Technische Universiteit
Eindhoven
University of Technology

Master Thesis Embedded Systems

Quality versus energy trade-off for real-time applications on a composable MPSoC

Date: February 29, 2012
Author: Ing. S.T.F te Pas [0663210]
Supervisor: Prof.dr. K.G.W. Goossens
Tutor: Dr. K.B. Åkesson
Ir. A.T. Nelson

Abstract

This work consider multiple application with temporal requirements that running concurrently on an MPSoC platform. Additionally we consider energy constrained devices. The work is based on a running example, namely two H.263 video decoder as a picture in picture application, that has this temporal requirements.

This thesis looks into different aspects and solutions for making the H.263 video decoder adaptive. To do this, different functions inside the H.263 decoder are made scalable. The used platform is an instance of the CompSoC platform with the CompOSe RTOS. The concept of composability is used in this work. Video output quality is trade-off for energy by means of a quality manager.

Acknowledgements

I want to thank everyone of the electronic systems group for the great work in the past, that made this project possible.

In particular, show my gratitude to my supervisor Kees Goossens for providing this project, the great collaboration and the feedback of my thesis.
Next, I want to thank Benny Åkesson. Thanks to Benny, I got this assignment, the thoroughly document review and Benny always had time available for me whenever needed.
Last but not least Andrew Nelson, also for the throughly document review and for resolving problems related to the platform.

Finally, classmates, friends and the fellow graduate students on the 9^{th} floor for sociability, drinking coffee and support. And of course my parents for believing in me.

Contents

1 Introduction **1**
- 1.1 (Soft) Real-time applications . . . 1
- 1.2 Quality . . . 1
- 1.3 Energy and power constrained devices . . . 1
- 1.4 Multiple applications . . . 2
- 1.5 Problem . . . 2
  - 1.5.1 Adaptive H.263 decoder . . . 2
  - 1.5.2 Energy versus quality trade-off . . . 2
  - 1.5.3 Multiple independent adaptive applications . . . 3
- 1.6 Overview . . . 3

2 Related work **4**

3 Background **5**
- 3.1 Application . . . 5
  - 3.1.1 H.263 . . . 5
  - 3.1.2 Picture in Picture application . . . 7
- 3.2 The CompSoC platform . . . 7
  - 3.2.1 CompSoC . . . 8
  - 3.2.2 CompOSe RTOS . . . 9
- 3.3 Measuring quality . . . 11
- 3.4 DVFS and calculating energy . . . 12

4 Scalable application and platform **13**
- 4.1 Scalable H.263 . . . 13
  - 4.1.1 Scalable functions . . . 14
  - 4.1.2 Ignoring AC values . . . 15
  - 4.1.3 Skipping of macro blocks . . . 17
  - 4.1.4 Up-scalers . . . 19
  - 4.1.5 Multi application: Picture in Picture (PiP) . . . 21
- 4.2 Changes in CompSoC . . . 23
  - 4.2.1 Platform instances . . . 23
- 4.3 Mapping . . . 25
  - 4.3.1 Mapping to the platforms . . . 25
  - 4.3.2 Application and task schedulers . . . 28
  - 4.3.3 Buffers and memory mapping . . . 29
- 4.4 Debug infrastructure . . . 30

5 Quality manager and slack **31**
- 5.1 Quality manager . . . 31
- 5.2 Slack . . . 32
  - 5.2.1 Time slack . . . 32
  - 5.2.2 Data dependency . . . 37

5.2.3 Energy slack and quality levels . . . 40
5.3 Policies . . . 42
5.4 Control loop . . . 43
5.4.1 API . . . 44

6 Experiments and results **46**
6.1 Experimental setup . . . 46
6.2 Evaluate application scalable functions . . . 48
6.2.1 Ignoring AC values . . . 48
6.2.2 Skipping macro blocks . . . 51
6.2.3 Up-scaler . . . 54
6.2.4 Configurations . . . 58
6.3 Evaluate mappings . . . 60
6.3.1 Reevaluated quality modes . . . 65
6.4 Evaluate quality manager . . . 66
6.5 Evaluate composability . . . 67

7 Conclusions **70**

8 Future work **71**
BibliographyI

List of Figures

1 Structure macro block . . . 5
2 The six process steps of the H.263 decoder . . . 6
3 Picture in Picture application . . . 7
4 A system based on the CompSoC platform . . . 7
5 CompSoC overview . . . 8
6 CompOSe . . . 10
7 CompOSe Application Execution Scheme . . . 10
8 Task graph H.263 decoder with up-scaler . . . 14
9 Influence of the adaptive function on the different tasks . . . 14
10 Resulting frame when AC values are ignored . . . 16
11 Resulting error in the frames when AC values are ignored . . . 17
12 Resulting frame when macro blocks are skipped . . . 18
13 Error in the image resulting from skipping of MB's . . . 19
14 Bi-linear interpolation, grid density different up-scalers . . . 20
15 Visual quality, different up-scalers . . . 21
16 PiP with the mux and mode . . . 22
17 Possible artifacts when changing PiP mode . . . 23
18 Hardware platform overview A . . . 24
19 Hardware platform overview B . . . 25
20 Mapping of the application on platform A . . . 26
21 Used network connections in platform A . . . 26
22 Mapping of the application on platform B . . . 27
23 Used network connections in platform B . . . 27
24 Task schedulers of the H.263 applications . . . 28
25 Debug infrastructure on the screen at runtime . . . 30
26 CompOSe application execution scheme with the quality manager . . . 31
27 Required number of cycles to produce a video frame for different movies . . . 33
28 Needed amount of cycles to produce a macro block for different movies . . . 34
29 Average required number of cycles to produce a macro block . . . 35
30 Data dependency between the cores . . . 37
31 Data dependency between the cores, core 2 runs one macro block behind . . . 38
32 Data dependency between the cores, multiple applications . . . 38
33 Deviation (Time) between frame produced and frame deadline, positive deviation means the frame is produced before deadline . . . 40
34 Quality level regions example . . . 41
35 Control loop of the implemented policy in the quality manager . . . 43
36 Resulting quality per frame depending on the number of AC values to process . . . 49
37 Resulting energy usage, number of AC values to process . . . 50
38 Influence on quality when skipping macro block, with different threshold . . . 52
39 Energy usage when ignoring AC values and potentially skipping macro blocks . . . 53
40 Resulting overall quality different up-scalers and number of AC values to process . . . 55
41 Results different up-scalers . . . 56
42 PSNR per Joule, for different up-scalers . . . 57

43 Energy results for different movies with different up-scalers . . . 57
44 PSNR vs. energy, grouped per movie . . . 58
45 PSNR vs. energy, grouped per up-scaler . . . 59
46 Usage energy to produce 24 frames, Tree128 movie at different frame rates . . . 60
47 Used energy for 5 movie frames at different buffering configurations, application quality mode: Best . . . 62
48 Used energy for 5 movie frames at different buffering configurations, application quality mode: Good . . . 63
49 Overall quality depending on the energy budget . . . 66
50 Energy usage per frame . . . 67
51 Energy difference between two runs . . . 68
52 Energy difference between different situations . . . 68
53 Energy usage per frame, no DDR . . . 69
54 Energy difference between different situations . . . 69

List of Tables

1 PiP modes . . . 22
2 Description for the different notations used in this chapter . . . 35
3 Movie test set . . . 47
4 Movie coding settings . . . 47
5 Quality modes . . . 60
6 Energy penalty for quality mode: Best . . . 62
7 Energy penalty for quality mode: Good . . . 64
8 Energy penalty for quality mode: Good, no skipping of macro blocks . . . 64
9 Revisited quality modes . . . 65
10 Summary energy penalty to buffering, at different quality modes . . . 65

1 Introduction

Multiprocessor Systems on a Chip (MPSoC) with multiple applications with real-time requirements are becoming more common. Applications that run on these systems may be dynamic, have variable execution times (ET) and may be distributed over multiple cores. Video decoders are one such class of soft real-time applications that run on these systems.

1.1 (Soft) Real-time applications

In this work, we focus on soft real-time applications. Real-time applications are applications that have temporal requirements. The severity of the requirement is commonly graded from hard to soft, with soft being the most relaxed.

As our running example in this thesis, we use a H.263 video decoder. With the requirement being dependent on the decoded video's frame rate. Each frame has a deadline in time when it needs to be shown on the screen.

1.2 Quality

Quality is often a subjective notion that may be applied to many aspects of the system. In this thesis we focus on the quality of the application output. With quality we mean the loss of information of the produced output of the application.

For the running example, the resulting output are the video frames. The user perception of video quality differs per person. Quality of a video frame can be measured objectively by the Peak Signal to Noise Ratio (PSNR). The PSNR indicates the mean deviation between the produced output frame and a reference frame (highest possible quality of the same frame). PSNR cannot be related one to one to the quality perceived by a person, but gives an objective value and is widely applied to measure image quality.

1.3 Energy and power constrained devices

An application needs execution time, resulted in energy usage. Energy and power constrained devices are ubiquitous in everyday use, e.g. mobile phones. We focus on energy constrained devices that run applications with soft real-time requirements.

To deal with power and energy constraints, different techniques exist. Dynamic Voltage Frequency Scaling (DVFS) and clock gating are common techniques that are used. It has also been shown how the techniques may be used in the context of real-time requirements. DVFS used the unused clock cycles to reduce the frequency and corresponding voltage. Another development is that systems become multi-core. The advantage of having multiple cores is that it increases the

computation power and lowers the energy usage. Multi-core has its limits. The computation power can only be used by the system if the application can be distributed efficiently among the different cores. Distributed applications are applications where the tasks are mapped to multiple cores.

1.4 Multiple applications

Where in the past for example a mobile phone is only used for calling, today multiple applications are running simultaneously on your smart phone. Current smart phones have an operating system that enable multiple applications to run concurrently on the same device. In this thesis we use the CompOSe OS to deal with multiple applications.

1.5 Problem

While much work has been done in the field of multiple concurrently running applications, applications with temporal requirements and energy reduction, there are still many open problems. In this thesis we contribute solutions for the following three problems.

1.5.1 Adaptive H.263 decoder

The example application needs execution time and has energy/power constraints. DVFS is used to lower the frequency and hence energy. To scale the energy down more, additional execution time need be saved.

We can decrease its execution time by lowering the quality. But for the example application, no functions to change the execution time and quality are available. The example application needs to be made adaptive. With an adaptive H.263 decoder, we get control over the loss of information in the produced output and the required execution time.

In this thesis, we introduce different scalable functions (in terms of required execution time and resulting output quality), to make the H.263 decoder adaptive.

1.5.2 Energy versus quality trade-off

We have an application with different scalable functions that produce different output quality, with different execution time.

Determining the needed energy for a quality is not known upfront. But we want to trade-off energy for quality and find optimal configurations in terms of energy and/or quality.

This thesis provides a solution that enables the given MPSoC to trade-off energy and output quality of the application by means of a quality manager. The different defined mechanisms in

the H.263 application are combined with an existing DVFS infrastructure to make this trade-off possible. We propose a quality management policy that satisfies temporal requirements and makes the energy budget should last just long enough to maximize quality given an energy budget. It observes the applications slack in execution cycles and the slack in energy budget at runtime. Slack is defined as the difference between the used and the available budget. We experimentally investigate the energy versus quality trade-off for the H.263 decoder.

1.5.3 Multiple independent adaptive applications

As stated in Section 1.4, applications may run concurrently on the platform and may have mixed time criticality.

Having multiple (adaptive) applications running simultaneously complicates the development due to inter-application interference. With inter-application interference we mean that the behavior of an application, including its timing, and energy profile and usage is affected by the absence or presence of other applications.

To develop and run the applications with their own quality manager independently, composability is taken into account. Composability helps to simplify the design and verification of a system that contains multiple applications. The CompOSe OS enables a virtual platform per application with no interference between applications (composability). We demonstrate the simultaneous execution of multiple H.263 decoder instances with quality managers on an FPGA instance of the platform.

1.6 Overview

In Chapter 2, the work that is related to this thesis is discussed. Background information is presented in Chapter 3, including the H.263 decoder, the CompSoC hardware platform and CompOSe OS, the existing DVFS technique and a discussion on how to measure video quality. Chapter 4 describes the scalable functions in the application, the multi-application, the used platform and the mapping. The slack, quality manager and policies are described in Chapter 5. The scalable functions in the application and mapping for the H.263 decoder are evaluated in Chapter 6. The policy and composability of applications using our technique is evaluated in Chapter 6 as well. Chapter 7 contains the conclusions. New research questions resulting from the results are given in Chapter 8.

2 Related work

Adaptive applications

Hentschel introduces resource-quality scalable video algorithms [11]. This is focused on consumer terminals, and the budget limitation is related to computation cycles. For example, to be able to run the application on a low-end and a high-end product (cost-efficient implementations).

For us, the limitations are related to the energy budget, the application needs to adapt on the available energy. We focus on adaptivity at an application level and not on a global (system) level because each application runs in its own virtual platform.

Trade-off

Different trade-offs for application configuration and the available resources, has been performed in the past. Quality trade-offs for video (terminals) is performed by Bril et al. [3, 24]. A resource manager and a quality manager are combined. It is focused on consumer terminals and processor usage. The resource manager controls the settings of the combined applications. We have controller per applications and per core, not for the overall system, and the manager takes the energy/power consumption into account.

A manager that takes energy into account is for example investigated by R. Nathuji et al. They show feedback driven Quality of Service to lower power consumption in virtualized servers [18]. We do the trade-off for one application in isolation whereas they do it per platform.

Multiple independent applications

Three parts need to be taken into account when integrating multiple applications with real-time requirements on a single platform. Namely the hardware platform, the applications running on it, and the operating system. The hardware architecture is important, in terms of processor architecture and the network on chip (NoC) used between the processors. The used hardware in this project is developed for composable execution of time critical applications. Research has been performed on the used platform [10, 15, 8], and the Æthereal NoC [6, 7].

Scheduling of applications is performed by the OS. For time critical systems a Real Time OS is needed. A Real Time Operating System is developed for the used platform named CompOSe [9]. This RTOS is developed with the concept of composability in mind. Composability is a recent development to enable the verification of the timing requirements, of concurrently executing applications. Power management is available with energy and power budgets in the implementation of the CompOSe RTOS [16].

We extend the CompSoC system to enable multiple quality managers on the platform. This has not been done before.

3 Background

In this chapter, additional information is given that may be helpful to understand this thesis. It covers the H.263 application, the hardware platform and the CompOSe real-time operating system (RTOS),

3.1 Application

H.263 was established as the running example in Chapter 1. H.263 is a coding standard for movies. This section describes the basic structures and process steps in a H.263 decoder. Scalable functions in the H.263 decoder are introduced in Section 4.1.2 and 4.1.3 are based on the data types of the H.263 standard.

3.1.1 H.263

This section describes the H.263 decoder application. We use the basic version of the H.263 standard [4]. This means that it has two type of frames:

- **Intra:** also known as I-frames. No motion compensation is applied on these frames. The frame is decoded independently, without information from other frames.
- **Inter:** also known as P-frames. These frames use the previously produced frame and applies the motion vectors to get the data from the previous frame. The difference between the produced frame and the actual frame is encoded in the P macro blocks.

Data structures

Our implementation of the application works with macro blocks that are sent between the different tasks. Each macro block contains six blocks. Four of these blocks contain information about luminance (Y) and two blocks contain chrominance information (Cr and Cb). Each block is 8x8 pixels. The structure of a macro block is given in Figure 1. This structure is used over the whole application flow.

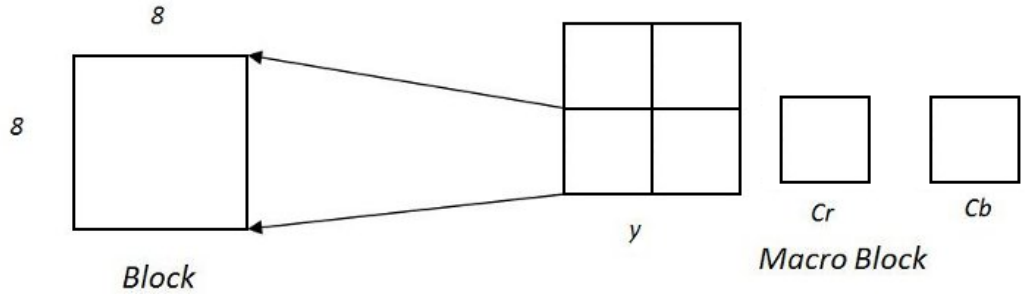


Figure 1: Structure macro block

Decoding process

The H.263 decoder has 6 process steps (VLD, inverse Quantization, Inverse DCT, Motion Compensation and Frame Reconstruction) that are executed to reproduce a frame. Additionally, a scaling step is applied afterwards. The steps are shown in Figure 2.

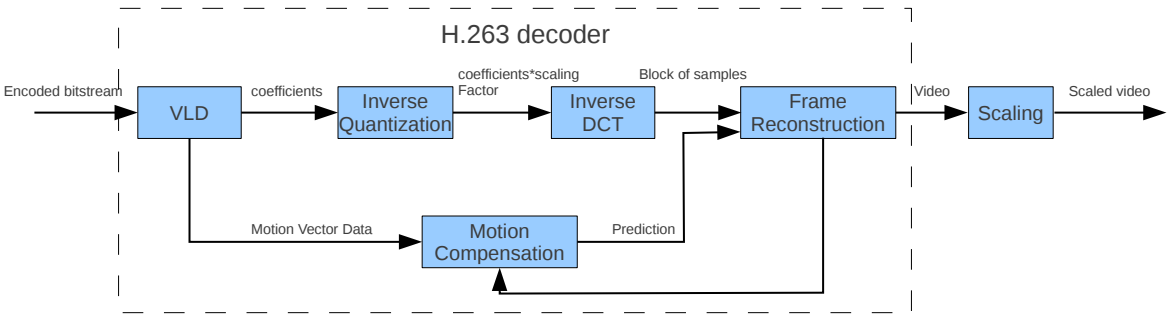


Figure 2: The six process steps of the H.263 decoder

Each process is responsible for the following action:

- **VLD:** The variable-length code that make up the H.263 bit stream is decoded to extract the coefficient values and motion vector information.
- **Inverse Quantization:** All coefficients are multiplied by the same scaling factor that was used in the quantizer of the encoder. Some information is lost in the encoding quantization process.
- **Inverse DCT:** It is the inverse of the Discrete Cosine Transform (DCT) operation in the encoder to create a block of pixel samples. IDCT performs a transformation from the frequency domain to the pixel domain.
- **Motion compensation:** The motion vector information is used to pick pixel data from the previous frame. The values of the block is added to get the output frame.
- **Frame Reconstruction:** The reconstructed frame is stored or is sent to the screen to be displayed.
- **Scaling:** Scales the resolution of the reconstructed frame up or down.

3.1.2 Picture in Picture application

In order to demonstrate the applicability of our technique to a composable multi-core, multi-application system, we use a Picture in Picture (PiP) application. We achieve this by decoding two different H.263 encoded streams at the same time and display them concurrently. Figure 3 shows the PiP system. Two instances of the H.263 decoder run in parallel. The resulting output is merged together to get the final output. Each H.263 decoder reads a separate movie input stream, processes independently and sends the result to the multiplexer (mux). The mux multiplexes the two results into a PiP image. The two H.263 decoders and the mux are (for composability reasons) decoupled, and explained in Section 4.1.5.

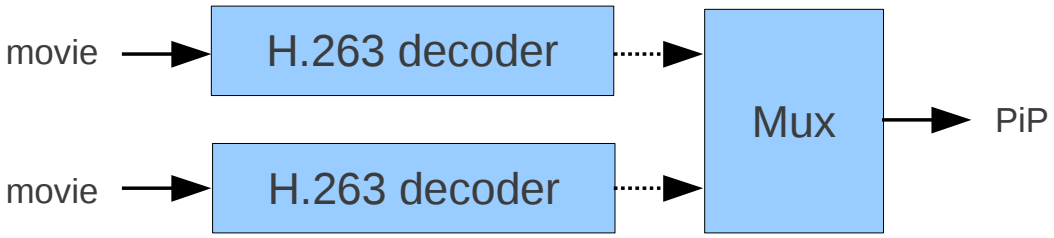


Figure 3: Picture in Picture application

3.2 The CompSoC platform

This section gives the platform overview. The used platform consists of the CompSoC hardware platform and the CompOSe Real Time Operating System (RTOS). The layering of the different components of the platform is shown in Figure 4. The CompSoC hardware platform and its components is described in Section 3.2.1, the CompOSe RTOS that runs on CompSoC is described in Section 3.2.2. Each application runs in its own virtual machine (VM) on top on CompOSe.

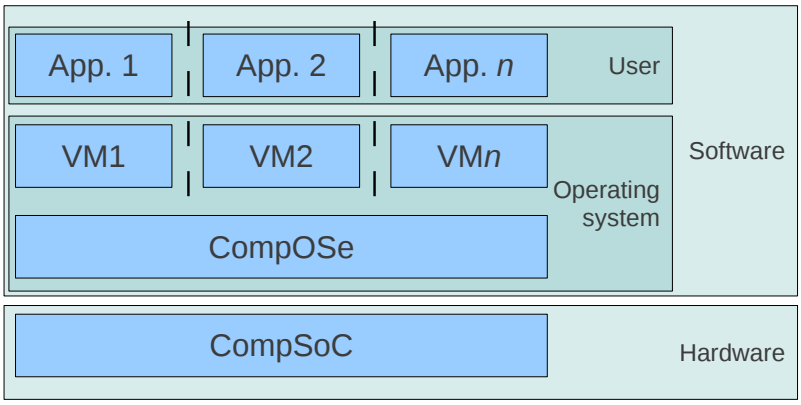


Figure 4: A system based on the CompSoC platform

3.2.1 CompSoC

This section introduces the CompSoC platform and its components. The CompSoC [10] platform is instantiated on a Xilinx ML605 FPGA [26]. Figure 5 illustrates the CompSoC platform at the IP-level. At the tile level, an instance of the CompSoC platform contains MicroBlaze tiles, a Host MicroBlaze tile, and a memory tile, all connected by an instance of the Æthereal Network on Chip (NoC) [7]. A MicroBlaze tile contains a MicroBlaze processor core and local memories. Each application needs its own communication memories for composability reasons.

