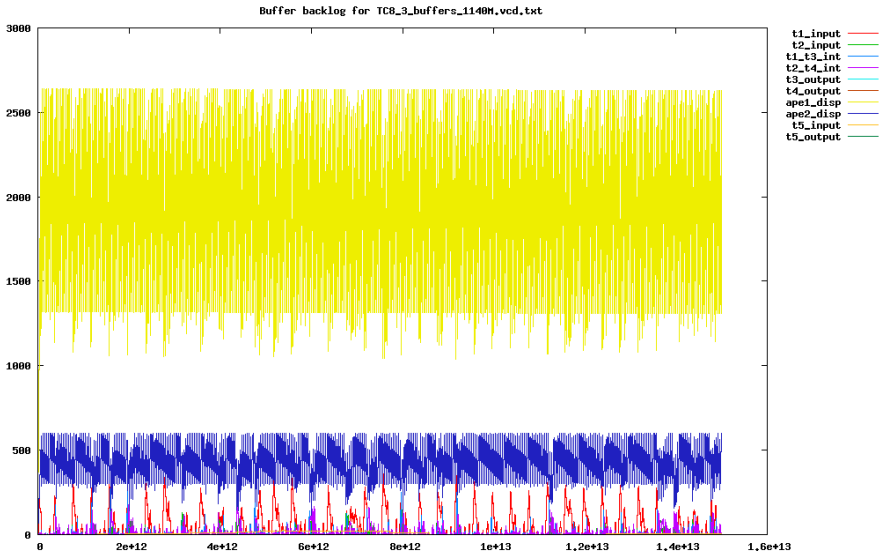

Figure H.115: *Buffer backlog for TC8.3 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

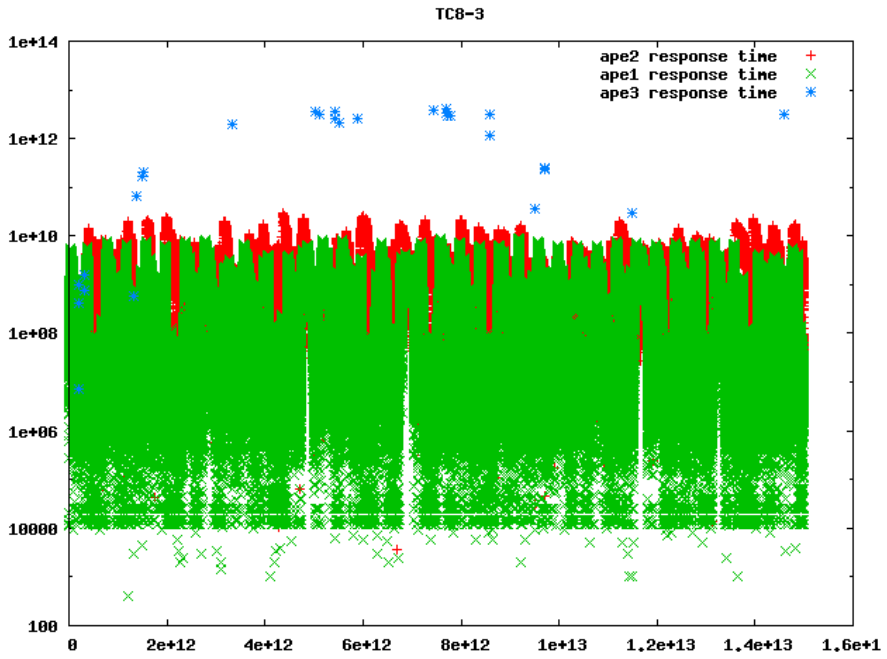

Figure H.116: *Idle response times for TC8.3 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

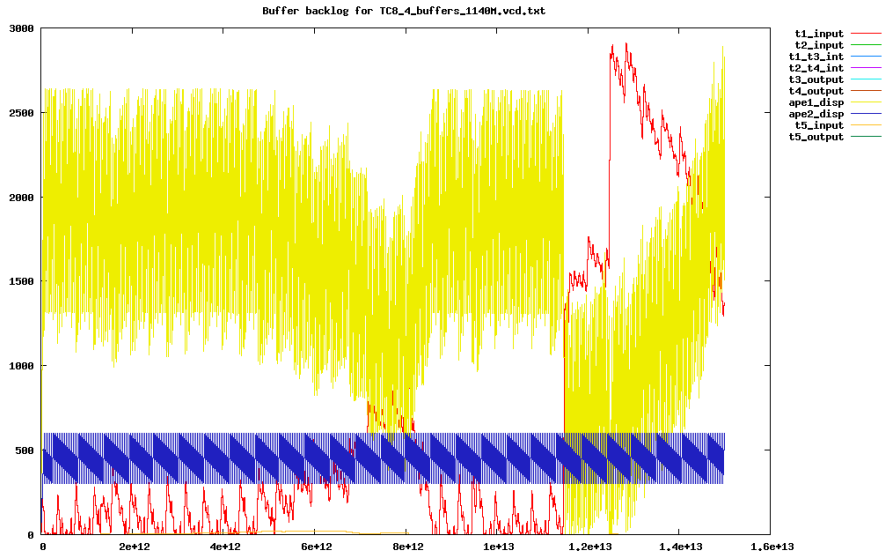

Figure H.117: *Buffer backlog for TC8.4 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