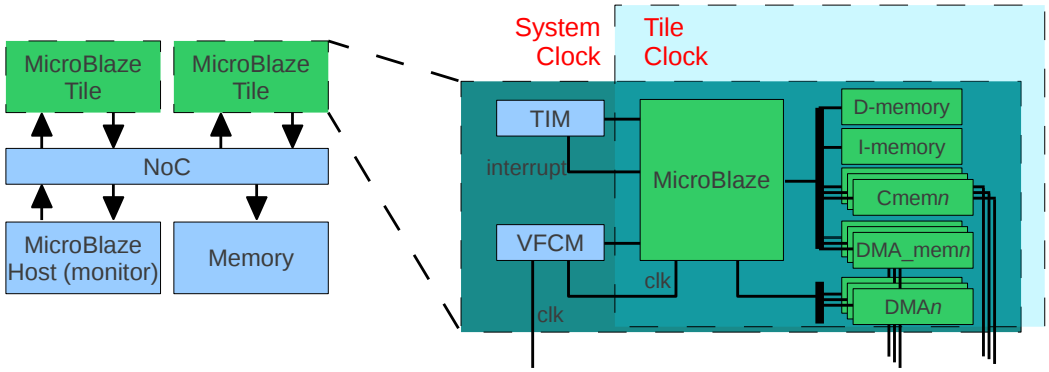


Figure 5: CompSoC overview

- **MicroBlaze Tile:** Consisting of a MicroBlaze core [25], DMA controllers and local memories. Each tile exists in its own clock domain, as illustrated by the tile clock bounding in Figure 5. The tile frequency is controlled via the VFCM that is connected directly to the tile's MicroBlaze core. This enables independent DVFS per tile.
  - **MicroBlaze core:** is a softcore from Xilinx and is based on a RISC architecture with separate data and address bus. It has registers (32), ALU, shift registers, two types of interrupts, and is pipelined (five stage pipeline).
  - **Direct Memory Access (DMA) controller:** Each tile has multiple DMA controllers. A DMA controller is a hardware module that can access memory independently from the CPU. The processor outsources data transfers to the DMA to enable parallel communication and computation. Each DMA controller is connected to a local scratchpad memory (DMA_mem). The controller can read and write to his own DMA_mem or external Cmem.
  - **Memories:** The MicroBlaze tile has local memories for different uses.
    - **D-memory:** in this memory the application data is stored.
    - **I-memory:** in this memory the application instructions are stored.
    - **Cmem:** each tile has multiple "Cmem" memories. The purpose of these memories is to facilitate inter-tile communication. These are the only memories that can be accessed accross the NoC.

    - ∗ **DMA_mem:** This memory is used by the DMA controller. The purpose of these memories is to facilitate inter-tile communication. Each DMA has access to one local DMA_mem. Data is placed here for communication out of the tile. Data can also be read into this memory from across the NoC.
  - **VFCM:** Voltage Frequency Control Module (VFCM) is used for DVFS of the tile.
  - **TIM:** Timed Interrupt Module.
- **Monitor core:** This is a special MicroBlaze core that is used to send debug messages over a UART serial connection, loading data into memory and synchronizing starting and stopping the different cores.
- **DDR:** The system has 64-bit DDR3-1066 with capacity of 512 MB, runs at 200 MHz memory. This is shared between all the components in the system and is only accessible over the NoC. Reading and writing data to and from the DDR takes more cycles than local memories. The DDR controller is not composable, hence applications may affect each other if they share the DDR. For a composable DDR controller see [1].
- **Network on Chip (NoC):** To connect the different components to each other, the Æthereal Network on Chip [7] [6] is used and is shared using Time-division multiplexing (TDM). TDM is an mechanism that gives time slots to specific network connections. Connections need to be specified between the different components that are connected to the network. This is specified when the platform is configured at design time.

3.2.2 CompOSe RTOS

CompOSe [9] is a composable Real-time Operating System (RTOS) and is introduced in this section. In essence, it provides services related to scheduling and power management.

The RTOS has two levels of scheduling. The first level is the application scheduler. The applications are scheduled by means of a TDM table, and is part of the CompOSe OS. An application has at least one task, with firing rules (conditions when a task is allowed to fire), its own task scheduler and power manager. The power manager allows you to set an energy/power budget for the application and to set a function that is called before the task execution. Tasks inside the application are scheduled by user specified task schedulers. The task scheduler is part of the application. The task scheduler used for this project is the Static Order (SO) scheduler. The order of execution of the tasks in the SO scheduler is always fixed.

To work with CompOSe and DVFS, the Voltage and Frequency Control Module (VFCM) is needed in the hardware platform. CompOSe guarantees that each application gets exactly the specified amount of processor time despite frequency scaling.

Figure 6 gives an overview of CompOSe.

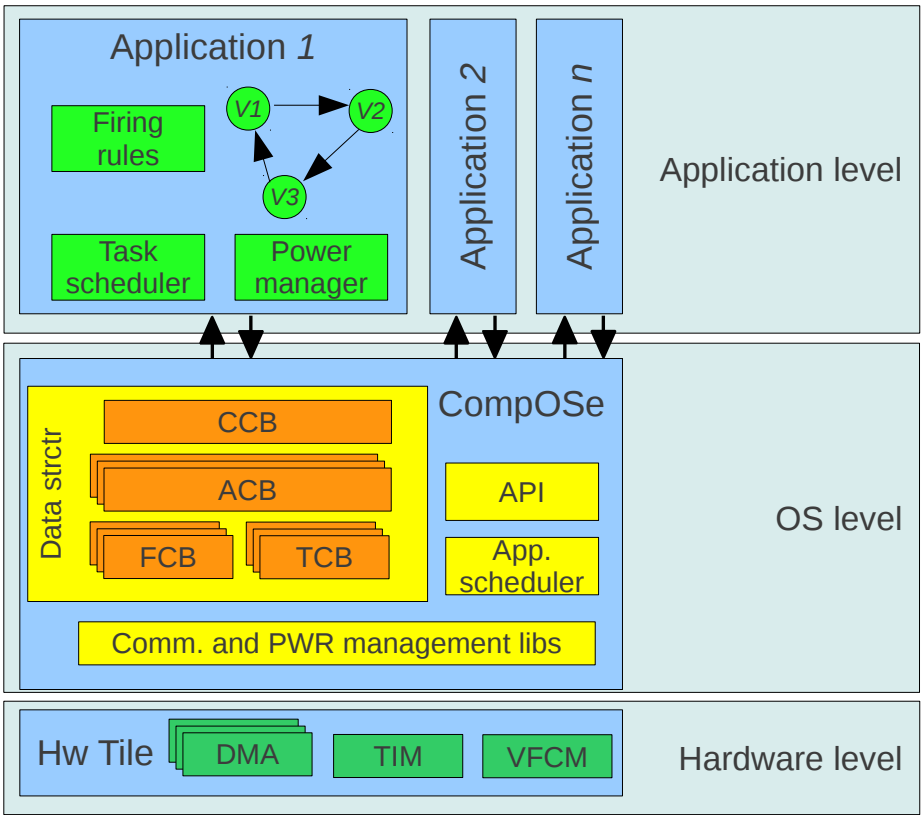


Figure 6: CompOSe

The application execution scheme in CompOSe is illustrated in Figure 7. When an application is executed, the task scheduler determines (using the firing rules, data and space availability) which task of that application is allowed to execute. After the task is selected, the data from the FIFO (First In, First Out) buffer is read. Next, the power manager function is executed, followed by the selected task. When the task has finished its execution, data is written into the tasks output FIFO's and returns to the task scheduler.

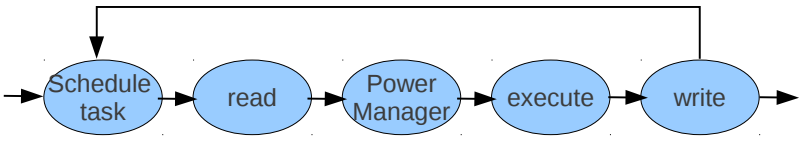


Figure 7: CompOSe Application Execution Scheme

3.3 Measuring quality

The used method of measuring the output quality of the application is given in this section. Measuring application output quality is dependent on the application. Measuring video quality is complicated [17] and different books are published on this topic [23].

In order to measure the quality of video frames (images), different objective quality measuring methods are proposed. PSNR is a metrics to measure quality. It uses pixel to pixel difference and disregards the viewing condition and the characteristics of human perception [27, 5, 20] . The Peak Signal to Noise Ratio (PSNR) is the most frequently used [19, 12]. Hybrid metrics are proposed in [22, 21] that takes the human perception into account. Zhou Wang and Bovik [19] introduced a new quality index for images and link the index to subjective quality measurements.

We use the PSNR metric to determine the produced quality of the frames and is calculated off-line by running the program on the computer. The PSNR Equation (1) uses MAX^2 (maximum value error of one pixel, based on the number of bit that is used per pixel (we have 8 bits)) and the Mean Squared Error (MSE) of the image. The MSE calculates the squared error between the produced image (K) and a reference image (I) over all the pixels and normalized over the number of pixels ($m * n$ (dimensions of the frame)). The MSE is defined in Equation (2).

$$\text{PSNR} = 10log_{10}\left(\frac{\text{MAX}_I^2}{\text{MSE}}\right) = 10log_{10}\left(\frac{255^2}{\text{MSE}}\right) \tag{1}$$

$$\text{MSE} = \frac{1}{m * n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \tag{2}$$

The H.263 decoders resulting images contain three values per pixel (Red, Green and Blue). The MSE for these frames is calculated using Equation (3).

$$\text{MSE} = \frac{1}{3 * m * n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{k\in r,g,b}[I(i,j,k) - K(i,j,k)]^2 \tag{3}$$

3.4 DVFS and calculating energy

This section defines the Dynamic Voltage and Frequency Scaling (DVFS) mechanism that is implemented in the system. DVFS is a mechanism that scales the frequency together with the voltage to reduce energy, that is enabled by the VFCM.

The CompSoC platform has multiple independent clock domains. There is one global system clock that runs at a fixed frequency. Each processor tile has its own tile clock frequency that can be changed dynamically at run time. Each tiles maximum frequency is equal to the system clock. The CompSoC platform has 16 frequency levels to switch between. Each 16 cycles of the global clock, F cycles are skipped. The resulting ratio of used cycles from the global clock is multiplied with the corresponding energy usage per cycle ($F_e(F)$), resulting in the energy used (E_{used}). This is computed in Equation (4), which is implemented in the current CompOSe version.

$$E_{used} = \frac{\#Cycles_{usedglobal} * (16 - F)}{16} * F_e(F) \tag{4}$$

$F \in 0..15, F_e(F)$Returns the energy usage at that voltage and frequency level

4 Scalable application and platform

This chapter gives the changes that need to be made in the application, changes in the CompSoC platform and the mapping of the application to the platform. Section 4.1 explains how to make the H.263 decoder scalable enabling the trade-off between execution time and quality. Section 4.2 explains the hardware instances and changes in the CompSoC hardware configuration. Section 4.3 defines the mapping of the H.263 decoder onto the hardware instances with the application and task schedulers. Section 4.4 shows the created debug infrastructure.

4.1 Scalable H.263

This section describes the variation that the H.263 decoder has by default and the scalable functions that are introduced to make the decoder adaptive.

The execution behavior of a H.263 decoder differs per frame and macro block. The execution of the H.263 is data dependent to generate the resulting frame. However, to produce a recognizable frame (image), most of the data is not necessary. We want to control the execution time and quality.

This decoder uses two different type of frames, namely I- and P-frames. The P-frames exploit the temporal DPCM (Differential pulse-code modulation) and consider motion of parts of the frames. To process P-frames, the previous frame is required in order to apply the motion compensation. For motion compensation, motion vectors are extracted from the bitstream. This step is not performed for the intra-frames. Macro blocks of inter-frames often contain less AC values, because it only encodes the differences after applying the motion compensation. Reconstructing the pixel of the inter-frames is often performed faster since less data needs to be processed. The advantage of the intra-frames is that they are not dependent on other frames. The number of AC values in a macro block of an intra-frame varies, depending on the content of the frame and quantization factor that was used in the encoder.

In order to achieve a composable system, we do not allow components that are not composable. The DDR memory controller is not composable. The decoder needs to buffer three frames. If the DDR is not available to buffer these frames, the frames need to be stored in local on-chip memory. This restricted the size of the frames. To still have a watchable movie, an up-scaler stage is added after the decoding stage. Resulting in the task graph shown in Figure 8. This figure shows the defined tasks of the implemented decoder. Macro blocks are reconstructed from the bitstream with the VLD task and inversed quantization is applied by the dQuantization task. The pixel sample values are reconstructed in the iDCT task and the frame reconstruction with the motion compensation process step is combined in the addBlock task. This task gets the macro blocks from the iDCT and the motion vectors from the VLD task. The frame is up-scaled or down-scaled and color conversion from YUV to RGB is applied in the last task. The explanation of each function can be found in Section 3.1.

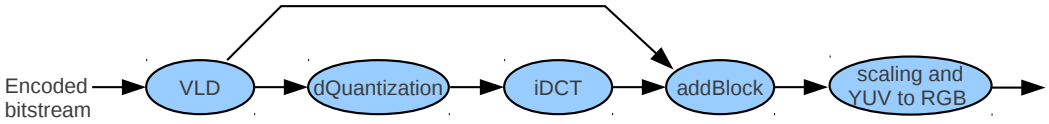


Figure 8: Task graph H.263 decoder with up-scaler

4.1.1 Scalable functions

In this section, the implemented scalable functions are defined. We introduce scalable functions in order to make the H.263 decoder adaptive. We trade-off quality for execution time. Scaling the needed amount of cycles to produce one token or frame, enables us to scale the frequency, reducing the power. Adaptive applications exist, but H.263 decoder is not one of them by default.

In the case of the H.263 decoder, the application is a slave of the encoder. The H.263 decoder has by default no scalable functions or modes. For the H.263 decoder, different mechanisms where introduced to make the application adaptive and influence the number of cycles needed to compute one token.

There exists three different mechanisms for trade-off quality for execution time in or system, these are:

1. Amount of AC values that are processed per block, explained in Section 4.1.2.
2. Dropping blocks of data, explained in Section 4.1.3.
3. Image upscaling method, explained in Section 4.1.4.

The tasks that are involved in the defined mechanisms are shown in Figure 9. Controlling the number of AC values, changes the execution time of the VLD, dQuantization and iDCT tasks. Dropping a macro block changes the execution time on all the tasks that are involved on reconstructing the frame. The up-scaler mechanism determines the execution time on the scaling and color conversion task.

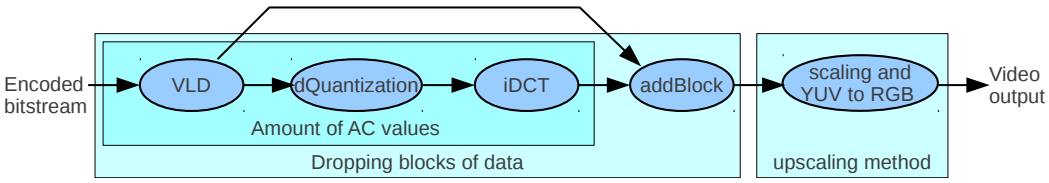


Figure 9: Influence of the adaptive function on the different tasks

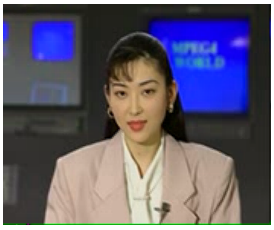
4.1.2 Ignoring AC values

This section describes the concept of ignoring AC values in the decoding stage and the consequences in the resulting output quality of the application.

The VLD function reads values out the bitstream. The dQuantization function reproduces blocks of one DC and 63 AC values. The first AC values represents low-frequency variation and have a bigger influence on the reproduced image, compared to the AC values that represent the high-frequency variation in the image. Not processing AC_{low} results in a larger quality error (low PSNR) compared to AC_{high}. Not processing AC_{high} often results in a less sharp image. When a poor up-scaler is used, the high-frequency components are less important because details are lost in a poor up-scaler. The available implemented up-scalers are defined in Section 4.1.4. The AC errors observed by a person differs on the post-processing steps that are performed at the end. When the up-scaler C or D, is applied on the movie afterwards, the artifacts are more visible.

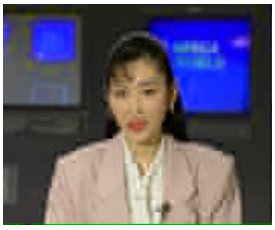
To get a feeling of the resulting visual quality of applying this method, example frames from different movies are given in Figures 10(a) to 10(l). It is the 10^{th} frame (P-frame) of each movie. Up-scaler D is applied afterwards on the movie. Because P-frames are dependent on the previous frames, the previous frames are also processed with the same amount of AC values. The images that use 63 AC values (Figure 10(a), 10(e) and 10(i)) are the original images. The quality degradation is not linear compared with the number of AC values that are taken into account, because the used number of AC values differs per macro block.

(a) Akiyo, 63 AC values processed



(b) Akiyo, 40 AC values processed

(c) Akiyo, 15 AC values processed

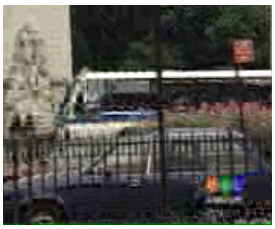


(d) Akiyo, 6 AC values processed

(e) Bus, 63 AC values processed

(f) Bus, 40 AC values processed

(g) Bus, 15 AC values processed



(h) Bus, 6 AC values processed

(i) Tree, 63 AC values processed

(j) Tree, 40 AC values processed

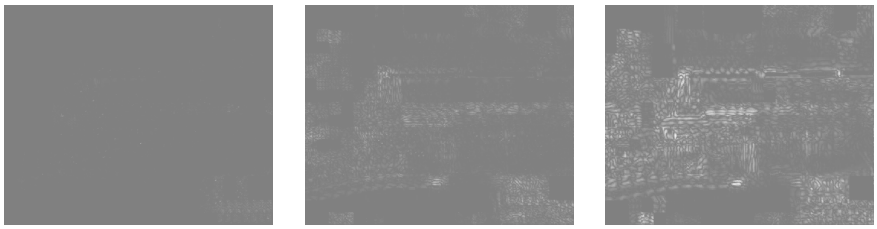
(k) Tree, 15 AC values processed

(l) Tree, 6 AC values processed

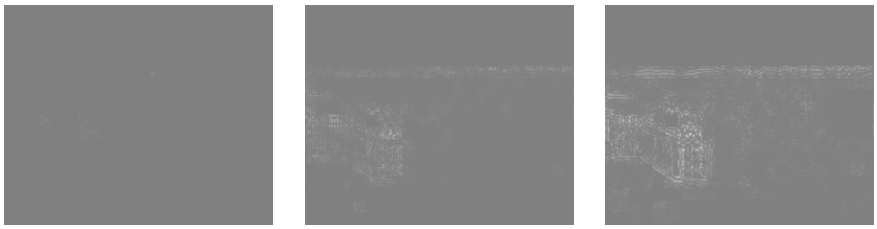
Figure 10: Resulting frame when AC values are ignored

Often movies do not have all the AC values to represent the original image. This is (also) related to the quantizer that is applied in the encoding stage. Ignoring $\frac{1}{3}$ of the AC values has limited influence on the resulting quality, see Figures 10(b), 10(f) and 10(j). This does not imply that execution time was reduced. It could be that the movies did not contain more AC values per block then the set threshold. When at most 15 AC values per block of the image are processed, still gives an acceptable result when looking at the actual image error. Figures 11(a) until 11(i) shows the actual image error that is introduced by ignoring AC values. Gray means no error, white is a positive error, and black is a negative error. Errors occur in the regions with high details and edges.

(a) Akiyo, 40 AC values processed, PSNR 45.12 (b) Akiyo, 15 AC values processed, PSNR 34.70 (c) Akiyo, 6 AC values processed, PSNR 28.84



(d) Bus, 40 AC values processed, PSNR 40.34 (e) Bus, 15 AC values processed, PSNR 31.06 (f) Bus, 6 AC values processed, PSNR 24.55



(g) Tree, 40 AC values processed, PSNR 43.36 (h) Tree, 15 AC values processed, PSNR 36.09 (i) Tree, 6 AC values processed, PSNR 30.57

Figure 11: Resulting error in the frames when AC values are ignored

4.1.3 Skipping of macro blocks

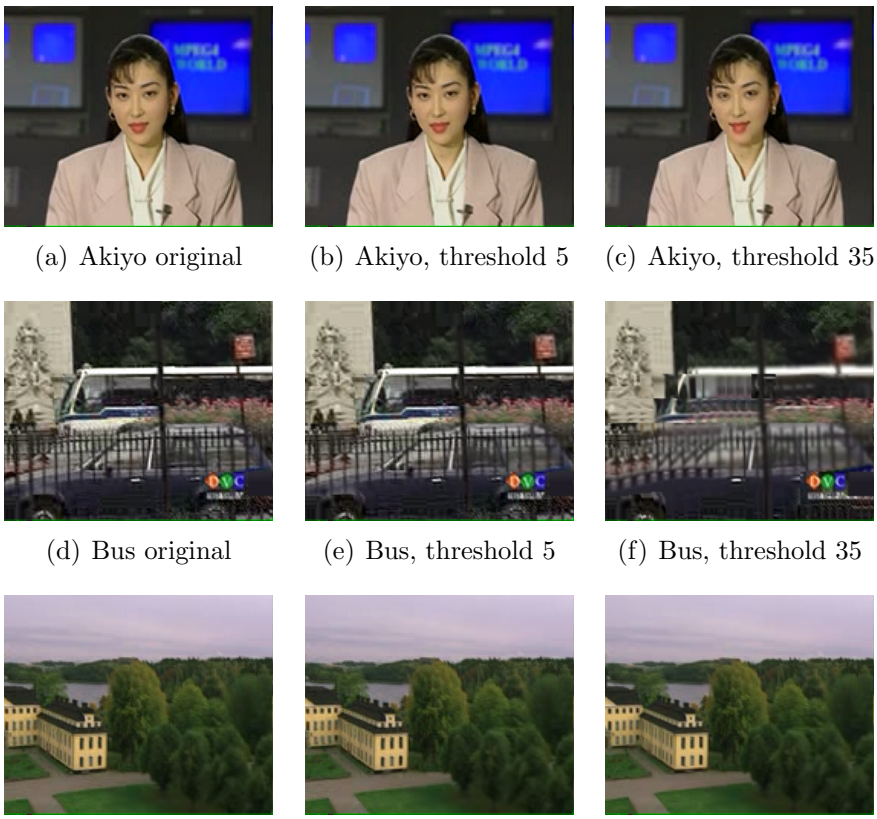
This section describes the concept of skipping macro blocks and the consequences in the resulting output quality of the application.

The application processes one macro block at a time. The contents of a macro block differs in terms of the number and the value of values. When macro blocks do not contain a lot of variation, skipping this block does not introduce a large error. This is often the case for P-frames. Determining if the block can be skipped, is done by looking at the number of AC values in that block. A threshold is used to distinguish if the block can be skipped. With different thresholds is experimented in Section 6.2.2.

Skipping part of a frame has also a risk, because the produced image is used as reference image for the next image(s). We skip only macro blocks that introduce the lowest error for the next frames. In practice, this means that this method is not suitable for I-frames because no motion vectors

are available. When a part of an I-frame is skipped, the resulting frames are unrecognizable. For the P-frames, it is suitable, because each block from the image contains data due to the motion vectors.

The resulting frames when macro blocks are skipped are shown in Figure 12(a) to 12(i). It is the 10^{th} frame (P-frame) of each movie. Up-scaler D (defined in Section 4.1.4) is applied afterwards on the movie. Because P-frames are dependent on the previous frames, the previous frames are also processed with the skipping of macro blocks. No blocks of I-frames were skipped. The threshold of skipping a macro block is set to 5. This means that whenever a block contains less then 6 AC values, it is skipped. In the second situation, the threshold is set at 35.



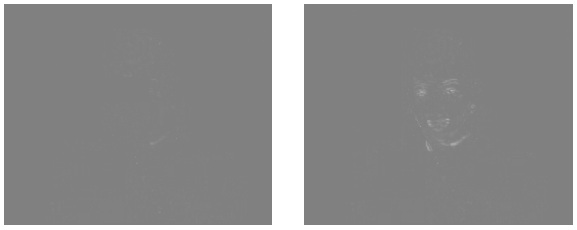
(a) Akiyo original (b) Akiyo, threshold 5 (c) Akiyo, threshold 35

(d) Bus original (e) Bus, threshold 5 (f) Bus, threshold 35

(g) Tree original (h) Tree, threshold 5 (i) Tree, threshold 35

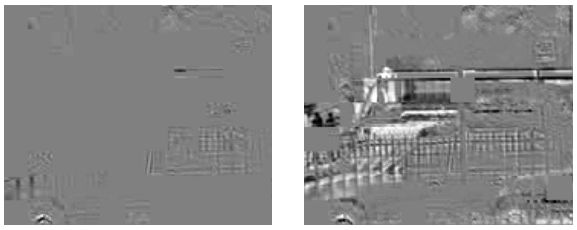
Figure 12: Resulting frame when macro blocks are skipped

The error that is introduced is shown in Figure 13(a) to 13(f). When looking at the error of skipping macro blocks is related to the motion compensation, for example 13(b), is only limited on the parts that moves. For this particular movie, only the face moves. The resulting error that is introduced by skipping of macro blocks is different compared to when ignoring AC values. The errors when ignoring AC values is related to the values in the frequency domain and the error is visible in the full macro block. The errors when macro blocks are skipped is related to the pixel domain, the visible error is the difference after the motion compensation.



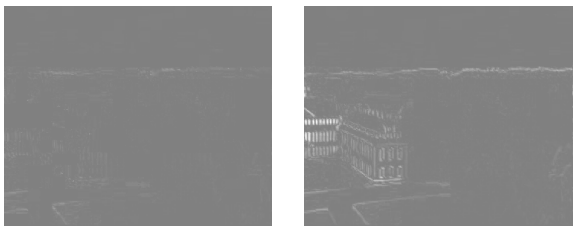
(a) Akiyo, threshold 5, PSNR 43.36

(b) Akiyo, threshold 35, PSNR 41.15



(c) Bus, threshold 5, PSNR 35.83

(d) Bus, threshold 35, PSNR 19.77



(e) Tree, threshold 5, PSNR 40.35

(f) Tree, threshold 35, PSNR 31.70

Figure 13: Error in the image resulting from skipping of MB's

4.1.4 Up-scalers

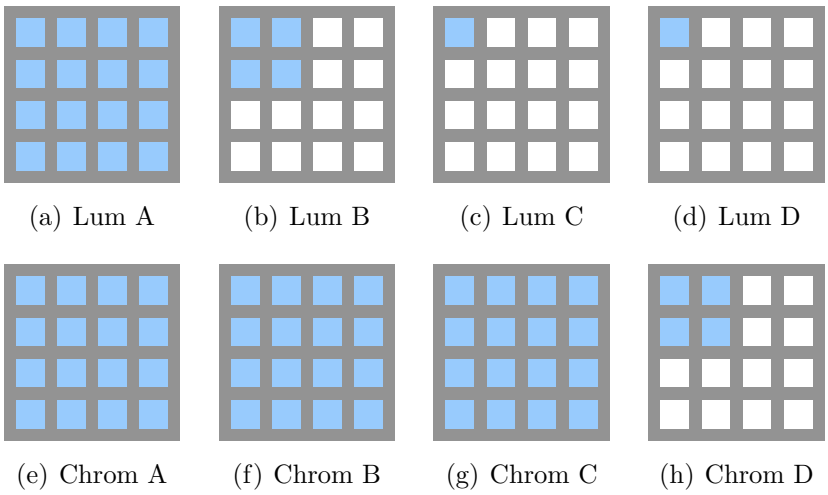
The third function that can be adapted is the frame resolution up-scaler. This section describes the implemented up-scalers and the consequences in the resulting output quality of the application.

There are different up-scale methods. We up-scale the movie by a factor of 2, because this is less compute intensive than, for example, an up-scaling of 1.7 times (then the weight per pixel differs). We choose to take an interpolation function to up-scale the images because this operation can be performed in the set time budget.

The different implemented up-scalers differ in the number of values (the grid density) that are interpolated and on the type of data (luminance, chrominance). The up-scaling step is performed on the luminance (lum) and chrominance (chrom) values instead of on the R,G and B values. Applying it on the luminance and chrominance reduces the amount of calculations and is less

sensitive to color drifts compared to when it is applied on RGB values. The following up-scalers are created and illustrated in Figure 14:

- **A** : 16 pixels get the same luminance value, 16 pixels get the same chrominance value.
- **B** : 4 pixels get the same luminance value, 16 pixels get the same chrominance value.
- **C** : luminance values are interpolated, 16 pixels get the same chrominance value.
- **D** : luminance values are interpolated, 4 pixels get the interpolated chrominance value.

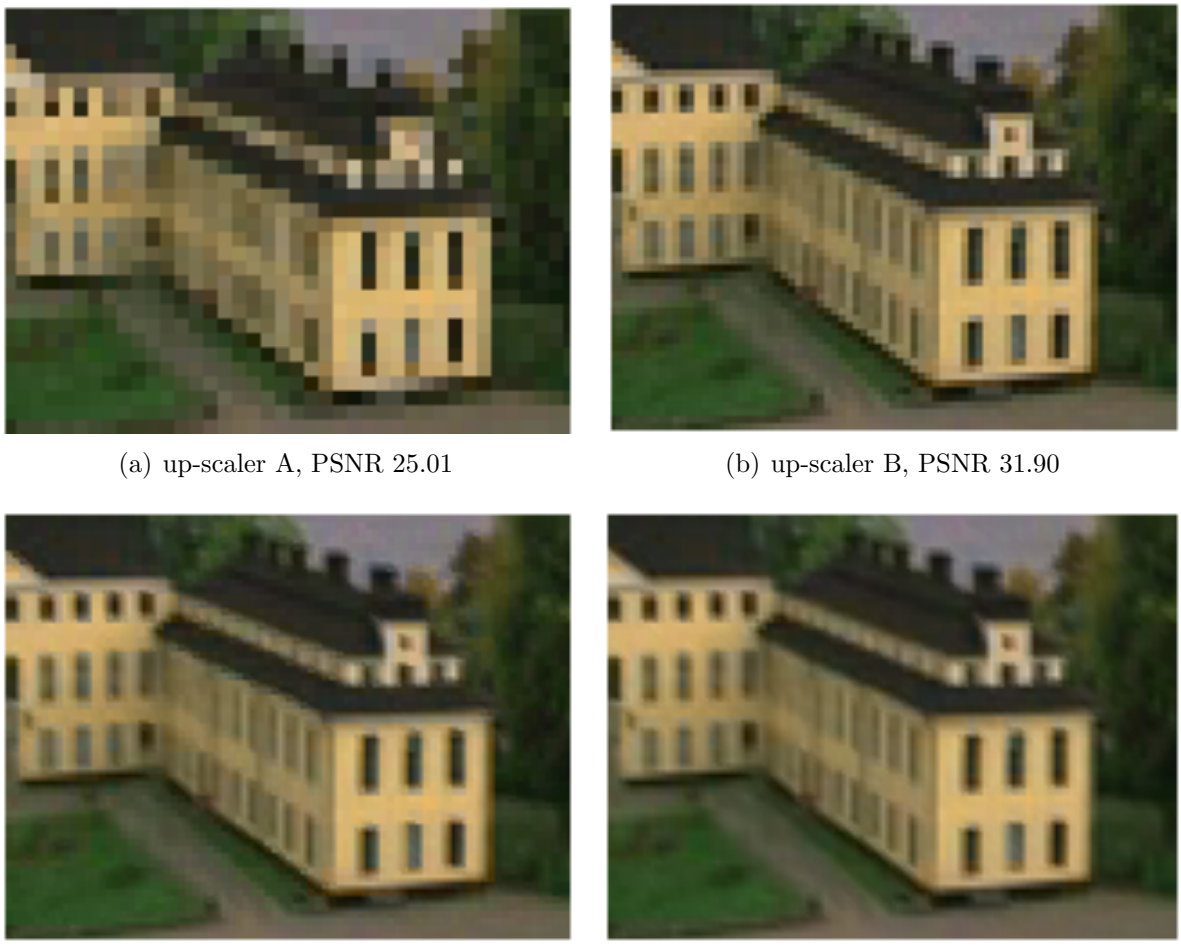


(a) Lum A (b) Lum B (c) Lum C (d) Lum D

(e) Chrom A (f) Chrom B (g) Chrom C (h) Chrom D

Figure 14: Bi-linear interpolation, grid density different up-scalers

The simplest method duplicates the values in both dimensions which results in a "blocky image" (up-scaler A). The more expensive (in terms of computation cycles) up-scaler does bi-linear interpolation between the different available points (up-scaler C and D). Because humans are more sensitive for luminance values, luminance values are interpolated in a finer grid than the chrominance values. Doing bi-linear interpolation takes additional cycles for one pixel. When the difference between the surrounding pixels is limited, interpolating does not result in a quality gain. For this reason interpolating is only applied when the difference between the two pixel values to interpolate between is bigger than 4. The resulting image quality of the different up-scalers is shown in Figure 15(a) till 15(d) with the corresponding PSNR value.



(a) up-scaler A, PSNR 25.01

(b) up-scaler B, PSNR 31.90

(c) up-scaler C, PSNR 39.10

(d) up-scaler D, PSNR 42.90

Figure 15: Visual quality, different up-scalers

4.1.5 Multi application: Picture in Picture (PiP)

We have a multi-core, multi-application system to demonstrate the concept and composability. This section gives the overview how the PiP previously presented in Section 3.1.2 is implemented. Both H.263 decoders work independently in parallel. To get the actual PiP, a multiplexer (mux) function is needed. If the applications are data dependent on sending data to the mux, results that the two applications may interfere, which is not composable. That is not what we want. To get rid of this dependence, each H.263 decoder sends the data to the mux without checking if the mux is ready to receive. The absence of this check can result in overwriting of data, but the applications are decoupled. This is similar to what is done in general in time-triggered architectures [14]. The mux is designed to be fast enough to process the data on time. It polls the input memory for new data from the applications.

The PiP system has a PiP mode, namely which movie is displayed full screen and which movie is the Picture in the Picture. We do not explicitly communicate the PiP mode between the applications, because this introduces a data dependency between the applications. The standard configuration is that H.263 decoder application 1 runs in full screen and H.263 decoder application 2 runs as the small image. If it is desired to change the PiP mode at run time, each application needs to read the PiP mode to run from memory. The PiP modes are defined in Table 1.

Table 1: PiP modes

Mode number	**Application 1**	**Application 2**
1	Full screen	Disabled
2	Disabled	Full screen
3	Full screen	Small
4	Small	Full screen

The mux needs to know where each movie needs to be displayed. To solve this problem, each application has two different data buffers, one for the full screen mode and the other for the small mode. Each buffer has a flag that indicates that new data has been written. If one of the two applications runs as a picture in picture, it results in refresh issues in the part of the screen where the small movie is displayed. To solve the overlapping problem, the applications and the mux need to know the PiP mode. A read-only memory is shared between these components, see Figure 16.

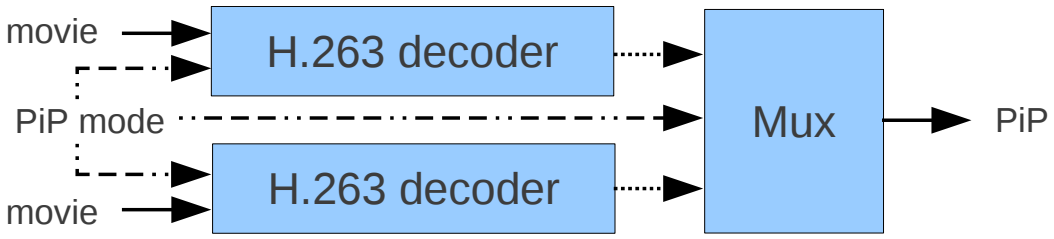


Figure 16: PiP with the mux and mode

When the mode changes, it changes immediately. The first next block of pixels is produced in the new mode and is not synchronized per frame. This creates no artifacts inside a movie block of pixels, but the current frame will contain parts of both movies.

Changing PiP mode per frame is not preferred, because each decoder can run at different frames per second. Changing after a frame introduces similar artifacts. An example that could happen at a PiP mode change is shown in Figure 17. This is at most one frame for the slowest fps that is set for one of the two decoders.

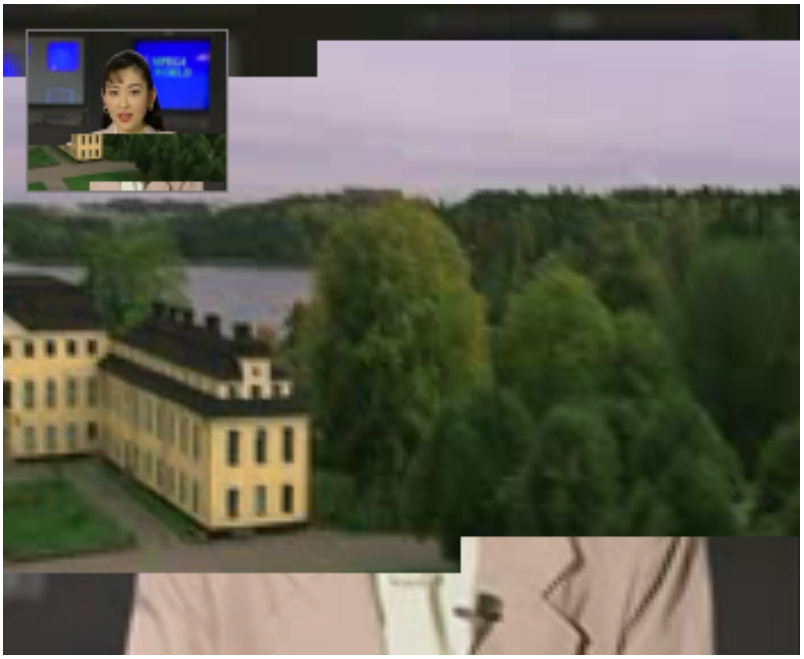


Figure 17: Possible artifacts when changing PiP mode

4.2 Changes in CompSoC

In this section, the changes in the platform and the used hardware configuration of the platform are given. We implemented two different platforms. One platform contains only composable components. This platform is used for verification of timings. Since this platform has no large memories, it cannot be used to show films. The other platform is used to show the video on the screen.

The platform to show the resulting video on the screen contains DDR and TFT controller. The DDR controller is not composable, timings cannot be guaranteed and more importantly, the time to read and write depends on the other applications, but the DDR is needed to display video on the screen. To merge the two video output streams, a mux is needed (explained in Section 4.1.5). There are two options to implement this mux, namely hardware or software. We choose to implement the mux in software on a MicroBlaze tile because a hardware video multiplexer is not available for the platform and additional debug information is added on that tile, see Section 4.4.

4.2.1 Platform instances

This resulted in the following two platforms:

A: 2 tile, 3 DMA per tile, local memory

- 2 tiles
  - 256 KB instruction memory
  - 256 KB of data memory
  - 3 DMA controllers per tile
    - 4 KB DMA_mem per DMA controller
  - 3 times 4 KB Cmem

B: 3 tile, 4 DMA per tile, local and DDR memory

- 3 tiles
  - 128 KB instruction memory
  - 256 KB of data memory
  - 4 DMA controllers per tile
    - 4 KB DMA_mem per DMA controller
  - 4 times 16 KB Cmem in
- 512 MB 64-bit DDR3-1066 running at 200 MHz, with TFT controller

In the first platform (A), the input movies are stored in local memory and the final resulting output is written to local memory. All the data is read from and written into local memory. This platform is completely composable. Figure 18 gives a schematic overview of the hardware platform. It has two processing tiles, NoC and the monitor core. The platform runs at a clock frequency of 120MHz.

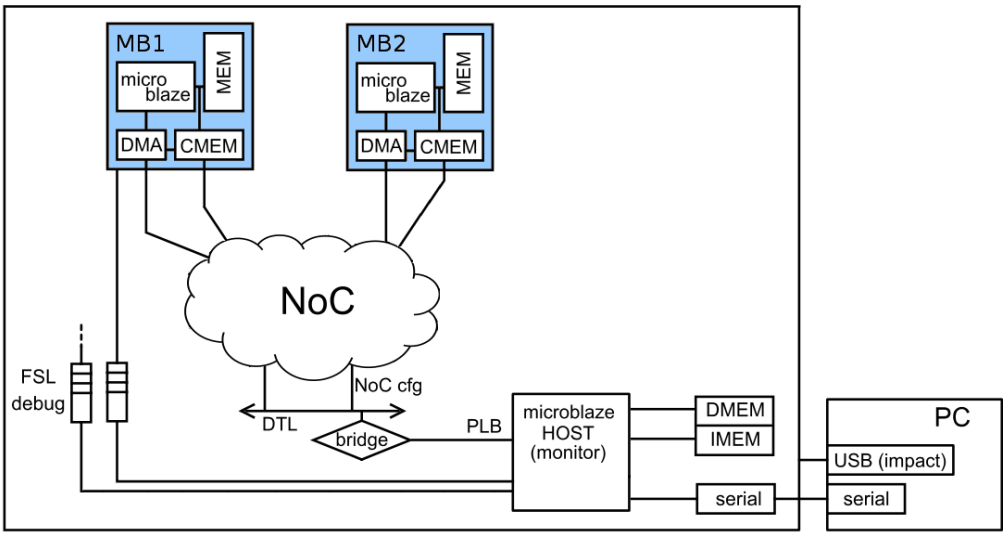


Figure 18: Hardware platform overview A

The second platform (B) additionally contains DDR memory, TFT controller and another MicroBlaze tile (MB3). The resulting movies are displayed on the screen. This platform is not fully composable because the DDR memory controller is not composable. Figure 19 gives a schematic overview of the hardware platform. It has three processing tiles, DDR and TFT controller, NoC, and the monitor core. The platform runs at a clock frequency of 100MHz.

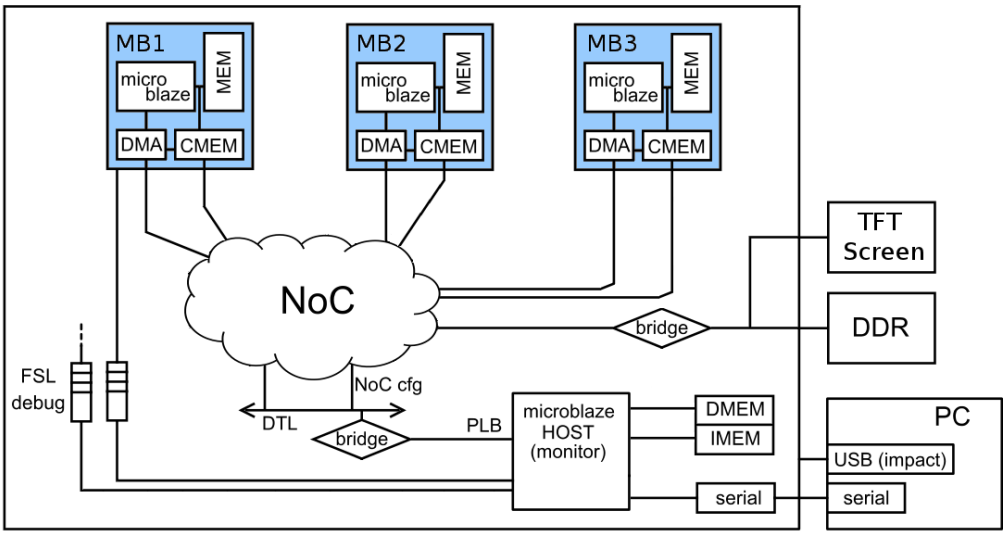


Figure 19: Hardware platform overview B

4.3 Mapping

This project is performed on two different hardware platforms. This section describes the mapping of the application to the platforms. The hardware platforms were previously presented in Section 4.2.1.

4.3.1 Mapping to the platforms

This section specifies the mapping of the applications and their corresponding tasks to the platforms. Besides specifying the mapping, the communications between the different hardware components is defined in this section.

We consider multiple distributed applications, each application mapped over multiple cores. Each core that runs CompOSe, contain at least two applications and the H.263 decoders need to be mapped over at least two cores.

Platform A

The mapping of the applications and tasks for platform A is shown in Figure 20. MB1 contains the VLD, dQuantization and iDCT tasks of both the applications. MB2 contains the addBlock and up/down-scaling tasks of both the applications. CompOSe runs on both the cores. Frequency scaling is applied on the both tiles.

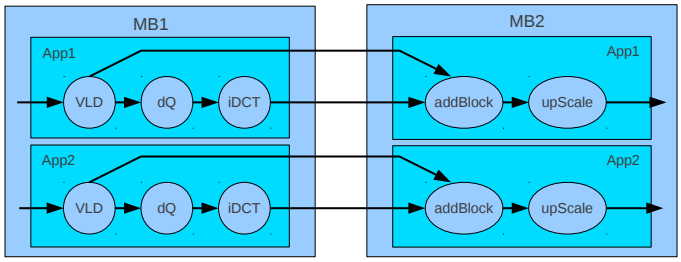


Figure 20: Mapping of the application on platform A

The network connections that are available on platform A are sketched in Figure 21. The VLD function of the H.263 decoder reads data from local memory. The VLD is mapped to the MB1 tile. Each application needs a separate DMA controller to send data to its tasks executing on the other tile to ensure composability. Application 1 uses the DMA1 controller of both the tiles and the DMA2 controller on each tile is used by the other application. This platform is without the DDR memory, the output data of both the applications is kept locally.

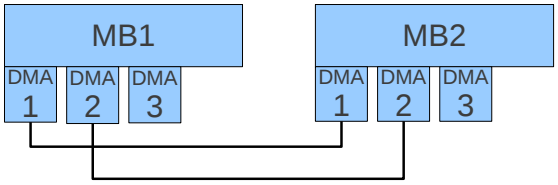


Figure 21: Used network connections in platform A

Platform B

The mapping of the applications and tasks to platform B is shown in Figure 22. MB1 contains the VLD, dQuantizer and iDCT tasks of both the applications. MB2 contains the addBlock and up/down-scaling tasks of both the applications. MB3 contains the mux function. CompOSe runs on MB1 and MB2. MB3 does not use the CompOSe operating system because the mux is the only application running on that tile. Frequency scaling is applied on the MB1 and MB2 tile but not on the MB3 tile because the energy consumption of the MB3 tile is not taken into account because we see the mux as a hardware module in the platform.