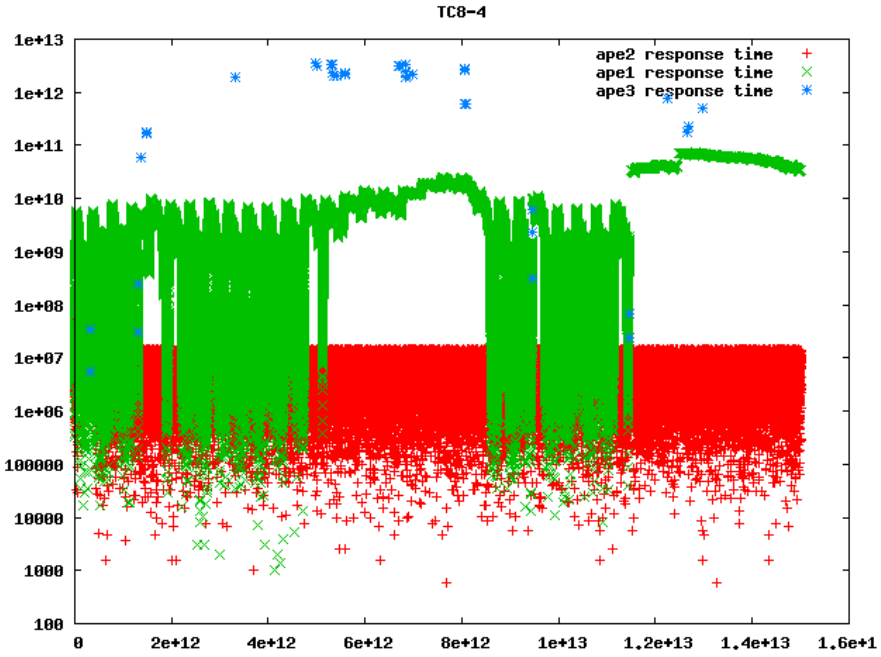

Figure H.118: *Idle response times for TC8.4 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

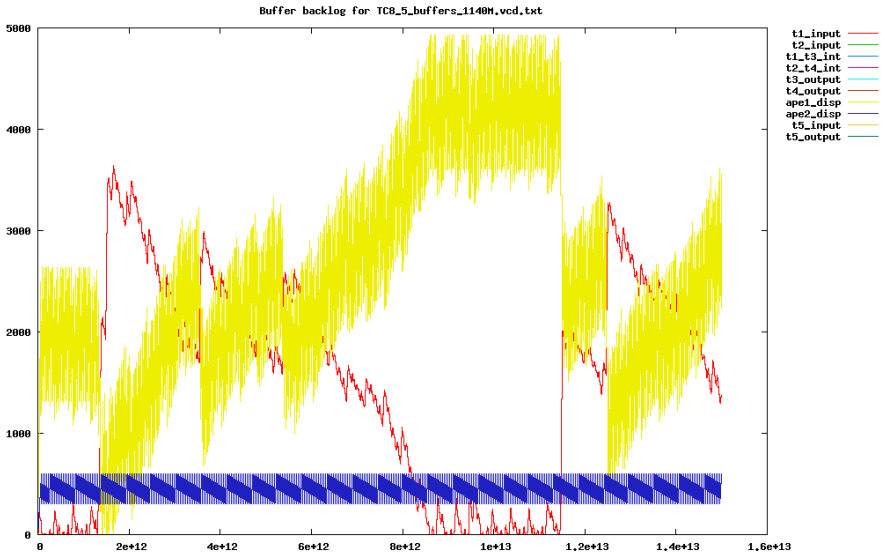

Figure H.119: *Buffer backlog for TC8.5 with the slow processing elements. Time is plotted in ps along the 1. axis, and the number of macro blocks in the buffer is plotted along the 2. axis.*