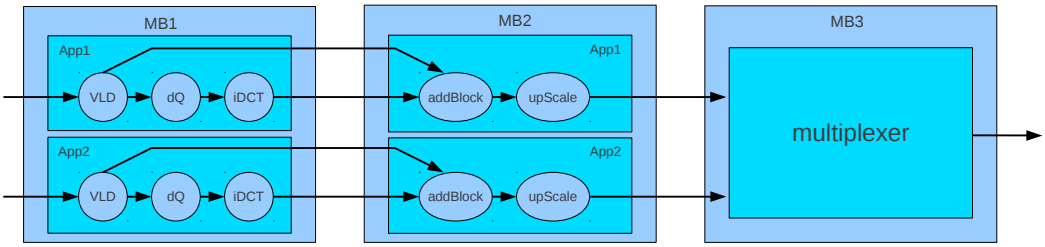


Figure 22: Mapping of the application on platform B

The connections that are available on platform B are sketched in Figure 23. The VLD function of the H.263 decoder reads data from the DDR memory. The VLD is mapped on the MB1 tile. Each application needs a separate DMA controller to get the data from the DDR. We use the connections of DMA3 and DMA4 controller of the MB1 tile to get the data from the DDR. The mux runs on MB3 tile. On this tile DMA3 and DMA4 controller is also used for sending the data to the DDR memory. Between the tiles themselves, two communication channels are available, one for each application.

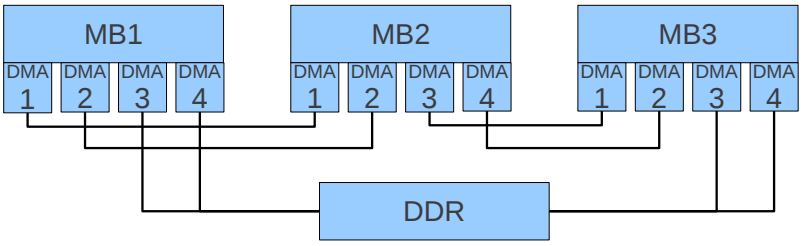


Figure 23: Used network connections in platform B

4.3.2 Application and task schedulers

For both platforms, the same application and task scheduler is used. Either H.263 decoder could run in full screen mode, or in the small image mode. This can change at run time. This results in the choice to statically give each application 50% of the available time. The platform allows dynamic changes of the application budgets, but this is not used here.

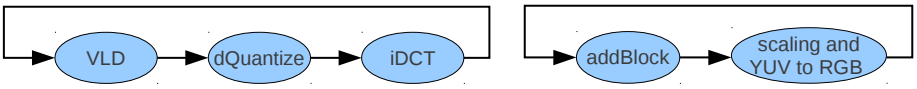
CompOSe works with a TDM table for scheduling the applications. The length of the slots in the table need to be defined (in #cycles). The first part in the slot is reserved for the operating system ($T_{systemslot}$) to execute the application scheduler, etc. The second part in the slot is reserved for the application itself ($T_{applicationslot}$).

Equation (5) shows the calculation for the application slot length ($T_{applicationslot}$). The application slot length is dimensioned to the maximum time to decode a macro block, while satisfying the minimum throughput requirement (fps_{max}). Each frame is composed of 99 macro blocks. Each application gets half of the application TDM slots. From this we subtract the operating system slot to provide enough time for scheduling before the execution.

$$T_{applicationslot} = \frac{F_{base}}{\text{fps}_{max} * 99 * 2} - T_{systemslot} \tag{5}$$

F_{base} : Base clock frequency, number of clock cycles/second

The order of task execution of the applications is always the same. The amount of data that is sent between the tasks is always fixed. This makes the application suitable to schedule the tasks with the static order scheduler. Using the static order (SO) task scheduler minimizes the switching overhead in the application itself. We use the static order scheduler. The execution order of the tasks is given in Figure 24(a) for MB1 and 24(b) for MB2.



(a) MB1 (b) MB22

Figure 24: Task schedulers of the H.263 applications

For this video decoder, after the application graph is executed once, it produces one new macro block. This behavior is always the same. The system keeps track (per tile), the number of times the graph is executed (I_{graph}). We use this counter to keep track of the deadlines for these macro blocks. Determining the deadlines is explained in Section 5.2.

4.3.3 Buffers and memory mapping

During configuration, at design time, the buffer capacity between the tasks is determined. For the buffers between tasks mapped on the same core, only one token is buffered. A token is the defined data structure that is sent from one task to another task. Larger buffers are not needed because of the Static Order scheduler and the task has a one-to-one relation between producer and consumer tasks. The amount of buffer capacity in the FIFO's between the tasks that are mapped on different tiles needs to be larger than one. The memory mapping and token sizes differs for the two platforms.

Memory mapping platform A

We choose to have a buffer capacity of 5 token between the cores. The reasoning behind this choice is explained in Section 5.2.1 and is the maximal number of tokens that fits into the 4 KB communication memory.

The video data input is defined as a predefined array that contains the bitstream. Each application has his own input array. A limited number of frames is stored in the memory. The number of frames that can be stored is dependent on the bit-rate of the movie and the movie type. Finally up-/downscaled frames are written in local memory.

Memory mapping platform B

We choose to have a buffer capacity of 16 tokens between the cores. For the video data input, the buffering is different. No buffering mechanisms that the CompOSe system provides were useful, because the number of bytes to read from the DDR is different per iteration, and the memory location to read from changes. To solve this a circular buffer containing 8 KB of data is created and the DMA controller fetches chunks of 4 KB at a time. When the application works on the first 4 KB of the buffer, the second 4 KB is filled by the DMA controller and the other way around. The first task never needs more than 4 KB to produce one macro block. Outsourcing the data transfer to the DMA controller limits the CPU waiting time on the input data.

Final up-/downscaled frames are written in a local buffer on the MB3 tile. That core is responsible for copying the data on time to the DDR.

4.4 Debug infrastructure

This section describes the debug infrastructure that is available on platform B. We wanted to have an overview of the used frequency, energy usage per frame, the quality mode each application runs at, and the remaining energy budgets available at runtime. To show the effect of quality scaling it has on the video and energy usage, graphs are shown at runtime on the screen. To show this information at runtime, additional methods are implemented to plot the information.

Figure 25 shows the debug information that is available on the screen. For each application, the average CPU usage of each core per frame and energy per frame is plotted. The energy graph shows the energy distribution between the two cores. The graphs loops around every 100 frames. Each application gets a virtual battery corresponding to its energy budget that is set before it starts. An information table is shown on the screen that shows the current frame rate of each movie, the amount of deadline misses and the current mode it runs in. This could be PiP mode or in one of the quality modes. When the battery is empty, the corresponding movie stalls. The two left most graphs correspond to application 1 and the second two graphs correspond to the other application.

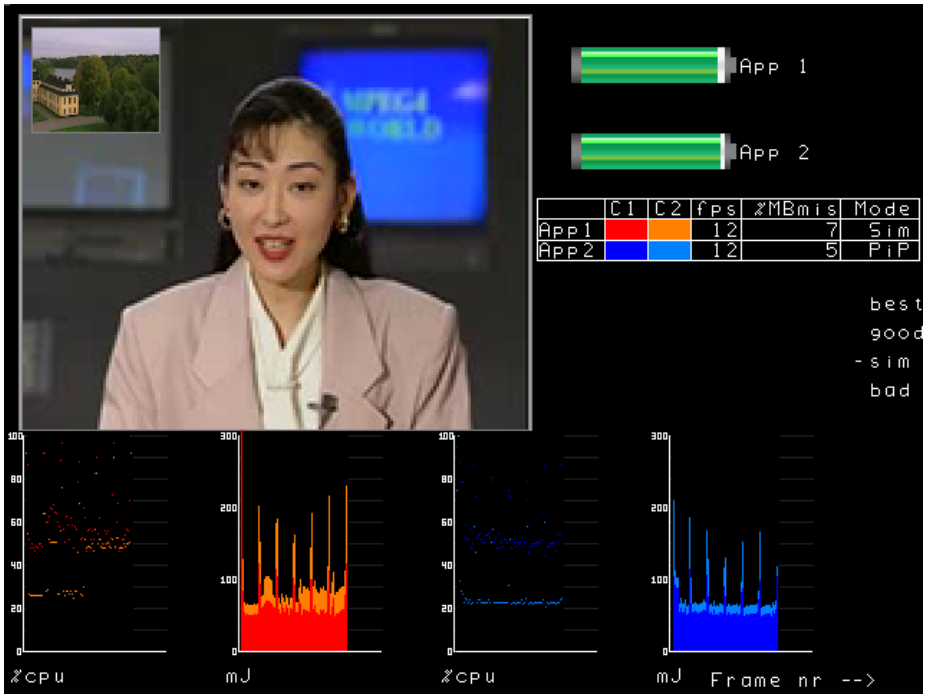


Figure 25: Debug infrastructure on the screen at runtime

5 Quality manager and slack

The slack, quality manager and policy are described in this chapter. Section 5.1 introduces the quality manager. Section 5.2 defines the time slack, the energy slack and the quality levels that are used by the quality manager. Section 5.3 defines the policy that is implemented. Section 5.4 defines the control loop of the implemented policy, including the API.

5.1 Quality manager

In this section, the quality manager is introduced. The Quality manager is implemented as a software function. The current implementation of the CompOSe operating system has a frequency scaling interface. CompOSe provides the option to the programmer to set a power manager function that is called before a task is executed. This function can change the frequency according to the system timing and time slack budgets. It cannot change data in the application or data in a task of an application. Unfortunately, it does not know the frequency the task needs to run at because the state of the application itself is not known. Calculating the required frequency level and the application quality level can only be performed in the application itself, because it is the only place the required information is known and we do this only once per graph iteration.

To change the quality level and frequency level, the first scheduled task of an iteration (determined by the SO scheduler) of the application (for MB1: VLD task, for MB2: addBlock task) calls the quality manager function with the structure that contains the quality levels and the execution time of a macro block corresponding to each quality level. The quality levels are explained in Section 5.2.3. The quality manager (see Figure 26) interacts with the CompOSe operating system and sets the frequency for the application (with an API). All tasks that belong to the same application on that core run at the same set application frequency level. The power manager sets the task frequency of an application at the set application frequency.

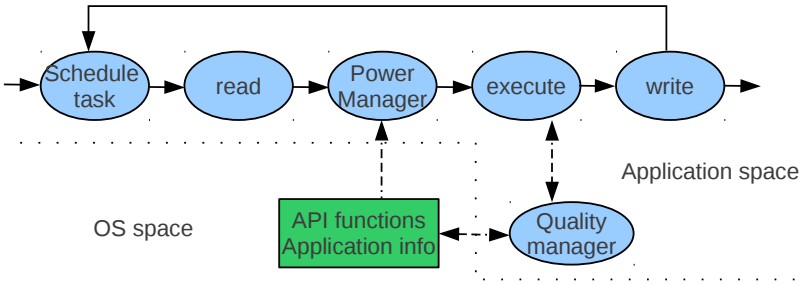


Figure 26: CompOSe application execution scheme with the quality manager

The quality manager returns if the quality level can be increased, decreased or can stay the same, based on the application info. This gives the programmer the freedom to translate the quality level to an actual application configuration. The application configuration contains the settings that each task needs to run in, e.g. adaptivity settings of Section 4.1. This freedom

is needed because each application is different. To translate the quality level to an application configuration, knowledge of the application itself is needed. Sharing the information of the quality level between the tasks of the application is done by adding the quality data inside the tokens that are communicated between the tasks. We send the additional information with the tokens themselves because then the information is also shared between the different cores that run independent instances of CompOSe.

5.2 Slack

This section defines the time slack. The time slack is used to apply frequency scaling. The calculation for the slack is based on deadlines. Determining the deadlines, the influence of the used mapping and the the global application scheduler is discussed.

5.2.1 Time slack

Application timing

The deadlines for video are normally set per frame. One frame in QCIF format contains 99 macro blocks. Each frame and macro block gets a cycle budget that it is allowed to spend on computation, determined by the throughput requirement, see Equation (5) in Section 4.3.2.

The required number of cycles per frame is plotted in Figure 27 for different movies. The movies are encoded with 64 kb/sec and 128 kb/sec. The needed number of cycles per frame (y-axis) differs. Frames 13, 25, 37 (x-axis) are I-frames. The I-frames take 50 % more time to decode than P-frames (for example, frames 14 to 24) on average. The used bit-rate to encode, movement, and details in the movies is responsible for the variation between the different movies. One thing that is visible in Figure 27 is the high peak in the startup of the bus64 (64 kb/sec encoded) and bus128 (128 kb/sec encoded) movie. It seems like the bit-rate mechanism of the encoder did not work in the beginning.

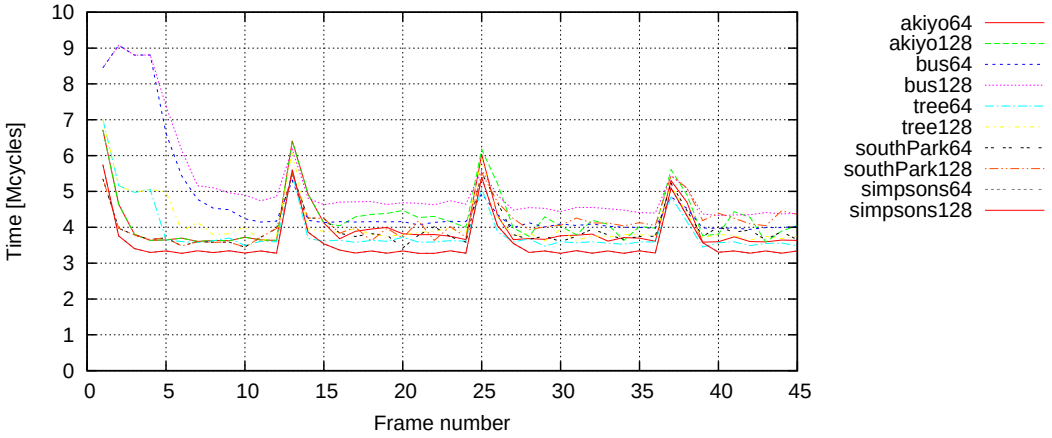
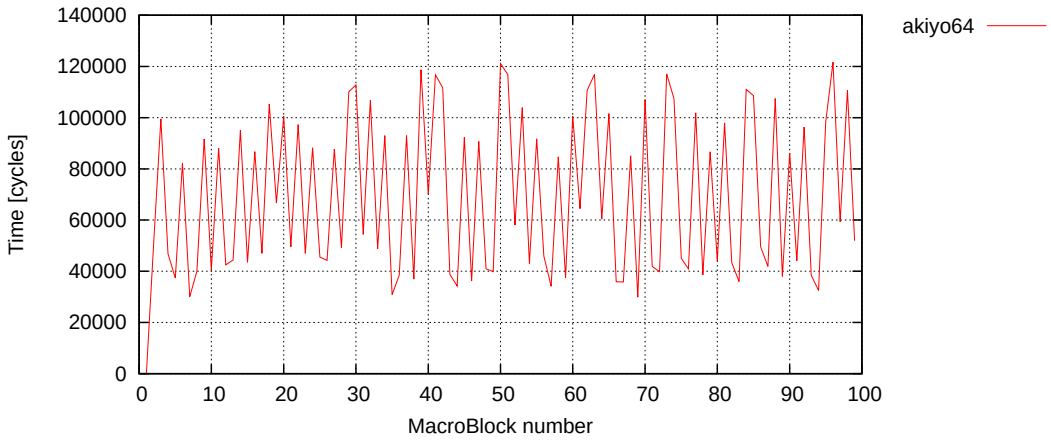
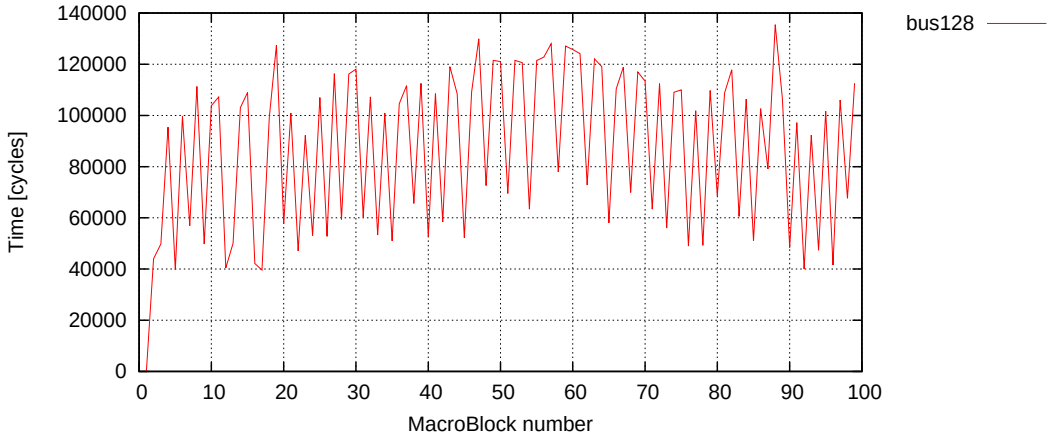


Figure 27: Required number of cycles to produce a video frame for different movies

Next, the required number of cycles per macro block is measured. Figure 28 shows the needed number of cycles to process a macro block, for the first I-frame. The variation in execution time between macro blocks within a movie is a factor 2 to 3 times. The variation depends on the data inside the macro block. Each movie has a similar variation in the execution time of a macro block. No pattern is visible between the macro blocks. Because there is no pattern between the macro blocks, we cannot predict the number of cycles needed for the coming macro blocks.



(a) Akiyo 64 kb/s



(b) Bus 128 kb/s

Figure 28: Needed amount of cycles to produce a macro block for different movies

There is a measured upper bound visible (140000 cycles) for a macro block, suitable for soft real-time (no higher number of cycles is measured, with the movie test set, for a macro block). If we use this measured upper bound for determining the throughput, we get $\approx$ 7.5fps ($\frac{100000000}{140000*99}$). In practice, this means that the system uses at most 50% of the CPU time, on average 70000 cycles is used for a macro block.

Because it is a soft real-time application, we do not want to use the worst-case situation, but base the deadlines on the average case. The average variation over a group of macro blocks is calculated next. Figure 29 shows the average required number of cycles that is needed to produce a macro block for the first I-frame for the akiyo movie.

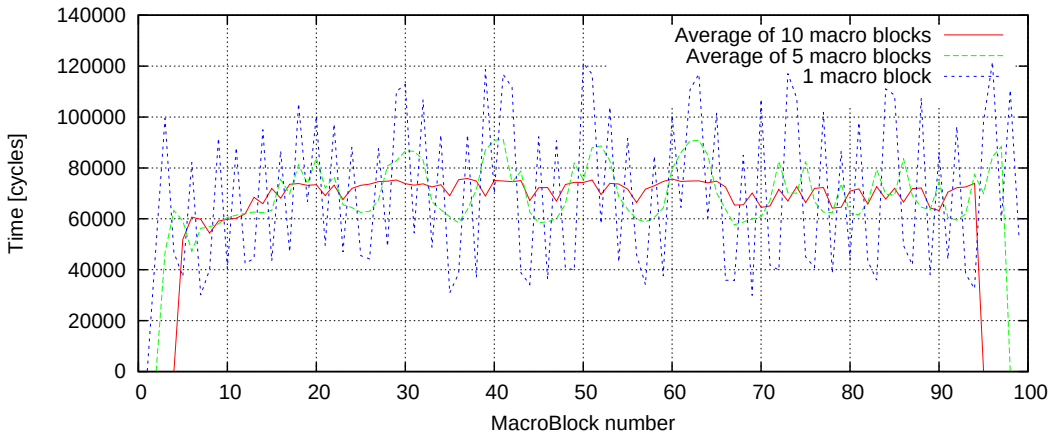


Figure 29: Average required number of cycles to produce a macro block

The deadlines are based per macro block and on the average required number of cycles. Enough buffer space is needed to cope with the variation. If the throughput is based on the worst case of the average needed cycles per 10 macro blocks we get $\approx$ 13.3fps ($\frac{100000000}{75841*99}$).

Calculating available time until deadline

In this thesis different notations are used for different (type of) definitions. Table 2 gives the description of the different notation.

Table 2: Description for the different notations used in this chapter

Notation	**Description**
$BCET_{macroblock}$	Best case execution time (for a macro block, differs per quality level)
$WCET_{macroblock}$	Worst case execution time (for a macro block, differs per quality level)
$ET_{macroblock}$	The actual set available execution time (for a macro block, differs per quality level)
T	Time budget
D	Absolute deadline
Q_{level}	Quality level.

A timing constraint per macro block is preferred, because smaller buffer is needed between the cores, compared to when it is per frame. Because each core works independently, a frame needs be buffered between the two cores. A timing constraint per macro block is preferred, because less buffer spaces is needed between the cores.

One iteration of the task graph results in one macro block, each macro block has an absolute deadline that the macro block needs to be produced. The deadlines are based on the set budget of cycles for a macro block ($T_{macroblock}$) (fixed number of cycles, set upfront).

The current activation iteration of the task graph (I_{graph}) (incremental counter), the set number of cycles of a macro block ($T_{macroblock}$), and the start time application ($TIME_{start}$), results in the absolute deadlines ($D_{macroblock}$) in time. The deadline minus current time ($TIME_{current}$) results in the available time ($T_{available}$) to process the macro block, see Equation (6).

$$\begin{aligned} D_{macroblock} &= I_{graph} * T_{macroblock} + TIME_{start} \\ T_{available} &= D_{macroblock} - TIME_{current} \end{aligned} \tag{6}$$

For firm real-time, the time available to process one macro block $\geq WCET_{macroblock}$ needs to hold to guarantee that never a deadline is missed. This $WCET$ of a macro block needs to be determined by a formal model of the application. We base this $WCET$ on the worst case measured execution time to produce one frame and divide to the number of macro blocks in one frame. Because it is measured we cannot guarantee that the set $WCET$ always holds for all movies.

Calculate frequency

The available time ($T_{available}$) and the required number of cycles for a macro block at the current quality level ($ET_{macroblock}$) is used for scaling the frequency level. The equation to calculate the frequency level (F) to run at is given in Equation (7). The frequency range that the system accepts lays between 0..15. The value is clamped by the system, see Section 3.4.

$$\begin{aligned} F &= \left\lfloor 16 - \frac{ET_{macroblock} * 16}{T_{available}} \right\rfloor, \\ F &\in (0..15), \text{ where } 0 : F_{max}, 15 : F_{min} \end{aligned} \tag{7}$$

Translating this frequency level (F) to an actual clock frequency (F_{actual}) is done by Equation (8). This equation takes the base clock frequency (F_{max}) into account.

$$F_{actual} = F_{max} * \left(1 - \frac{F}{16}\right) \tag{8}$$

The required number of cycles for a macro block ($ET_{macroblock}$) differs per application configuration (quality level), because each configuration requires different number of cycles to process one macro block. If this is known upfront, frequency scaling is applied using the difference as slack (plus the additional observed slack of the previous iteration(s)). This is implemented as a lookup

table, each quality level has an entrance in this table with the corresponding number of cycles for a macro block.

The frequency to run at is calculated per core and per macro block. This function is also suitable to apply to multiple iterations, for example per video frame.

5.2.2 Data dependency

The application is distributed over multiple cores. This section discusses the problems related to the data dependency, caused by the mapping.

Each core works independently and tries to keep as close to the application deadlines as possible with frequency scaling. The cores work independently, but have a data dependency between the VLD, iDCT and addBlock tasks. The data dependency needs to be decoupled, otherwise the cores cannot scale the frequency independently. To decouple the core, buffers are used.

Figure 30 shows what happens when frequency scaling is applied on each core. Each absolute deadline is known upfront represented by the arrows. The first macro block on the core 1 did not have slack time from the previous macro block. It will run at the highest possible frequency. When the first macro block is produced earlier than its deadline, the slack time is used by the next macro block. The next macro block is executed at a lower frequency computed by Equation (7). The same happens on core 2. When no data is ready to process, core 2 polls the FIFO's as shown by the red circles.

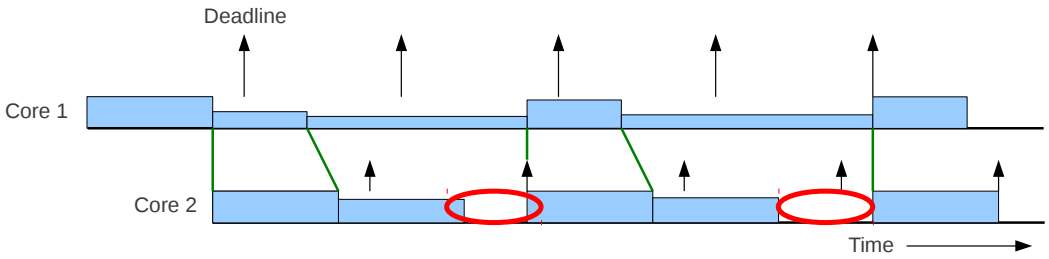


Figure 30: Data dependency between the cores

We want that each core always works on data and no time is spent on polling the FIFO's because polling the FIFO's cost energy (polling is no useful work). To resolve this, core 2 runs one macro blocks behind. The situation when core 2 runs one macro block behind is given in Figure 31. For running the second core one macro block behind, the deadline calculation on core 2 is defined as: $D_{macroblock} = (I_{graph} + 1) * T_{macroblock} + TIME_{start}$. Now, when core 2 gets his first macro block, it starts with slack time and applies frequency scaling already on the first macro block.

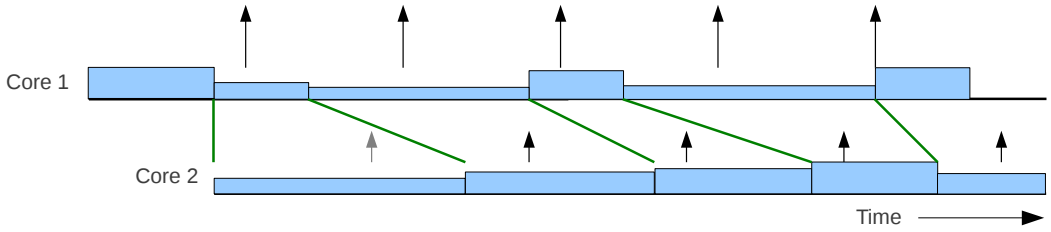


Figure 31: Data dependency between the cores, core 2 runs one macro block behind

In the examples, shown in Figure 30 and 31, there is low variation in execution time for a macro block. For the H.263, this variation is bigger for a macro block, see Figure 28. The minimum buffer capacity such that the processor never stalls waiting for data is dependent on the variation.

There are variation constraints that need to hold, namely $\frac{T_{macroblock}}{BCET_{macroblock}} < 16$. This is related to the maximum frequency scaling range. The system can at most scale down the frequency with a factor of 16. When the $BCET_{macroblock}$ is smaller than $\frac{1}{16}T_{macroblock}$ for both cores, a drift from the deadline may occur.

It could happen that decoding of a macro block takes longer than the deadline if the proper time for a macro block is not set ($T_{macroblock} \leq WCET$). The consequence is minimized by the buffers between the cores and the head start for core 1. If the buffering is insufficient, core 2 may stall on lacking of macro blocks or core 1 may stall on lacking of free space in the buffer.

Data dependency with multiple applications

We have a multi-application system. Each application gets its own application scheduler time slot(s), see Section 3.2.2. Equation (6) that calculates the available time to decode a macro block, $T_{available}$, does not take into account the application scheduler. Part of the time is reserved for other application(s), this needs to be known to determine the available number of cycles for $T_{available}$ for the application. Only the operating system has the knowledge of the application scheduler. This problem is outlined in Figure 32. The parts shown in red are reserved for another application. The absolute deadlines (indicated with arrows) in time may lie in the slot of the other application. Currently, the operating system does not have the notion of virtual application time built in [16].

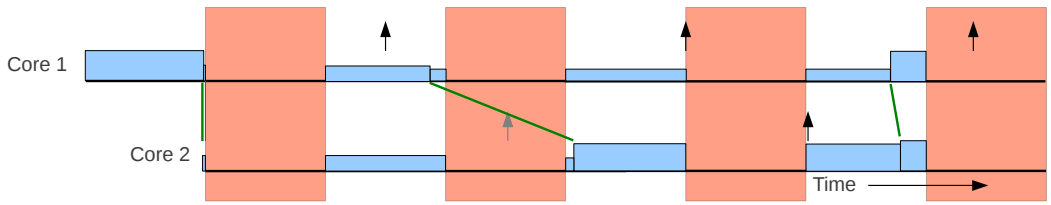


Figure 32: Data dependency between the cores, multiple applications

To get the notion of virtual application time, a function is defined that calculates the virtual application time available ($LT_{available}$) until a deadline. This function is an iterative process, see Algorithm 1. It is executed and evaluates each application slot.

Algorithm 1 Determine $LT_{available}$

```
for each application slot do
  if deadline lies in this slot then
    if it is my application slot then
      add (Deadline - start time this slot) to LT_available
    end if
  else
    if it is my application slot then
      add this slot to LT_available
    end if
  end if
end for
```

When the function is ready, the computation time it took to calculate the $LT_{available}$ is subtracted from the result. There is an upper bound on the time it takes Algorithm 1 to complete, unless deadlines can be an infinite number of slots in the future.

A simpler solution could be to specify the application slot length to exactly $\frac{1}{2}T_{macroblock}$, assuming two slots, one for each of the two H.263 decoder applications with equal application slot length. This way it is known that always exactly one slot can be used. A drawback is that both video decoders need to run at the same fps.

Evaluate the timing constrains

The method of $LT_{available}$ is implemented. We now proceed to verify experimentally how accurate the mechanism is in deadline misses. When the $LT_{available}$ is used, more energy is used because additional calculations need to be performed. But also deadlines are shifted to earlier times, hence need to run faster, which also takes energy.

The deviation of the moment the frame is produced to the absolute deadline of the frame is shown in Figure 33. Producing one frame takes $100ms$. A positive deviation means that the frame is produced too early, and a negative deviation means that the frame is produced too late. I-frames (frames 13, 25) are produced closer to the absolute deadline of the frame. The I-frames need more computation time, resulting in less slack time remaining that can be passed to the next macro block in comparison to P-frames. For the P-frames, less cycles are needed on average, resulting in higher slack time and that the frames are produced earlier than I-frames.

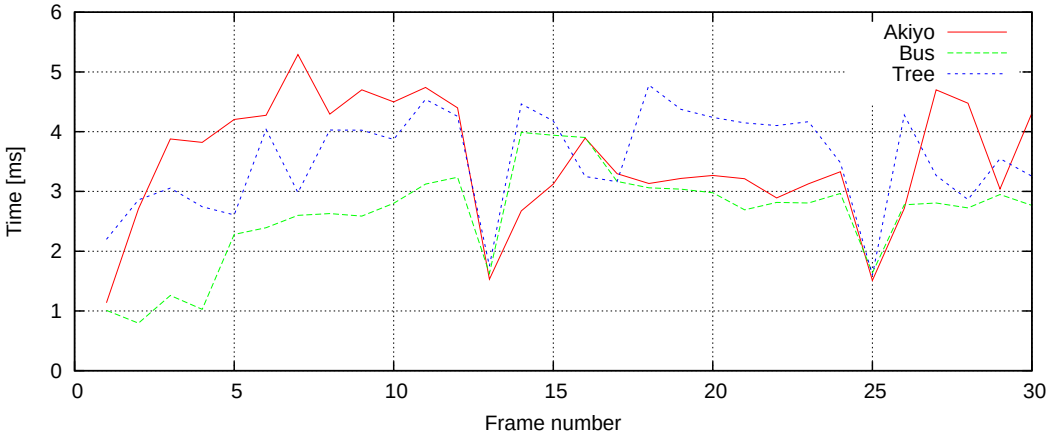


Figure 33: Deviation (Time) between frame produced and frame deadline, positive deviation means the frame is produced before deadline

5.2.3 Energy slack and quality levels

In the previous section, scaling the frequency is explained. This section defines what we mean by energy slack and how it is related to the applications energy consumption and the energy budget.

Each application gets an energy budget (E_{budget}) that is set upfront. The budget is the total budget for the movie divided per frame (the number of frames is set upfront), resulting in a target energy usage per frame. We do not divide per macro block because the granularity is too small. The difference between the energy target and the energy usage is called energy slack (E_{slack}). The energy slack is used to switch between the different quality levels. We only increase the quality when there is enough energy slack available. The energy slack computation is based on the allowed energy and the used energy (E_{used}), see Equation (9). The allowed energy is based on the energy budget per frame and the number of produced frames. The number of produced frames ($Current_{frame}$) is determined by dividing the current iteration count of the graph (I_{graph}) by the number of macro blocks in one frame (99).

$$\begin{aligned} Current_{frame} &= \frac{I_{graph}}{99} \\ E_{slack} &= \frac{E_{budget}}{\text{Total number of frames in the movie}} * Current_{frame} - E_{used} \end{aligned} \tag{9}$$

Each quality level (Q_{level}) corresponds to an energy slack region, see Figure 34. The regions are defined with thresholds. These thresholds are given by the designer of the function that is part of the quality manager.

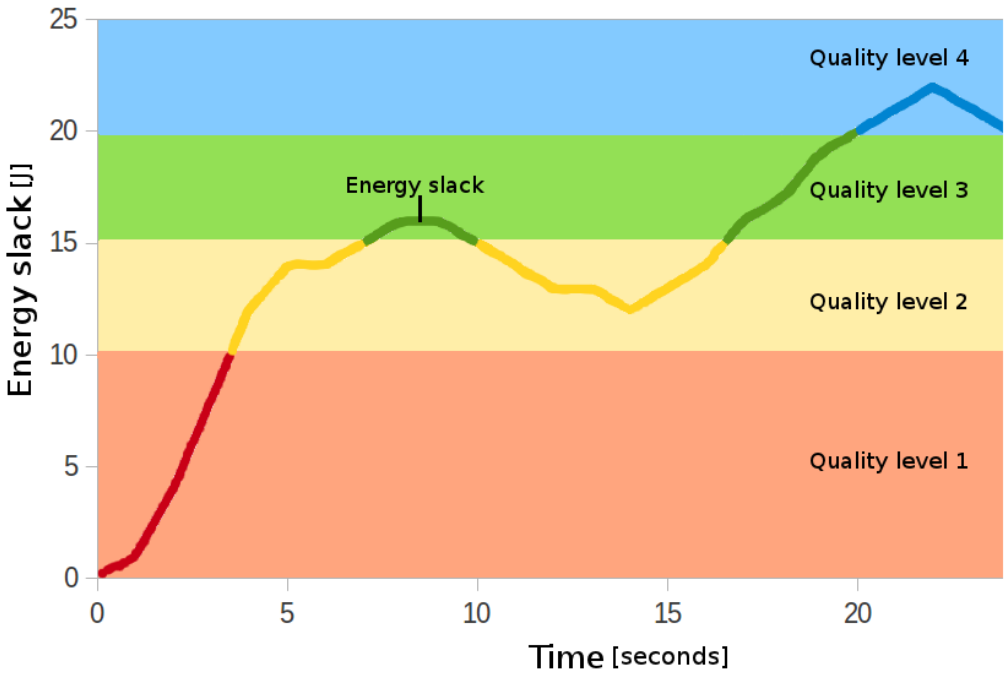


Figure 34: Quality level regions example

We choose to call this function only once per 10 frames (the frequency to evaluate the quality level is up to the programmer). The movie starts always in the lowest quality. Alternatively, energy slack could be given at the start of the movie, resulting in that the movie does not start in the lowest quality level.

5.3 Policies

This section defines policies that may be used with the quality manager. The behavior of the quality manager is determined by the policy that is implemented. A policy defines the rules that guides the decisions on DVFS and application output quality. We present three possibilities, discuss their pros and cons.

Variable energy, constant quality

There are different approaches to save energy. A way to saving energy is by applying frequency scaling for a fixed quality level. The needed number of clock cycles for a video is not known upfront. To guarantee that the end of the movie is shown, the worst-case situation is assumed. This means that for an available given energy budget, a lower quality output is selected which in practice results in a poor quality output and (low) remaining budget.

Fixed energy, variable quality

Another approach could be that we want to run the application at a fixed (lower) frequency, resulting in a constant power consumption and changing the quality level of the application at runtime. This is only possible when an application has a constant execution time in clock cycles at a fixed quality level and the gain that can be achieved is predictable for each configuration in the application (otherwise the timing cannot be guaranteed). This is not the case for the example H.263 decoder.

Average energy, variable quality

The third option is an average energy usage and variable quality. This average energy usage is related to the policy described above (fixed energy, variable quality), but is less restricted. Changing quality could be annoying for users. For example, when each frame of a movie is produced at a different quality level, it is perceived as having a lower quality than when constantly using the lowest quality level. The frequency of changing quality needs to be low, but the changing of the operating frequency needs to be done often, otherwise deadlines may be missed.

This thesis is based on the Average energy, variable quality policy. Applying policies can be done conservatively or speculatively. Conservative policies work with proven generated slack. Deadline cannot be missed with conservative policies. Speculative policies assume slack will appear in the future, deadlines can be missed. We use speculative approach because it enables better energy saving and deadline misses are not critical for the H.263 application, because it is soft real-time.

5.4 Control loop

In this section, the selected policy is translated into a control loop. The control loop is responsible for determining the clock frequency and the application quality level, the logic of which is located in the quality function, see Figure 26. The control loop given in this section is based on the "average energy, variable quality" policy and consists of two parts. The first part is responsible for calculating the required clock frequency by using the calculated time slack and the second part is responsible for calculating the quality level using the thresholds that are given to the control loop. The control loop is shown in Figure 35.

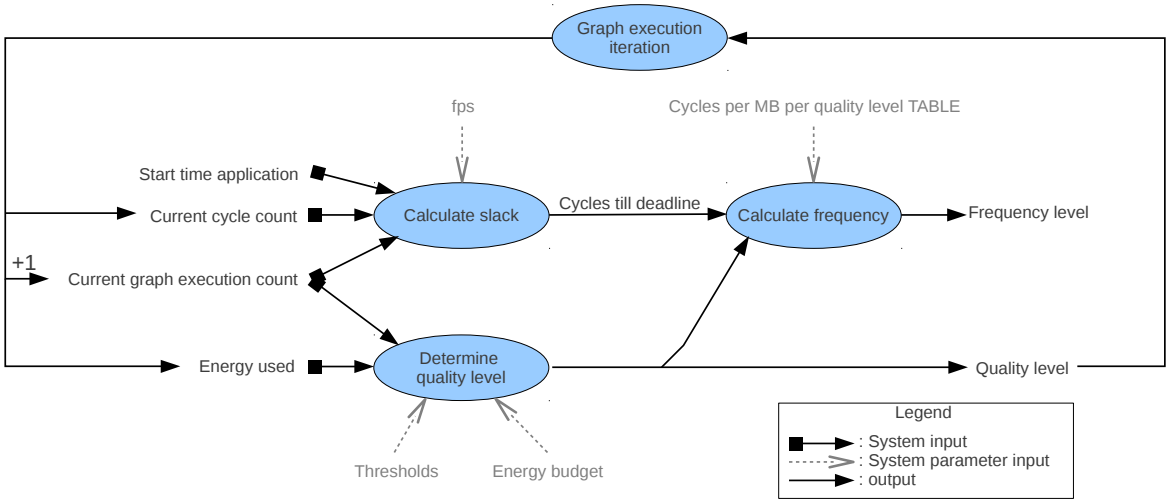


Figure 35: Control loop of the implemented policy in the quality manager

The control loop has the following in- and outputs:

- **Inputs**
  - Current graph execution (I_{graph}).
  - Start time application (cycle count, $TIME_{start}$).
  - Current cycle count (current time, $TIME_{current}$).
  - Energy used (nJ).
- **Parameters**
  - Frames per second (fps), used to determine the available number of cycles until the deadline.
  - Cycles per MB per quality level TABLE, is the number of cycles per macro block for each quality level (integer value, fixed when application starts).
  - Energy budget, for each movie (nJ).

  - Thresholds (to change quality), how aggressive the application needs to change the quality, (table with integer values, corresponding with energy slack).

- **Output**
  - Frequency level, (integer value, $\in$ 0..15).
  - Quality level, translated later in an application configuration ({AC, drop, Up-scaler}).

All the inputs are obtained from the system and the selected frequency is returned to the system. We represent the quality level as an integer value that is returned by the quality function. It is up to the programmer to make the translation from this value to an application configuration.

The parameters for each application are initialized before the application starts executing. These parameters do not change between iterations. An application may switch between a set number of quality levels. The thresholds to change the quality level are given as a table to the quality function.

5.4.1 API

In practice, to have a quality manager, the following two functions need to be called from inside a task of an application:

DetermineFrequency(...)

This function determines the frequency-level according to the progress of the application. The function returns the frequency level (F). The range is between 0 to 15.

The function requires the following inputs:

- `Iteration`

  integer, corresponds with I_{graph}, defined in Section 4.3.2.

- `Start`

  TIME, corresponds with $TIME_{start}$, defined in Section 5.2.1.

- `Current`

  TIME, corresponds with $TIME_{current}$, defined in Section 5.2.1.

- `Quality_level`

  integer, corresponds with Q_{level}, defined in Section 5.2.3.

- `Time_macro_block`

  integer, corresponds with $T_{macroblock}$, defined in Section 5.2.1.

- `Execution_time_table`

  TABLE, corresponds with $ET_{macroblock}$, defined in Section 5.2.1.

An example defined table is given here below, that contains N different $ET_{macroblock}$, one for each Q_{level}:

```
TABLE Exetuction_time_table = { 0, N,
  int ET_level_0,
  int ET_level_1,
   ...
  int ET_level_N }
```

SelectQuality(...)

This function determines the quality level, according to the available energy slack. The function returns the quality level (Q_{level}) to run at. The enumerated range is between 0 to N, defined in the table that is given to the function. It is up to the programmer to translate the returned Q_{level} to an application configuration.

The function requires the following inputs:

- `Iteration`

  integer, corresponds with I_{graph}, defined in Section 4.3.2.

- `Used_energy`

  ENERGY, corresponds with E_{used}, defined in Section 3.4.

- `Budget_energy`

  ENERGY, corresponds with E_{budget}, defined in Section 5.2.3.

- `Quality_table`

  TABLE, corresponds with Q_{level}, defined in Section 5.2.3.

An example defined table is given here below, that contains N different thresholds that marks the different Q_{levels}, see Section 5.2.3:

```
TABLE Quality_table = { 0, N,
  int TH_level_0,
  int TH_level_1,
   ...
  int TH_level_N }
```

6 Experiments and results

In this chapter, the performed experiments and the results are described. First the experimental setup and test method are defined. Then, the following experiments are performed.

- Energy and quality of the application scalable functions, earlier defined in Section 4.1.1.
  - Evaluates the solution for the problem defined in Section 1.5.1.
  - Platform used: A, earlier defined in Section 4.2.1.
  - Single H.263 decoder is considered, see Section 3.1.
  - Measurements are performed per core separate.
- Application mapping.
  - Evaluates the solution for the problem defined in Section 1.5.2.
  - Platform used: A, earlier defined in Section 4.2.1.
  - The influence of the mapping on the energy on the MPSoC platform is evaluated.
- Quality manager and policy.
  - Evaluates the solution for the problem defined in Section 1.5.2.
  - Platform used: B, earlier defined in Section 4.2.1.
  - The resulting quality of the movie for different energy budgets.
- Composability.
  - Evaluates the solution for the problem defined in Section 1.5.3.
  - Platform used: B, earlier defined in Section 4.2.1.
  - Verify that other applications cannot influence the adaptive H.263 decoder.

6.1 Experimental setup

In this Section, the test methods and the used test movies are defined. For the experiments, different types of movies are selected. The test set contains stationary movies, movies with fast movement, real-life and cartoon movies. This is important to have a variety of movies because each movie has a different required execution time for a macro block or frame. The following movies are used and is given in Table 3.

Table 3: Movie test set

Movie	**Reference name**	**Source**
Akiyo	Akiyo	http://media.xiph.org/video/derf/
Bus	Bus	http://media.xiph.org/video/derf/
In_to_tree	Tree	http://media.xiph.org/video/derf/
South park	Southpark	http://www.youtube.com/watch?v=ANfxroJ_0m0
Simpsons	Simposns	http://www.youtube.com/watch?v=PulHR2LfPVk

The settings of the encoder are given in Table 4 and the commands to encode the movies are shown in Listing Listing 1. Each movie is encoded with 64 kbit/second and 128 kbit/second. The encoder bitrate is added to the reference name in this document, example Akiyo64 (64 kbit/second), Akiyo128 (128 kbit/second).

Table 4: Movie coding settings

Parameter	**Value**
Codec	H263
Width	176 (352 upscaled)
Height	144 (288 upscaled)
Bit-rate	64 and 128 kbit/second [13]
Frames per second	20 fps
Disabled modes	pbframes, advanced MC

```
1  ffmpeg -i bus_qcif_15fps.y4m        -b 64k  -f h263 -s qcif bus64.h263
2  ffmpeg -i bus_qcif_15fps.y4m        -b 128k -f h263 -s qcif bus128.h263
3  ffmpeg -i akiyo_qcif.y4m            -b 64k  -f h263 -s qcif akiyo64.h263
4  ffmpeg -i akiyo_qcif.y4m            -b 128k -f h263 -s qcif akiyo128.h263
5  ffmpeg -i in_to_tree_420_720p50.y4m -b 64k  -f h263 -s qcif tree64.h263
6  ffmpeg -i in_to_tree_420_720p50.y4m -b 128k -f h263 -s qcif tree128.h263
7  ffmpeg -i simpsons    -f ffflv      -b 64k  -f h263 -s qcif simpsons64.h263
8  ffmpeg -i simpsons    -f ffflv      -b 128k -f h263 -s qcif simpsons128.h263
9  ffmpeg -i southPark   -f ffflv      -b 64k  -f h263 -s qcif southPark64.h263
10 ffmpeg -i southPark   -f ffflv      -b 128k -f h263 -s qcif southPark128.h263
```

Listing 1: Ffmpeg command to create the testset

Quality related experiments are performed off-line by executing the application on the computer. Video quality is determined by computing the PSNR (per frame), as previously explained in Section 3.3.