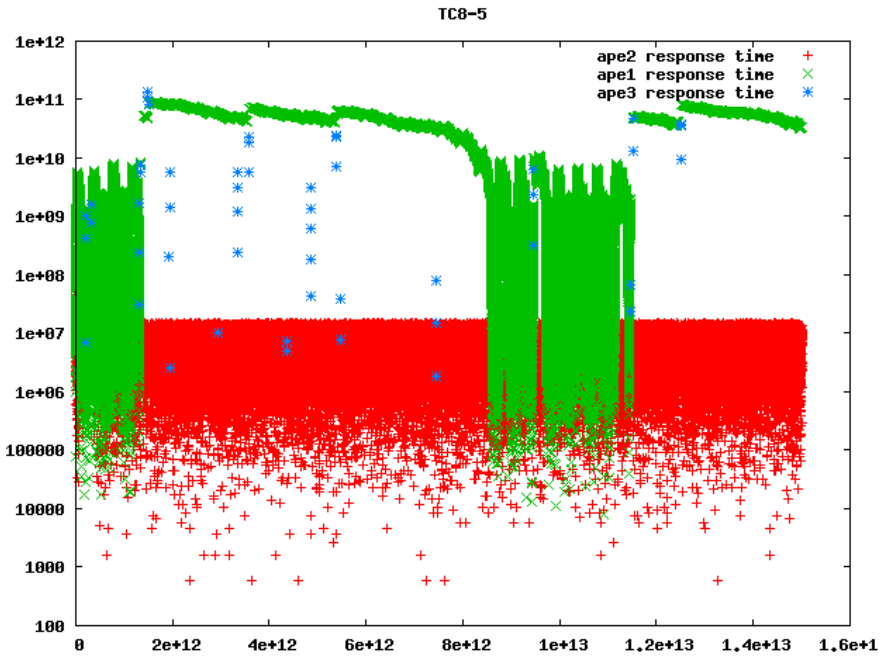

Figure H.120: *Idle response times for TC8.5 with the slow processing elements. The time in ps is plotted along the 1. axis, and the idle time in ps is plotted along the 2. axis.*

H.10 APE3 histogram

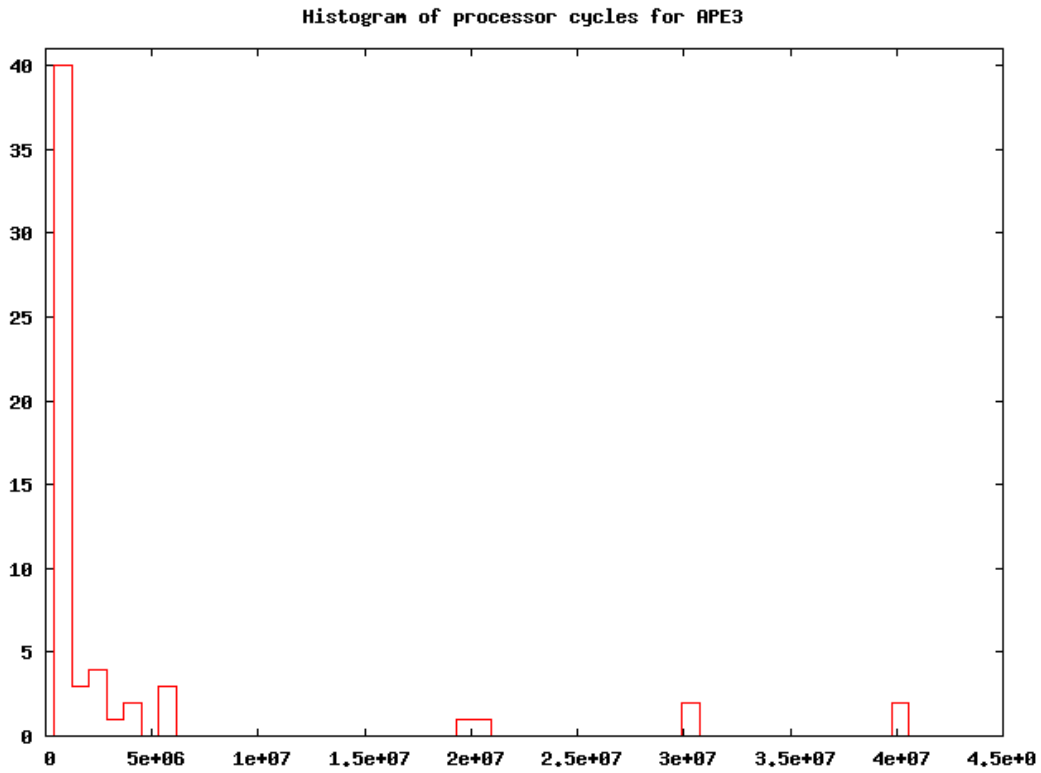

Figure H.121: *Histogram of the processor cycles required for the APE_3 stream (GUI). Most of the below task request require below 20.000 cycles, but some require up to 40.000.000 cycles.*