The energy consumed per frame is given in Joules $[J]$. Calculation of the energy is explained in Section 3.4. Energy consumption is measured at run-time on an instance of the test platform executing on an FPGA.

6.2 Evaluate application scalable functions

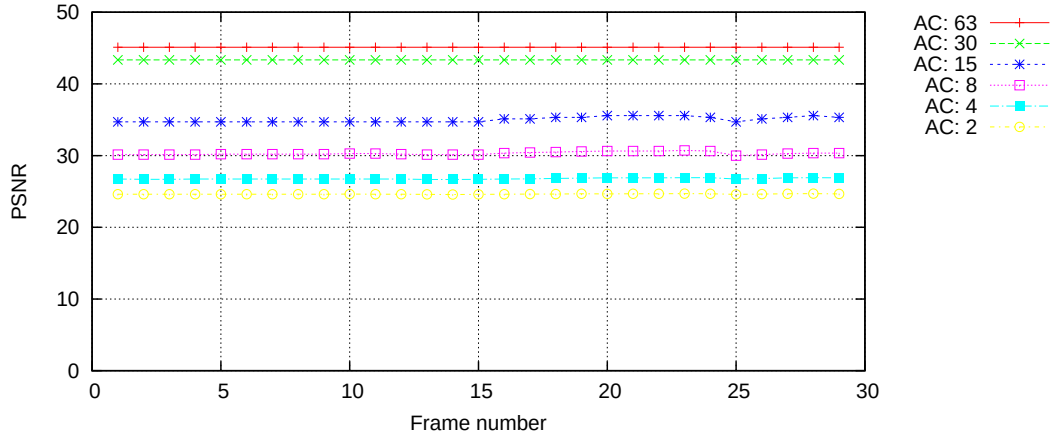
In this section the individual scalable functions are evaluated in terms of energy usage and output quality. DVFS is applied on one core at a time. This choice is made to exclude the interference that may occur with data dependencies. The data dependency is related to the mapping of the application onto the platform and is evaluated in Section 6.3. We start by analyzing each mechanism individually. Afterwards the results of the individual mechanisms are combined to find pareto points (optimal trade-off points).

6.2.1 Ignoring AC values

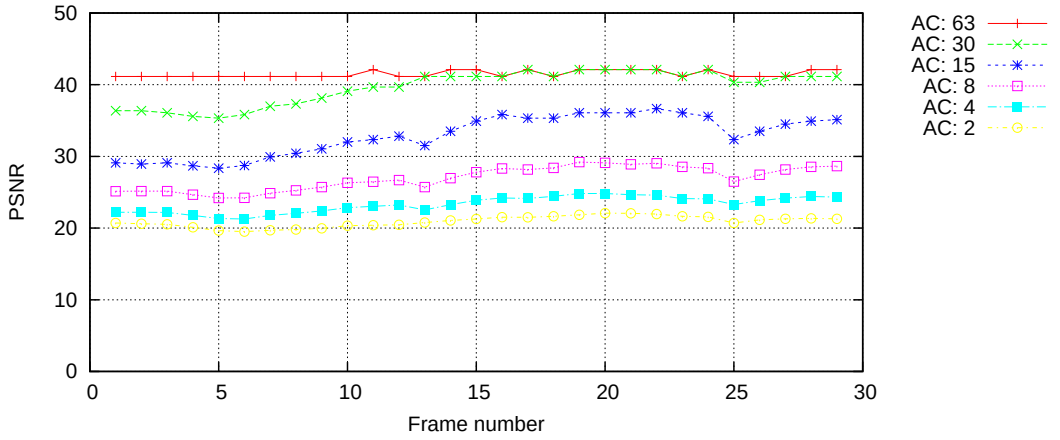
This section presents the resulting quality and the energy usage of the movie frames when AC values are ignored. The method of ignoring AC values is previously explained in Section 4.1.2.

Quality

Ignoring AC values negatively influences the resulting quality. The quality of the individual frames in a movie, when the number of AC values to process is changed, is shown in Figure 36(a) and 36(b). Decreasing the number of AC values to process, decreases the quality rather evenly over the movie frames in time. The quality for the Bus128 movie increases in time, for the situations where less AC values are processed. This is due to that the frames, for example frame 16 until 24, contains always less then 30 AC values in a macro block. This is the result of the encoder, that choose to use a higher quantization value (controlled by the bit-rate mechanism in the encoder), resulting in less AC values.



(a) Akiyo128

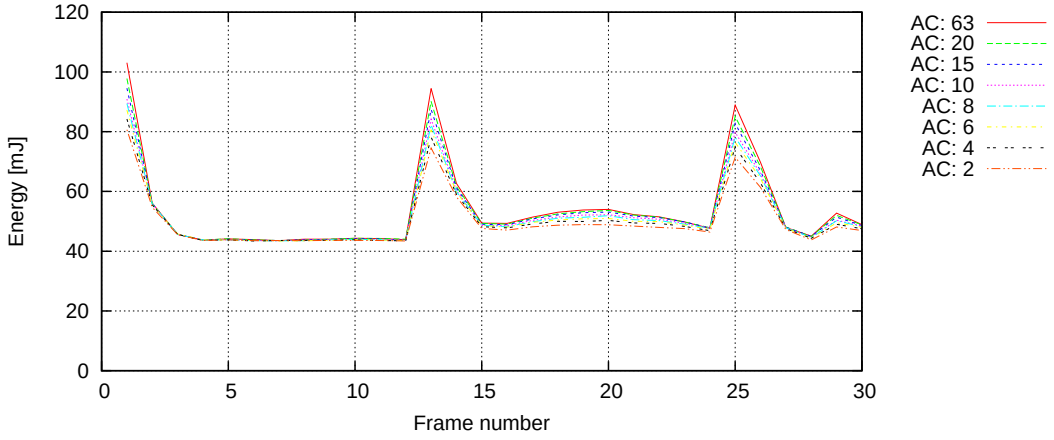


(b) Bus128

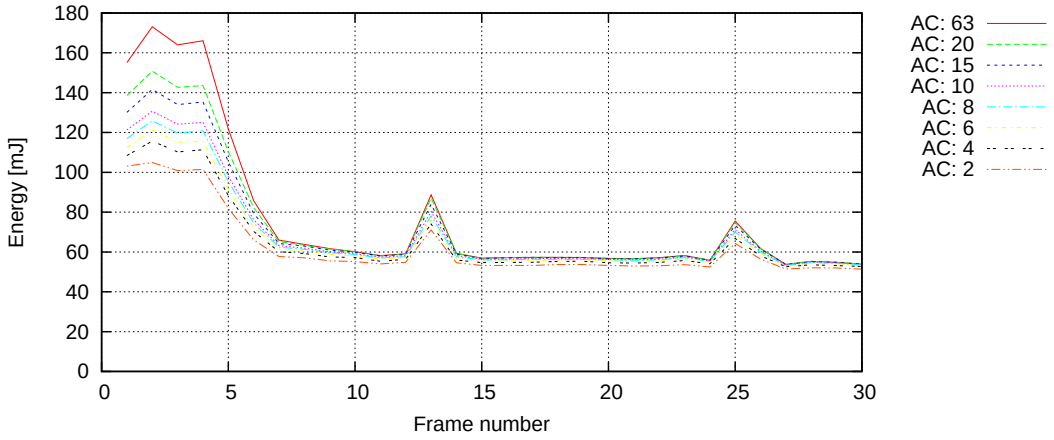
Figure 36: Resulting quality per frame depending on the number of AC values to process

Energy

The influence of the ignoring AC values (defined in Section 4.1.2) on the energy usage is evaluated in this section. Figure 37(b) and 37(a) shows the resulting energy (y-axis) per frame (x-axis) for the movie Akiyo128 and bus128 on the first core. Both figures show spikes that correspond to I-frames. The magnitude of these spikes are limited by using less AC values. For P-frames, the energy savings when fewer AC values are taken into account is reduced by at most 5 % (difference for the P-frames between AC:63 and AC:2). On average the energy is reduced (difference between AC:63 and AC:2) by 8% for the Akiyo128 movie and 22.8% for the Bus128 movie measured over the 30 frames. The larger gain in energy reduction for the Bus128 movie is due to the startup part, compared to the Akiyo128 movie. The gain in the resulting energy usage is dependent on the amount of AC values.



(a) Akiyo128



(b) Bus128

Figure 37: Resulting energy usage, number of AC values to process

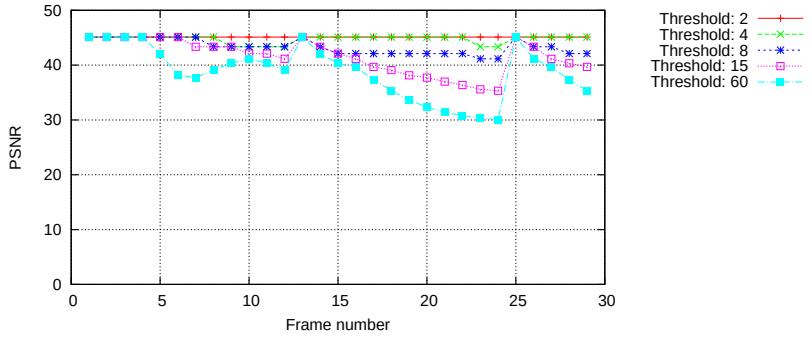
In conclusion: the energy savings is on average between the 8% (Akiyo128 movie) and 23% (Bus128 movie). Combined the quality and energy, this method does not seem interesting. In these tests, up-scaler D is applied on the movie, the results differ in terms of quality degradation and energy usage with different up-scalers, see Section 6.2.3.

6.2.2 Skipping macro blocks

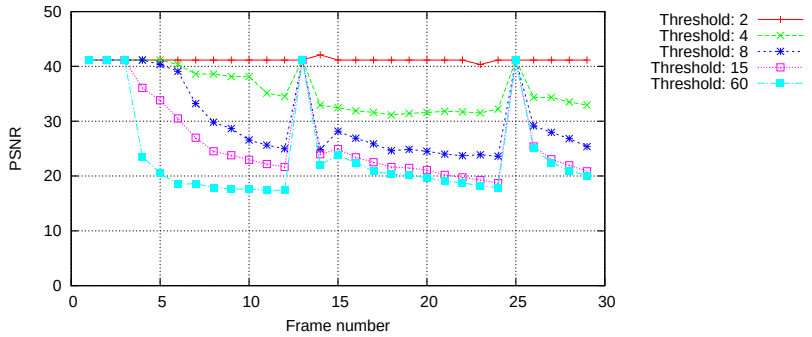
In this section, the influence on skipping parts of an P-frame is investigated. We drop a macro block, when the number of AC values in that macro block, is lower than the set threshold. The mechanism to skip macro blocks is described in Section 4.1.3.

Quality

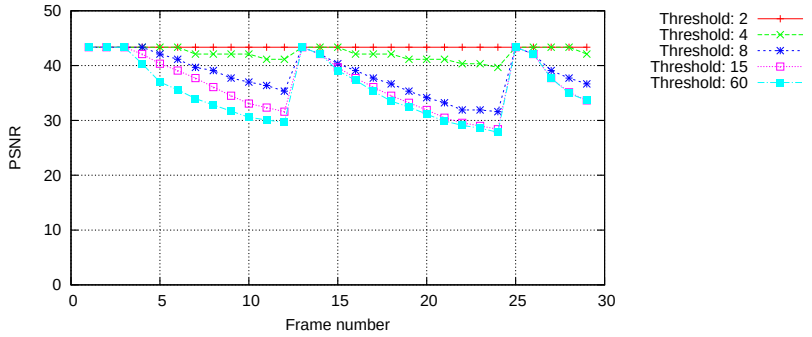
The resulting quality per frame is evaluated after skipping macro blocks. The motion vectors are decoded and applied in the addBlock function. The values in the macro block itself are ignored according to the threshold. The threshold determines if the block can be skipped. The influence of the threshold on the quality per frame is shown in Figure 38(a) to 38(c). The noise, introduced by ignoring a macro block, is passed to the next frames and cannot be restored for the other coming P-frames. Figure 38(c), the resulting quality (y-axis) is given per frame (x-axis) for the movie Tree128. The first three frames, 13^{th} and 25^{th} frame are I-frames. For the I-frames, no skipping of blocks is performed. There is therefore no degradation of quality. The quality for every next P-frame after an I-frame decreases. This effect is visible in Figure 38(a).



(a) Akiyo128



(b) Bus128



(c) Tree128

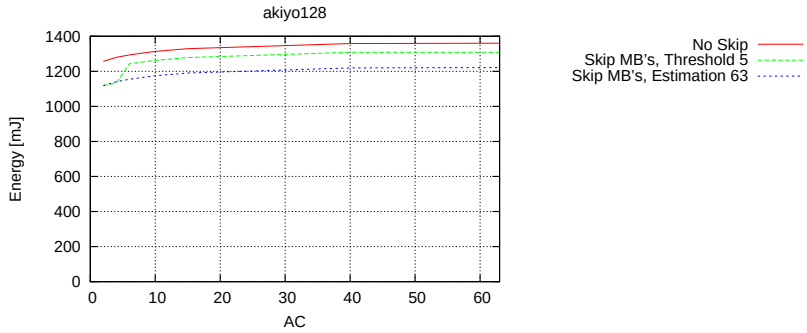
Figure 38: Influence on quality when skipping macro block, with different threshold

The difference between frames in time (x-axis, Figure 38(a)) for the Akiyo128 movie is minimal. The Tree128 movie has a smooth slow movement, and the Bus128 movie has rapid movement of objects in different directions. More values are encoded in the macro blocks for the Bus128 movie because it could not be coded by only the motion vectors. Ignoring the macro blocks for the Bus128 movie, resulted in rapid quality degradation.

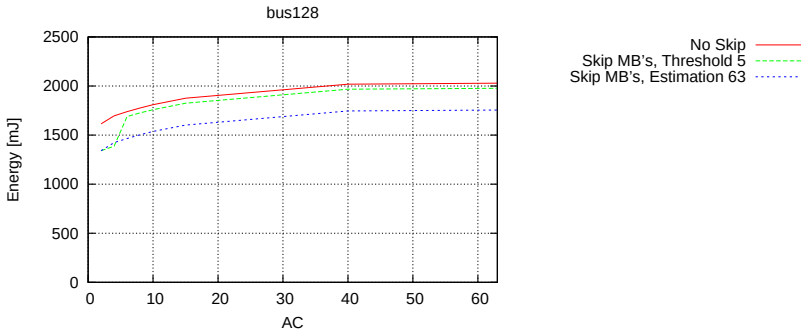
With threshold 60, only the motion vectors are processed. The Akiyo128 movie has an average (measured over 30 frames, threshold:60) PSNR of 38.8 and the Tree128 movie has an average PSNR of 35.7. But for the Bus128 movie an average PSNR of 23.7 is measured.

Energy

Figure 39(a) and 39(b) shows the resulting energy usage on the first core (y-axis), to process 25 frames for a different number of AC values (x-axis, related to the ignoring AC values mechanism, defined in Section 4.1.2) with and without skipping macro blocks. In this test, the threshold to skip a macro block is set at 5 and an estimation is given with threshold 63. With threshold equal to 63, only the motion vectors are applied and no macro blocks are processed. Applying ignoring AC values and skipping macro blocks methods gives the best results.



(a) Akiyo128



(b) Bus128

Figure 39: Energy usage when ignoring AC values and potentially skipping macro blocks

The energy saving with the skipping macro blocks method is almost constant, compared to no skipping, for AC $\geq$ 6 (x-axis in Figure 39(a) and 39(b)). This is due to the threshold used. The number of AC values to process (ignoring AC values method, previously defined in Section 4.1.2) is alway performed. The increase in the number of AC values that are taken into account (x-axis), increases the used energy. If the macro blocks contain less than 6 AC values, that blocks is ignored. This explains the energy drop when the skipping of macro blocks is performed for AC $<$ 6. This means that for AC=2, no data in the macro blocks is processed and taken into account for the reconstruction of the image. The energy difference between no skipping macro blocks and the energy used when skipping is applied (for 2 AC values), is equal to the energy used to process the data inside the macro blocks.

Figure 39(b) shows that the the Bus128 movie contains more AC values than the Akiyo128 movie because the curve increases until about 40 AC values. After the 40 AC values, the curve flattens out. The energy savings achieved by this method is on average (measured over 25 frames) for the Bus128 movie: 6% (difference between no skipping and skipping with threshold 5) and estimated energy savings of 15% (difference between no skipping and skipping with the estimated threshold 63). For the Akiyo128 movie: 5.5% with threshold of 5 and estimated energy savings of 10% with threshold 63.

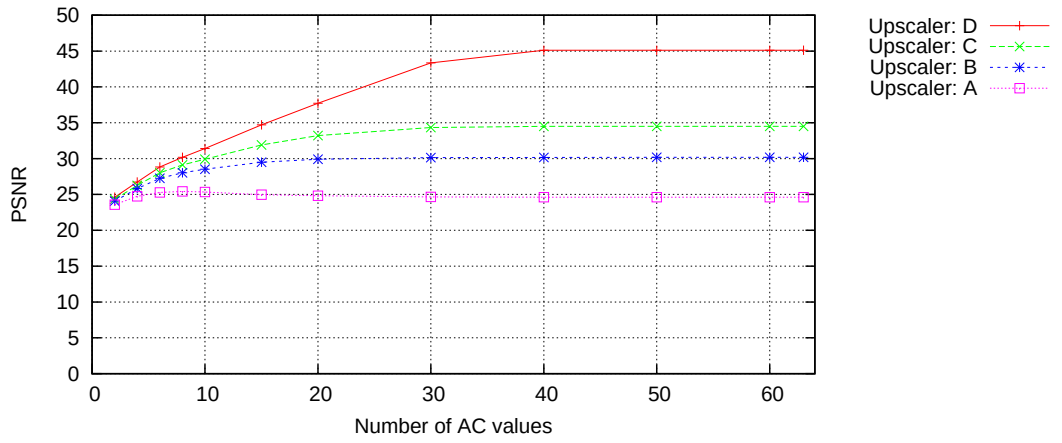
6.2.3 Up-scaler

In this section, we evaluate the quality and energy trade-off with different up-scalers ranging from A to D. The different up-scalers were previously specified in Section 4.1.4.

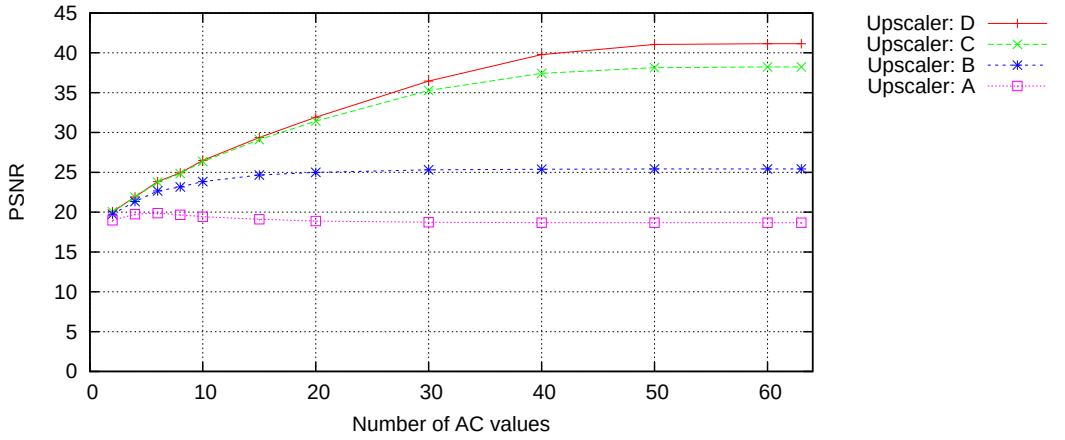
Quality

The average PSNR over the first 25 frames is measured while changing the number of AC values to process (described in Section 4.1.2) and the up-scaler for each movie.

Figure 40(a) and 40(b) shows the resulting quality (y-axis) when the number of AC values to process (x-axis) is changed. This experiment is performed four times and each time with a different up-scaler. It can be seen from the graphs presented in Figure 40(a) that the quality of the decoded frames increases with the number of AC values. There is one exception, namely that using less AC values result in higher quality for up-scaler A. Up-scaler A uses the same value for 16 pixels.



(a) Akiyo128



(b) Bus128

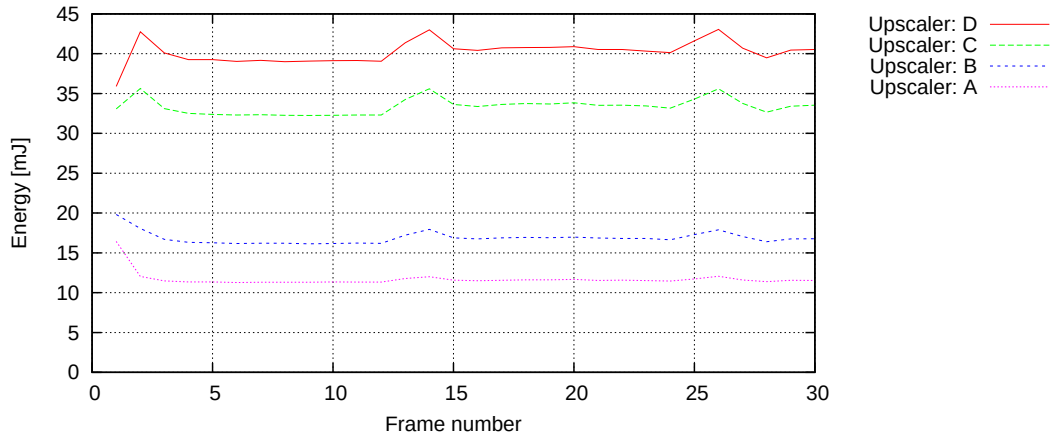
Figure 40: Resulting overall quality different up-scalers and number of AC values to process

The influence of the number of AC values to process on the PSNR is different per up-scaler. For up-scalers D, the PSNR is a factor two higher with all the AC values processed, compared to the PSNR were only two AC values are processed. With up-scalers A, the quality (in terms of PSNR) difference is at most 5% for different AC values.

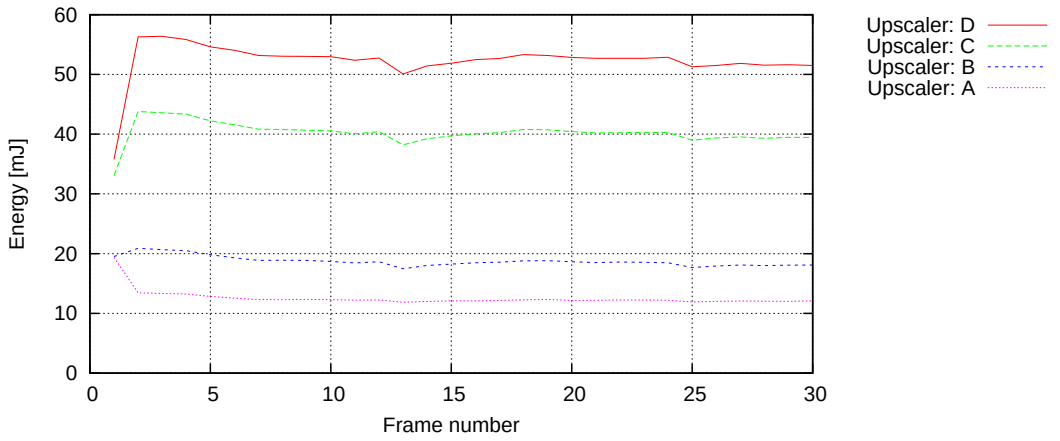
When looked at Figure 40(a) changing from up-scaler D to up-scaler C, the PSNR decreases 10 (with 63 AC values). For the Bus128 movie (Figure 40(b)), changing from up-scaler D to up-scaler C, the PSNR decreases only 3 (with 63 AC values). The difference between the D and the C up-scaler is that the chrominance values in up-scaler D are interpolated, and this is not performed in the up-scaler C. The difference needs be in the color transitions in the images for the different movies.

Energy

Figure 41(a) and 41(b) show the used energy (y-axis) per frame (x-axis) per up-scaler, only for the second core, where the up-scaler function is implemented. The number of AC values is set at 63, and no skipping of blocks. Similar energy results where measured for the other movies. It can be seen in Figure 41(a), the energy usage for up-scaler D is approximately three to four times the energy usage of up-scaler A.



(a) Akiyo128



(b) Bus128

Figure 41: Results different up-scalers

Figure 42, shows the PSNR per Joule (for the second core) for the different up-scalers. The up-scaler A gives the most quality per Joule, up-scaler C and D the least. For the Akiyo128 movie, up-scaler C gives less PSNR per Joule because of the quality drop. This is because this movie gains more benefit from interpolating the colors, compared with the Bus128 movie.

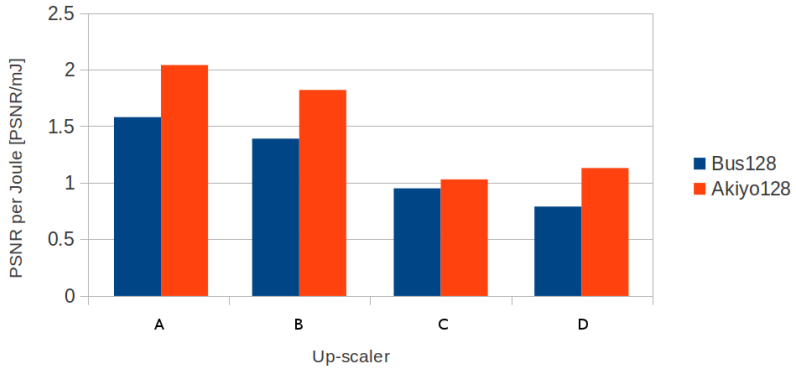


Figure 42: PSNR per Joule, for different up-scalers

Figure 43 shows the energy usage (y-axis) to process 25 frames for the different up-scalers (x-axis). For the different movies (bars), the energy usage differs when a different up-scaler is used. The energy usage for up-scaler A is similar for all the tested movies, the absolute difference (between the Simpsons64 and Bus128 movie) is within 12%. This is due to the code structure of the program, the program always executes the same code to generate a pixel. This is different for up-scaler D. Up-scaler D, only does bi-linear interpolation when the difference between the pixel values (to interpolate between) is bigger than 4 (adaptive up-scaler). When a movie has a large difference between neighboring pixels, it needs to perform the expensive bi-linear interpolation function more often. This explains the difference between the movies when up-scaler D is used. As expected the Bus movie uses the most energy because it contains the largest difference between pixels.

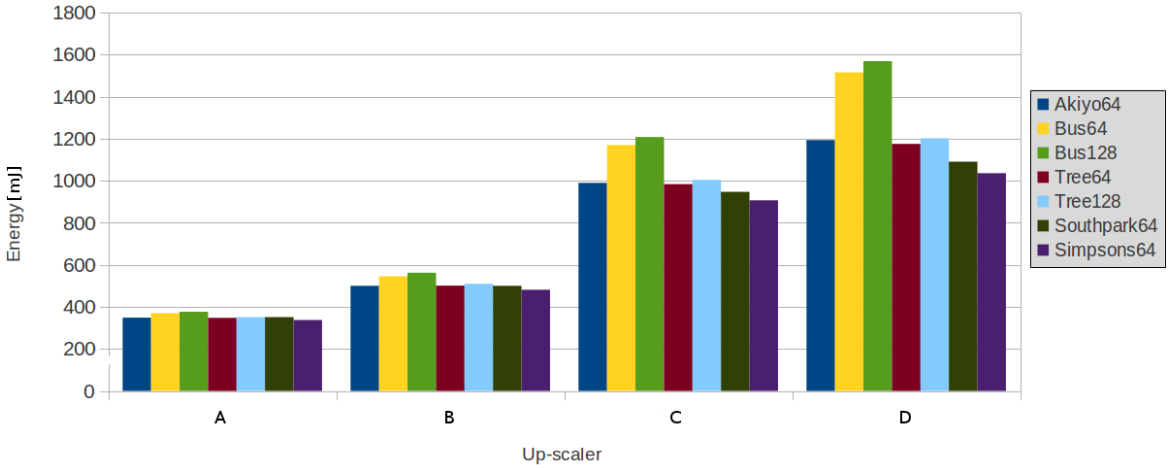


Figure 43: Energy results for different movies with different up-scalers

6.2.4 Configurations

In this section, the results of the previous experiments are combined to find pareto points (optimal trade-off points).

The energy needed to process 25 frames and the average overall quality differs per movie, see Figure 44. Each dot of the same type (and color) corresponds with a movie. Each dot of the same movie represents a configuration. To process one movie at the highest possible quality needs less energy than processing another movie at a lower quality mode. This graph presents justification for having a run-time quality manager because an application configuration cannot be linked with an energy usage level at design time.

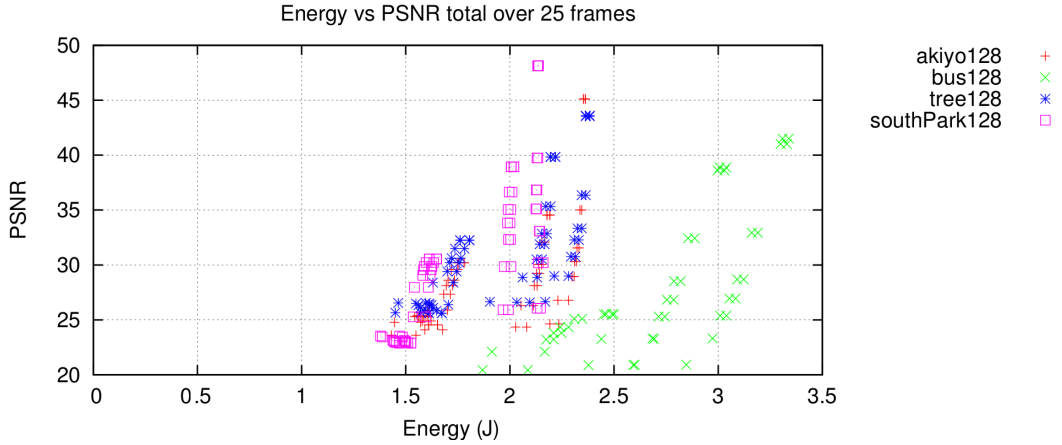
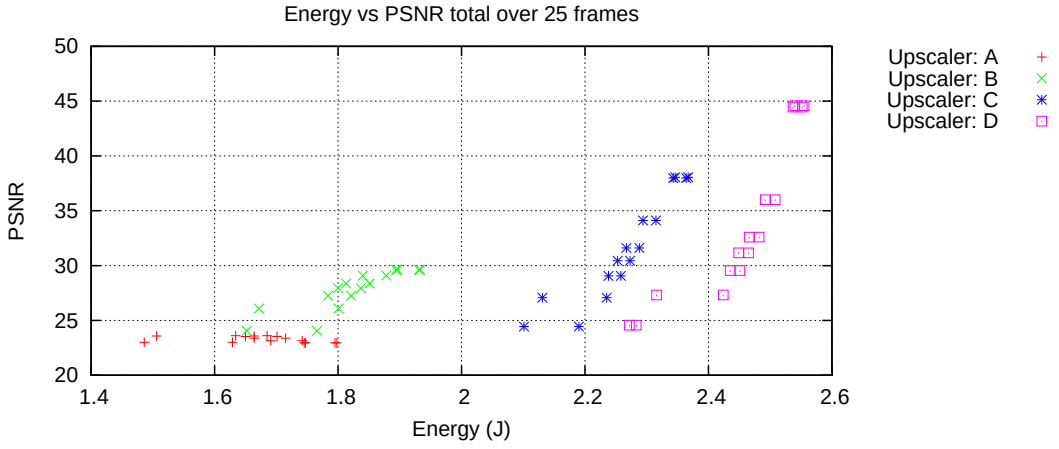
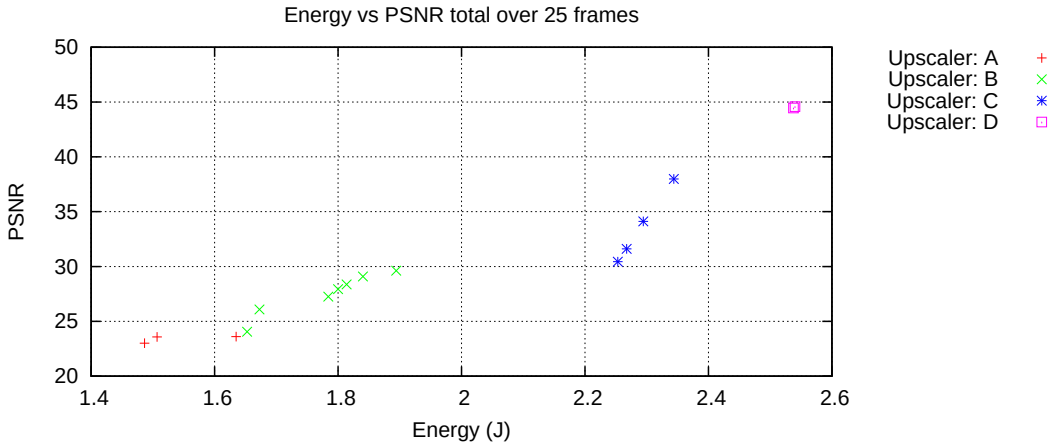


Figure 44: PSNR vs. energy, grouped per movie

When the average case between different frames and movies with the same configuration is plotted, it results in the trade-off shown in Figure 45(a). Each dot is a configuration, and grouped per up-scaler. Each of the four up-scalers are tightly grouped. The energy gain between the different up-scalers is higher compared to the gain of changing the number of AC values or the skipping of macro blocks.



(a) All points



(b) Pareto points

Figure 45: PSNR vs. energy, grouped per up-scaler

Selected points

When looking at Figure 45(a), then there are only a few configurations that are optimal, the other configurations give worse results in quality and/or energy.

For this application, only four configurations were selected from the pareto points in Figure 45(b). The application switches between these configuration points. The next experiments are performed on these configuration points. The resulting quality configurations are summarized in Table 5.

Table 5: Quality modes

Configuration name	Up-scaler	Nr. of AC values processed	Skipping blocks
Bad	A	6	Yes
Simple	B	15	Yes
Good	C	40	Yes
Best	D	63	No

6.3 Evaluate mappings

In this section, the influence of the data dependency between the cores, as described in Section 5.2.2 is evaluated. We also look into the consequence of having more variation on each core and how this influences each other.

In the previous experiments, the data dependencies resulting from the mapping was ignored. Data dependencies increase the needed energy if no free buffer spaces are available, or new input data is not available, explained in Section 5.2.2. We do not want to have different buffer configurations between the cores for the different movies and quality configurations. It is preferred to find one optimal buffering configuration that works in all situations. The penalty (in terms of energy) of the data dependencies is given as a % and is calculated as follows: $\frac{\text{Energy actual situation}}{\text{Energy optimal situation}} * 100 - 100$. The optimal situation in the calculation is taken from the previous Section (6.2.4).

Decreasing the frame rate could have a negative influence on the energy usage. It seems intuitive that decreasing the frame rate, reduces the frequency and energy because the available time to process a macro block/frame increases. But the contrary is true, When a lower frame rate is selected, more energy is spend on polling the FIFO's and waiting on data. Figure 46 shows the used energy to produce 24 frames at a different frame rate. It shows that at a lower frame rate, producing a frame takes more energy.

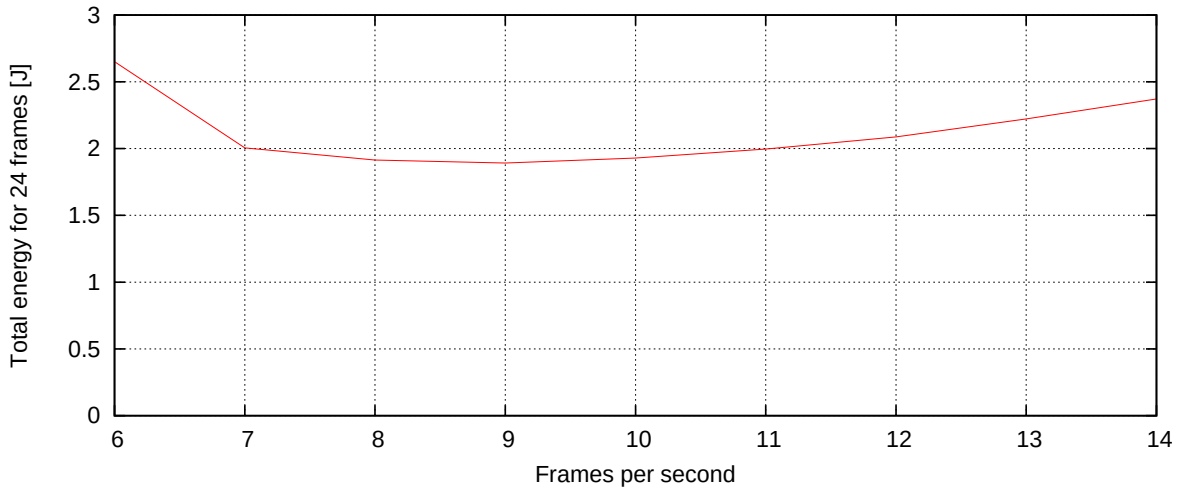


Figure 46: Usage energy to produce 24 frames, Tree128 movie at different frame rates

The experiments were performed by setting the frame rate at 10fps at 100MHz. When looking at Figure 46, this is not the best optimal point for this movie.

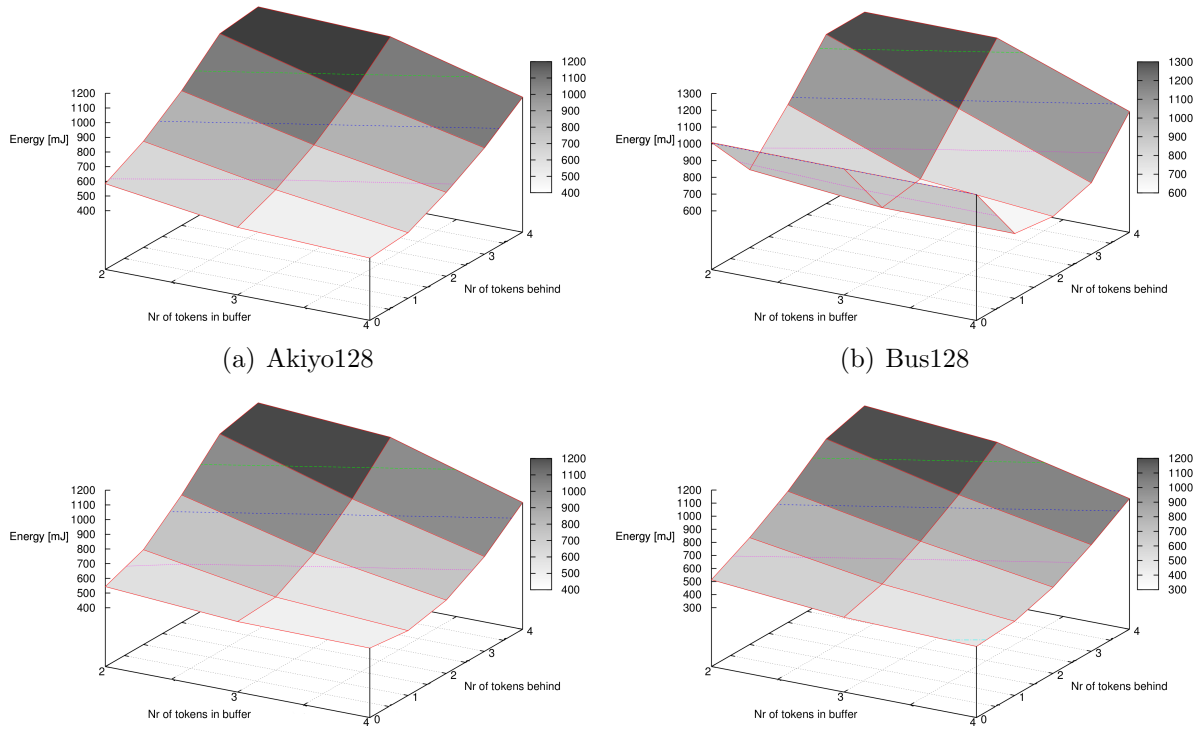
The following experiments in this section, are performed on the four application configuration points defined in Section 6.2.4. We assume similar behavior in data dependency penalties for the other configurations. This experiment is performed on all the test movies. The first 5 frames are ignored because some movies have a different behavior during startup due to the absence of the bit-rate mechanism.

The number of tokens that are buffered between the cores is indicated on the x-axis and the number of tokens that the second core runs behind on the y-axis. With the available platform the number of tokens that could be buffered was limited to 4, due to the limited communication memory. The total resulting energy usage for 5 frames is shown on the z-axis.

Application quality configuration: Best

This experiment is first performed on the configuration: up-scaler D, 63 AC values, no skipping blocks. This configuration has the least variation in execution time for the macro blocks on the first core, because no macro blocks are skipped. The influence of variation is explained in Section 5.2.2.

Figure 47(a), 47(c) and 47(d) shows the same trend, namely running behind is not needed (y-axis). For these movies, it holds that when the second core has finished producing a token, a new token is available. Both cores are in balance. This is different for Figure 47(b). This movie contains more motion and variation in the frames, resulting in more variation on the first core. Running second core behind is needed for this movie and running one token behind works.



(a) Akiyo128 (b) Bus128

(c) Tree128 (d) Southpark128

Figure 47: Used energy for 5 movie frames at different buffering configurations, application quality mode: Best

The energy penalty of having the data dependencies between the core are given in Table 6. The table indicates the additional % in energy that is needed, at different buffer configurations. The buffer configurations are represented as: {Nr of tokens in buffer (x-axis), Nr of tokens behind (y-axis)}. With the best found buffer configuration in terms of used energy, the energy penalty stays below 3%. For the worst tested buffering configuration ({2,4}), three times more energy is needed to play the same part of the movie.

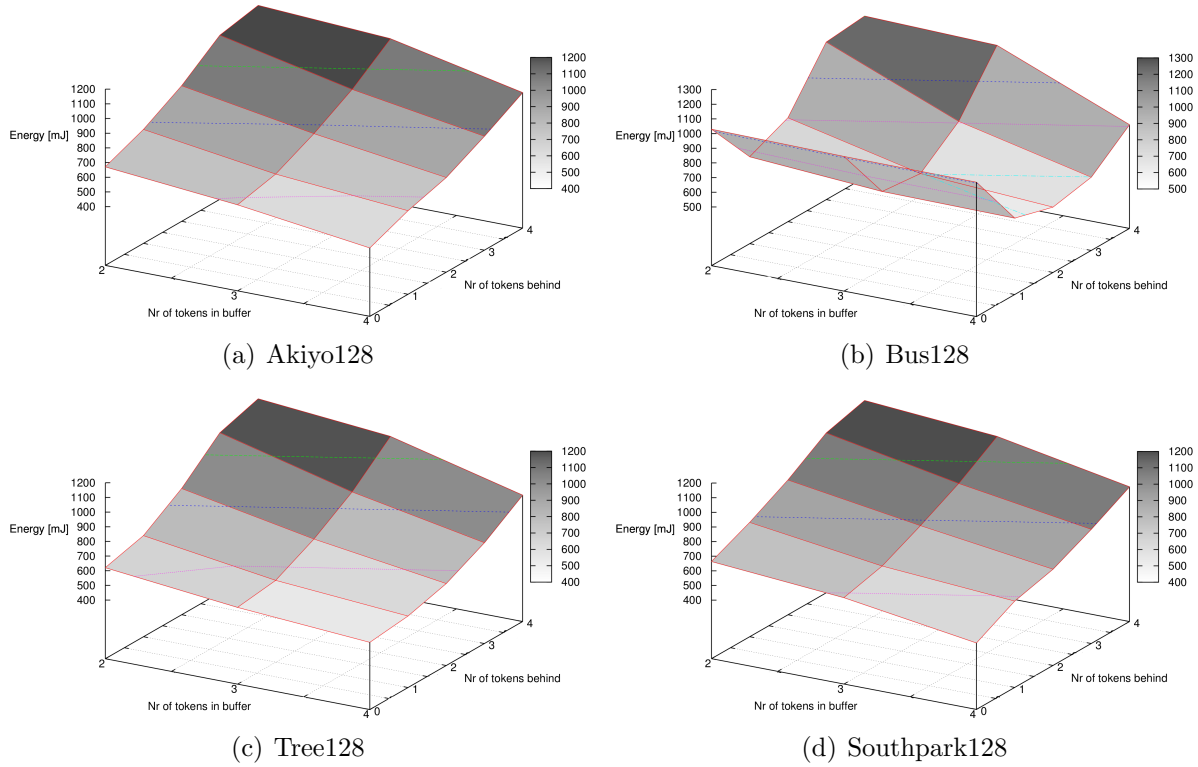
Table 6: Energy penalty for quality mode: Best

Movie	**Configuration:{4,1}**	**Best found configuration**
Akiyo128	7.6 %	2.23%, config{4,0}
Bus128	5.6 %	0.75%, config{4,2}
Tree128	1.24%	1.24%, config{4,1}
Southpark128	7.1 %	0.93%, config{4,0}
Average	5.39%	1.29%

Application quality configuration: Good

A similar behavior is expected for the found application quality configuration: Good. When a lower quality setting is selected, the amount of work on the second core decreases more compared to the first core. Since the up-scaler is less complex, but the rest is more or less the same, or at least scaling AC values and dropping blocks makes less of a difference. Over the different application quality configurations, a shift over the amount of tokens to run behind is expected.

Figure 48 shows the trade-off for the configuration: up-scaler C, 40 processed AC values, skipping macro blocks. It shows a similar pattern in terms of the used energy at different buffer configurations, as for the previous quality configuration. The only difference is the amount of tokens running behind for the Bus movie, namely running two tokens behind.



(a) Akiyo128 (b) Bus128

(c) Tree128 (d) Southpark128

Figure 48: Used energy for 5 movie frames at different buffering configurations, application quality mode: Good

The energy penalty for the data dependencies between the cores is given in Table 7. These results were surprising, because this mode used more energy and produced a lesser quality compared to the best mode. On average the energy usage increased with 3% compared to the used energy for the application quality configuration: Best.

Table 7: Energy penalty for quality mode: Good

Movie	**Configuration:{4,1}**	**Best found configuration**
Akiyo128	50.00%	22.76%, config{4,0}
Bus128	16.15%	2.15%, config{4,2}
Tree128	23.44%	15.97%, config{4,0}
Southpark128	60.87%	23.78%, config{4,0}
Average	37.62%	16.17%

This result shows the problem explained in Section 5.2.2. When the application skips a macro block, it increases the variation in execution time on the first core and so increases the number of stall cycles on the first core, due to full buffers. This hypothesis explains that the "Bus" movie has a limited energy penalty due to the dependency, because less macro blocks where skipped in this movie.

This issue can be resolved by buffering more tokens. Unfortunately on the test platform, there is not enough communication memory available to increase the buffer size. When the same experiment is repeated, but without skipping macro blocks, it results in the results shown in Table 8.

Table 8: Energy penalty for quality mode: Good, no skipping of macro blocks

Movie	**Configuration:{4,1}**	**Best found configuration**
Akiyo128	4.48%	4.48%, config{4,1}
Bus128	17.84%	1.88%, config{4,2}
Tree128	2.25%	2.25%, config{4,1}
Southpark128	2.73%	1.83%, config{4,0}
Average	6.83%	2.61%

When skipping of macro blocks in this application quality configuration is disabled, it uses less energy than the Best application quality configuration, namely an average decrease in energy of 8.65% for the same movies between the Best and the Good application quality configuration. Similar trends and results are observed for the other application quality modes. Skipping macro blocks introduces more variation in execution time, resulting in an increases of energy usage.

6.3.1 Reevaluated quality modes

According to the previous results in this section (Table 7 and 8), the application quality modes specified in Section 6.2.4 are not the optimal points when the mapping is taken into account. Skipping macro blocks with this mapping is therefore not beneficial option in terms of energy consumption. The settings that can be changed on the first core is now limited to only the number of AC values to process. The reevaluated modes are given in Table 9.

Table 9: Revisited quality modes

Configuration name	Up-scaler	Nr of AC values processed	Skipping blocks
Bad	A	6	No
Simple	B	15	No
Good	C	40	No
Best	D	63	No

The same experiment is performed again now without skipping macro blocks. Table 10 shows the energy penalty of having the data dependencies for the newly defined quality modes. The penalties are limited when the best found buffering configuration is used. By having a suitable buffering configuration, the data dependencies can be minimized.

Table 10: Summary energy penalty to buffering, at different quality modes

Movie	Best	Good	Simple	Bad
Akiyo128	2.23%	3.94%	2.55%	3.06%
Bus128	0.75%	0.54%	2.36%	1.68%
Tree128	1.24%	0.68%	0.96%	1.35%
Southpark128	0.93%	2.59%	1.98%	2.89%

Conclusions

As long as the variation in execution time for a macro block on each core is limited, the energy penalty through blocking actions of having the data dependencies can be minimized. For this reason, skipping of macro blocks increases the energy usage on waiting. The variation in execution time could not be caught by the 4 place FIFO buffer between the cores with the used mapping. Skipping macro blocks is an option for a single core system because when the data dependency is not taken into account (see Section 6.2.4) energy is saved with this mechanism.

When finding an optimal application quality configuration (pareto point), the dependencies between tasks, and buffers between the tasks need to be taken into account. Otherwise, the wrong configurations are found in terms of energy consumption.

6.4 Evaluate quality manager

Each mechanism related to frequency scaling and quality scaling is evaluated in the previous experiments, resulting in the four configurations. In this section, the control loop is evaluated earlier defined in Section 5.4.

The policy is evaluated for one video decoder that displays a full screen output. In total, an energy budget for 1200 frames is given before the decoder starts. If a test movie does not have enough frames, it is repeated. The resulting quality is given as a number from 0 to 4. 0= no quality, 1=Bad, 2=Simple, 3=Good, 4=Best, related to the found configurations (Section 6.3.1). When the decoder runs out of energy before all the frames are produced, it results in quality 0. The overall quality is calculated as follows: $\frac{\sum_{i=1}^{1200} Qualitylevelframe_i}{1200}$. Each movie runs at 10fps at 100MHz and the application gets 50% of the application slots. The energy budget is plotted on the x-axis, it is given in mJ per frame $\frac{EnergyBudgetTotal}{1200}$.

The resulting overall quality for a given energy budget for the different movies is shown in Figure 49. All the movies shows the same pattern, except the "bus" movie. This movie has that the first frames use more energy than the rest of the frames. The movie has in total 150 frames, it is looped 8 times, resulting in that the part in the beginning is processed 8 times.

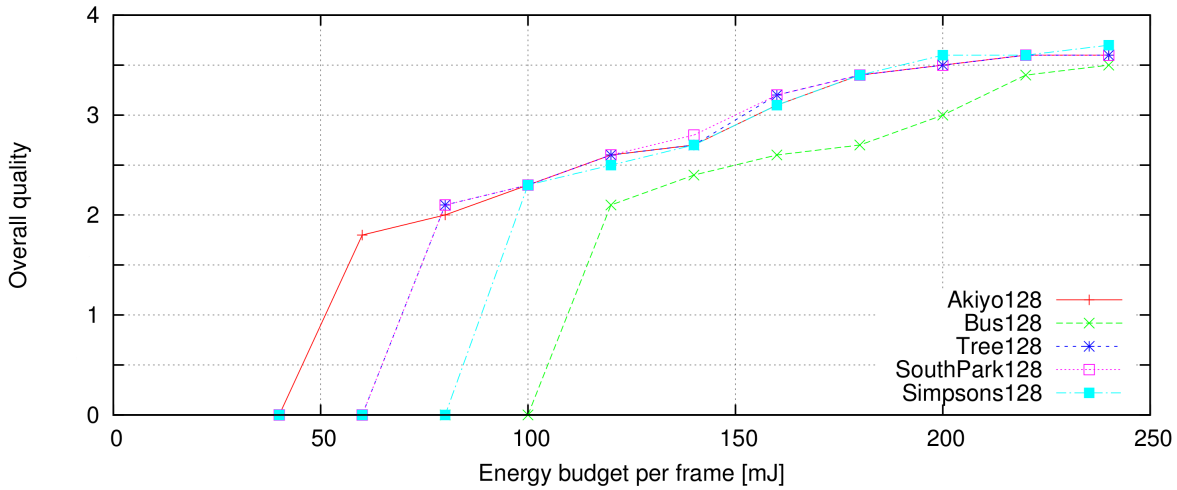


Figure 49: Overall quality depending on the energy budget

The conservative behavior is visible in Figure 49. It is hard to get a high overall quality. When the budget is set to more than $200mJ$ per frame, the application has energy budget left over after the 1200 frames that could be spent on a higher movie quality. A solution for larger energy budgets (200mJ $\leq$ energy budget per frame), is to start the application with energy slack, so that the application can use the slack in order to run the first frames in a higher quality configuration.

6.5 Evaluate composability

Composability is taken into account at system design time, see Chapter 4. This section verifies experimentally that the system is composable. To prove (energy) composability, applications cannot interfere with each others timings. First, the verification is performed on platform B (defined in Section 4.2.1), with DDR memory.

The PiP application described in Section 3.1.2 is used, each application gets 50% of the application slots in the application TDM schedule, see Section 4.3.2. The energy usage is measured only for the application under test.

Situation 1: both applications processes a movie. This situation is executed twice, to check if there is a difference between the two runs.
Situation 2: application under test processes a movie, the other application idles. This situation is executed twice.

The application under test, processes the movie: "Tree" with the best quality mode, the other application processes the "simpsons" movie as a picture in picture. Both applications process the movies at 10fps on cores running at 100MHz.

Figure 50 shows the used energy per frame for the different situations and runs. It shows that there is a difference (for example frame 23) between the different situations and runs, but it is limited.

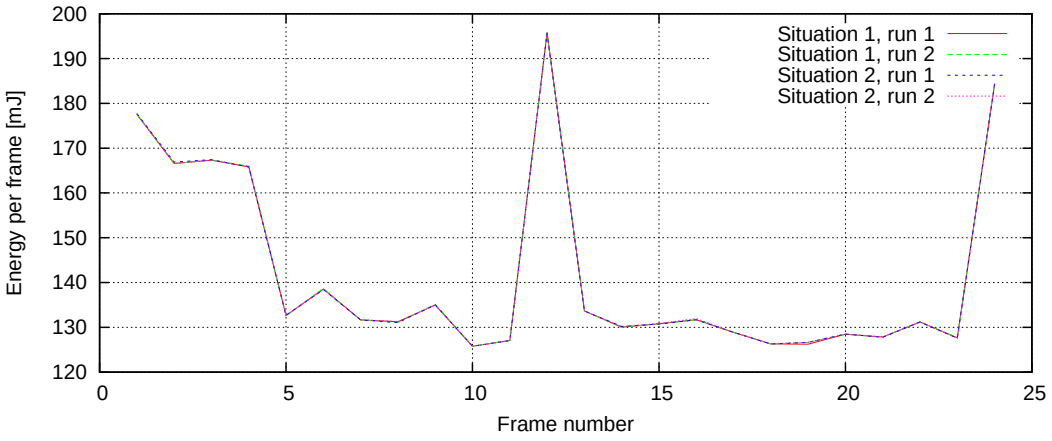


Figure 50: Energy usage per frame

Variation between runs in the same situation

First, the difference of the energy between the different runs in the same situation is measured. The difference in % is plotted in Figure 51. The red line shows the difference in situation 1 and the green line shows the difference in situation 2. There is difference between the runs in the same situation, but stays within 0.5%. This variation in energy is due to the DDR controller. The problem (in difference in energy) could be the result that it takes time to open/close row in memory banks and to switch from reads to writes and vice versa [2]. Applications can therefore

interfere temporally. The difference between the runs could also be as a result of the clock domain crossing to DDR, or that the interaction between the memory controller and the physical memory device is non-deterministic.

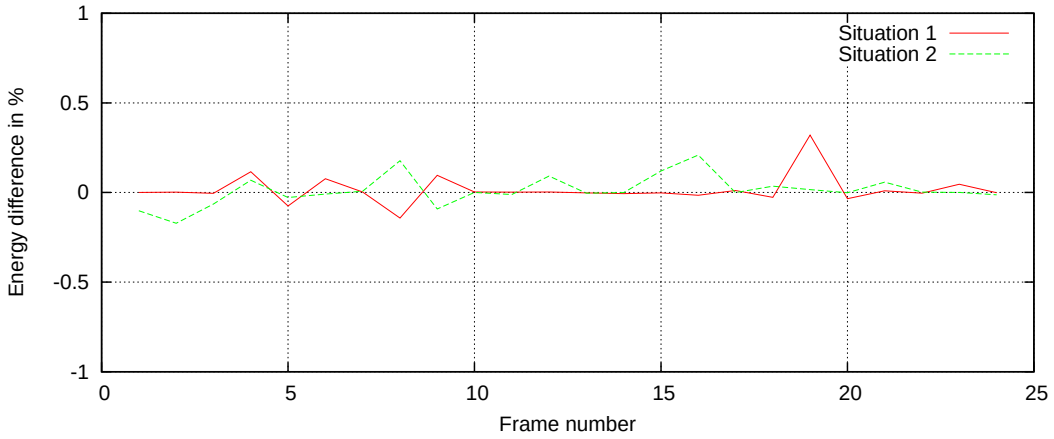


Figure 51: Energy difference between two runs

Variation between situations

The energy difference between the different situations is measured next. The result is shown in Figure 52. The difference in energy between runs within the same situation, stays within 0.5%. This result also confirms that the CompSoC platform we use as the running example is not composable. The DDR could be the reason for this energy difference.

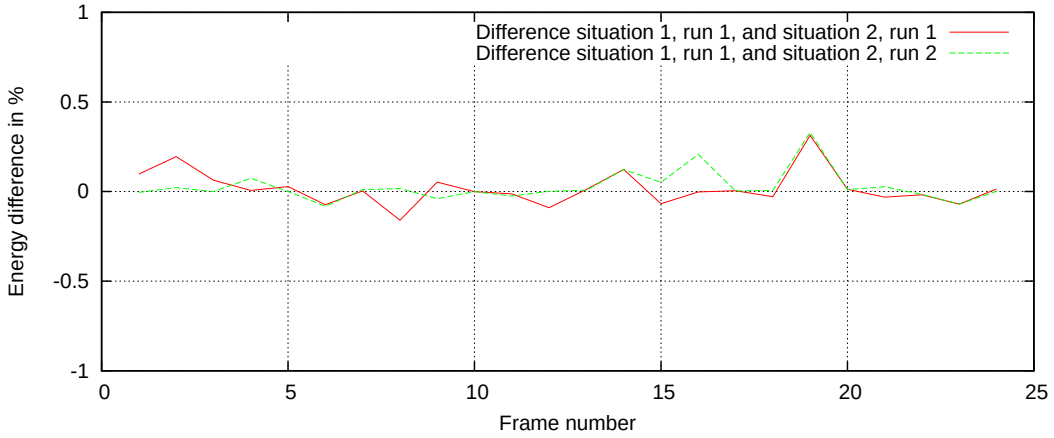


Figure 52: Energy difference between different situations

Without DDR

The composability experiment is repeated, with the movie in local memory. The experiment is performed on platform A that does not contain DDR memory. The rest of the experimentation remains the same, i.e. the same two situations are evaluated and each situation is executed twice.

The four runs are plotted in Figure 53. Each run used the same amount of energy. The runs on platform A uses less energy than when the DDR is enabled. This is due to the shorter memory latency.

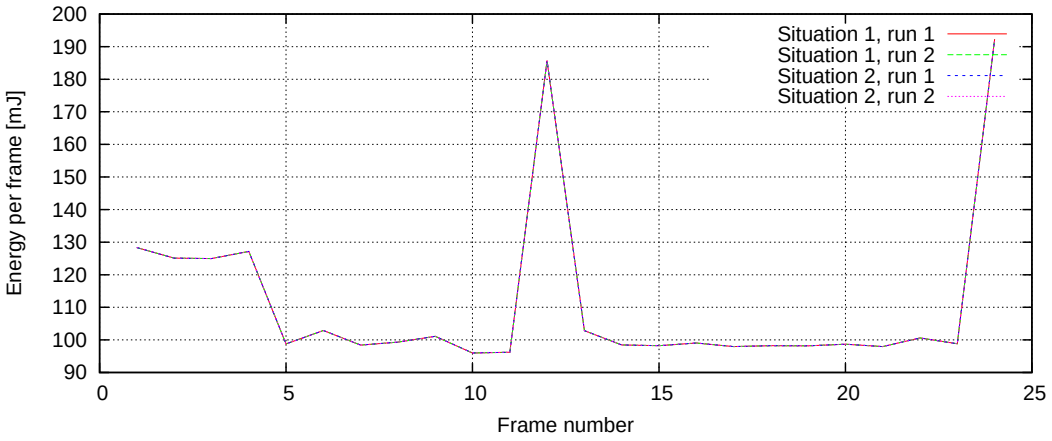


Figure 53: Energy usage per frame, no DDR

The difference in energy between the different situations is shown in Figure 54. There is no energy difference between the different runs and different situations. This shows that the applications do not interfere with each other in terms of energy and that the experimental platform is composable.

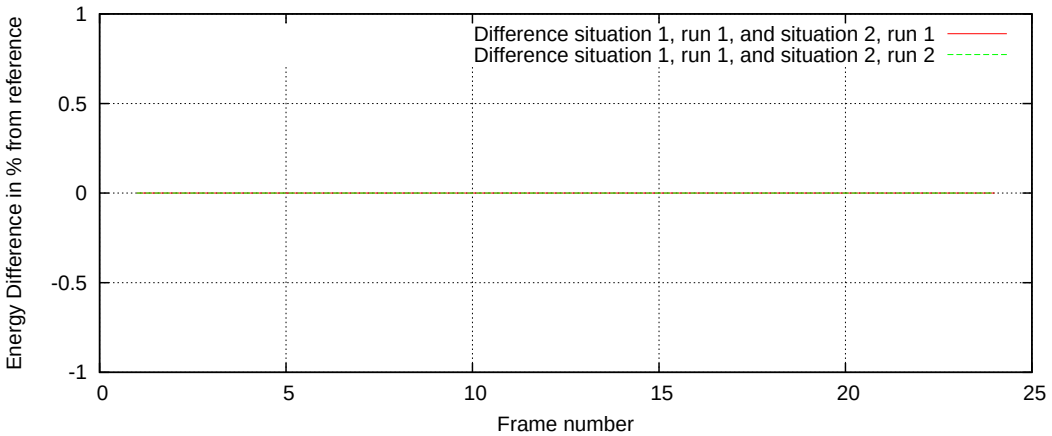


Figure 54: Energy difference between different situations

7 Conclusions

The conclusions are grouped into the three problem statements.

Adaptive H.263 decoder

In this thesis we show that applications can be made adaptive. We have shown how this may be achieved for the H.263 decoder, but these methods could be applied to other applications. The adaptively in the example application has an impact on the application output quality and execution time. We showed for the H.263 decoder that an application configuration can be coupled with a quality level. The consequence of changing the quality level needs to be known at run-time and the influence it has on the worst case execution time (WCET) and the best case execution time (BCET).

We demonstrated applications that are mapped over multiple cores running on an MPSoC platform. Having distributed applications influences the effectiveness of the scalable functions of the application on the energy consumption. Mapping needs to be taken into account when the used combinations of the scalable application functions are chosen.

Quality modes need to be selected carefully. A mode that performed well in the optimal situation can perform worst when it is run on the actual system. Variation in execution time is used to scale energy. But too much variation has a negative consequence on the energy consumption.

Quality manager and policy

We have introduced a quality manager related to the quality of the application output and energy usage. This gives the user the control to make the trade-off for quality versus energy. The CompOSe RTOS has budgets per application that is used by the quality manager. The quality manager consists of two parts, namely the DVFS part with variable thresholds and the quality function. The quality function controls which quality mode to run in and the thresholds for the frequency scaler function.

Multiple independent adaptive applications

Composability simplifies the integration and verification of multiple applications executing on the same hardware resource.

We have shown that applications do not interfere and that applications can independently manage their quality. If hardware is shared that is not composably arbitrated, variation in execution time is introduced within the execution in the application.

8 Future work

General method for data decoupling

We mapped the application over two cores. Additional energy is used on synchronization and waiting on buffer spaces. This is due to the data dependency between the tasks that are mapped on different cores. We could limit the waiting time on data and free buffer spaces of the mapping, but when an application is mapped on more cores, or more tasks are dependent on each other, additional energy is spent blocking on the dependencies between the cores.

At design time the buffering need to be dimensioned appropriately to enable energy/power reduction. A general solution to decouple the data is preferred to simplify the development and verification of applications.

Run-time dynamism to statically allocated budgets

The current budgeting methods that are implemented in CompOSe are per application. Application budgets are divided per core at design time. With adaptive applications the dividing of the budget over the cores is not known at design time. Adding run-time dynamism to statically allocated budgets within the applications, but on multiple cores, resolves the budget imbalance between the cores.

References

[1] Benny Akesson. *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, February 2010. ISBN: 978-90-386-2169-2.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256. ACM Press New York, NY, USA, September 2007.

[3] R.J. Bril, C. Hentschel, E.F.M. Steffens, M. Gabrani, G. van Loo, and J.H.A. Gelissen. Multimedia qos in consumer terminals. In *Signal Processing Systems, 2001 IEEE Workshop on*, pages 332 –343, 2001.

[4] G. Cote, B. Erol, M. Gallant, and F. Kossentini. H.263+: video coding at low bit rates. *Circuits and Systems for Video Technology, IEEE Transactions on*, 8(7):849 –866, nov 1998.

[5] Bernd Girod. Digital images and human vision. chapter What's wrong with mean-squared error?, pages 207–220. MIT Press, Cambridge, MA, USA, 1993.

[6] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414 – 421, sept.-oct. 2005.

[7] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 306 –311, june 2010.

[8] K. Goossens, D. She, A. Milutinovic, and A. Molnos. Composable dynamic voltage and frequency scaling and power management for dataflow applications. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 107 –114, sept. 2010.

[9] Andreas Hansson, Marcus Ekerhult, Anca Molnos, Aleksandar Milutinovic, Andrew Nelson, Jude Ambrose, and Kees Goossens. Design and implementation of an operating system for composable processor sharing. *Microprocessors and Microsystems*, 35(2):246 – 260, 2011. Special issue on Network-on-Chip Architectures and Design Methodologies.

[10] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14:2:1–2:24, January 2009.

[11] Christian Hentschel. Scalable video algorithms for resource constrained platforms. *CMedia Technology, Brandenburg University of Technology Cottbus.*

[12] Q. Huynh-Thu and M. Ghanbari. Scope of validity of psnr in image/video quality assessment. *Electronics Letters*, 44(13):800 –801, 19 2008.

[13] ITU-T. H.263 video coding for low bit rate communication, 2005.

[14] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112 – 126, jan 2003.

[15] A. Molnos and K. Goossens. Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 409 –418, aug. 2009.

[16] A. Nelson, A. Molnos, and K. Goossens. Composable power management with energy and power budgets per application. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 396 –403, july 2011.

[17] M.H. Pinson and S. Wolf. A new standardized method for objectively measuring video quality. *Broadcasting, IEEE Transactions on*, 50(3):312 – 322, sept. 2004.

[18] Parag Sharma Ripal Nathuji, Paul England and Abhishek Singh. In *Feedback Driven QoS-Aware Power Budgeting for Virtualized Servers*, 2009.

[19] Zhou Wang and A.C. Bovik. A universal image quality index. *Signal Processing Letters, IEEE*, 9(3):81 –84, mar 2002.

[20] Zhou Wang and A.C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *Signal Processing Magazine, IEEE*, 26(1):98 –117, jan. 2009.

[21] Zhou Wang, Alan C. Bovik, and Ligang Lu. Why is image quality assessment so difficult? In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 4, pages IV–3313 –IV–3316, may 2002.

[22] S. Winkler and P. Mohandas. The evolution of video quality measurement: From psnr to hybrid metrics. *Broadcasting, IEEE Transactions on*, 54(3):660 –668, sept. 2008.

[23] Stefan Winkler. *Vision models and metrics*. 2005.

[24] Clemens C. Wst, Liesbeth Steffens, Wim F. J. Verhaegh, Reinder J. Bril, and Christian Hentschel. Qos control strategies for high-quality video processing. *Real-Time Systems*, 30:7–29, 2005. 10.1007/s11241-005-0502-1.

[25] Xilinx. Microblaze soft processor core, 2010.

[26] Xilinx. Virtex-6 fpga ml605 evaluation kit, 2010.

[27] Zhenghua Yu and H.R. Wu. Human visual system based objective digital video quality metrics. In *Signal Processing Proceedings, 2000. WCCC-ICSP 2000. 5th International Conference on*, volume 2, pages 1088 –1095 vol.2, 2000.