Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

Master's Thesis in Embedded and Intelligent Systems

Süleyman Savaş
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

February 2011
The figure on the cover page is the overview of the decoder and shows the blocks which build the decoder.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

Acknowledgement

I want to thank Professor Bertil Svensson for providing the opportunity of working on this thesis. Furthermore, I want to thank John Bartholomew, a senior software architect at Nethra Imaging and the technical lead of the software tools team, for his help about the ambric architecture and performance analysis of the ambric chip. To Carl Von Platen, a senior researcher at Ericsson, for his help about the CAL actor language and for the performance results of the CAL implementation and to Zain-Ul-Abdin for his support and help about the thesis. Additionally I want to give my special thanks to my family in Turkey and to my fiancé, who helped me with the language of the report. As last, thanks to Husni Khanfar who provided the parser block of the decoder.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

Abstract

The high demand of the video decoding has pushed the developers to implement the decoders on parallel architectures. This thesis provides the deliberations about the implementation of an MPEG-4 decoder on a massively parallel processor array (MPPA), Ambric 2045, by converting the CAL actor language implementation of the decoder. This decoder is the Xilinx model of the MPEG-4 Simple Profile decoder and consists of four main blocks; parser, acdc, idct2d and motion. The parser block is developed in another thesis work [20] and the rest of the decoder, which consists of the other three blocks, is implemented in this thesis work. Afterwards, in order to complete the decoder, the parser block is combined with the other three blocks.

Several methods are developed for conversion purposes. Additionally, a number of other methods are developed in order to overcome the constraints of the ambric architecture such as no division support. At the beginning, for debugging purposes, the decoder is implemented on a simulator which is designed for Ambric architecture. Finally the implementation is uploaded to the Ambric 2045 chip and tested with different input streams. The performance of the implementation is analyzed and satisfying results are achieved when compared to the standards which are in use in the market. These performance results can be considered as satisfying for any real-time application as well. Furthermore, the results are compared with the results of the CAL implementation, running on a single 2GHz i7 intel processor, in terms of speed and efficiency. The Ambric implementation runs 4.7 times faster than the CAL implementation when a small input stream (300 frames with resolution of 176x144) is used. However, when a large input stream (384 frames with resolution of 720x480) is used, the Ambric implementation shows a performance which is approximately 32 times better than the CAL implementation, in terms of decoding speed and throughput. The performance may increase further together with the size of the input stream up to some point.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array
Contents

Acknowledgement........................................................................................................3
Abstract..........................................................................................................................5
Contents..........................................................................................................................7

1 Introduction................................................................................................................11
  1.1 Motivation..............................................................................................................11
  1.2 Goals.....................................................................................................................11
  1.3 Problem Statement & Approach............................................................................11
  1.4 Contribution..........................................................................................................12
  1.5 Structure ..............................................................................................................12

2 The Architecture of Ambric Processor Array Family ..............................................14
  2.1 Introduction..........................................................................................................14
  2.2 Ambric Compute Unit, RAM Unit, Brics............................................................14
  2.3 Ambric Registers and Channels............................................................................15
    2.3.1 Registers.......................................................................................................15
    2.3.2 Channels.......................................................................................................16
    2.3.3 Hardware Objects & Clocking.....................................................................16
    2.3.4 Data Extent...................................................................................................17
      2.3.4.1 TMS Packets.........................................................................................18
  2.4 Processor Architecture..........................................................................................18
    2.4.1 SR and SRD Processors...............................................................................18
  2.5 Memory Architecture............................................................................................20
  2.6 Programming Model.............................................................................................21
    2.6.1 Structural Object Programming Model......................................................22
    2.6.2 Programming and Analysis Tools & Languages..........................................23

3 Video Encoding/Decoding & MPEG........................................................................30
  3.1 Video Introduction..............................................................................................30
  3.2 Color Spaces.......................................................................................................30
  3.3 Video Compression..............................................................................................31
    3.3.1 Redundancy Reduce....................................................................................32
      Spatial and Temporal Redundancy:...................................................................32
      Frequency Domain Redundancy.......................................................................32
    3.3.2 Motion Estimation.......................................................................................33
    3.3.3 Motion Vectors............................................................................................34
    3.3.5 Discrete Cosine Transform (DCT)...............................................................34
    3.3.6 Quantization.................................................................................................35
    3.3.7 Zig-Zag Scan...............................................................................................35
    3.3.8 Entropy Coding............................................................................................36
  3.2 MPEG..................................................................................................................36
    3.2.1 Overview......................................................................................................36
    3.2.2 MPEG-1.......................................................................................................37
    3.2.3 MPEG-2.......................................................................................................37
    3.2.4 MPEG-4.......................................................................................................37

4 Reconfigurable Video Coding (RVC) & CAL Actor Language...............................39
  4.1 RVC Framework Introduction..............................................................................39
  4.2 ISO-MPEG Standardization................................................................................39
7 Performance Analysis & Comparison ....................................................... 70
7.1 Performance Results ........................................................................... 70
7.2 Comparison ......................................................................................... 71
    7.2.1 Ambric vs Intel Processor using the CAL implementation ................. 72
    7.2.2 Comparison of the Results for Two Different Input Streams ............... 73
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

8 Conclusions........................................................................................................................................... 75
8.1 Similarities between CAL and Ambric Languages ............................................................................ 75
8.2 Differences between CAL and Ambric Languages ............................................................................ 75
8.3 Performance ...................................................................................................................................... 76
8.4 Development Process & Tools ........................................................................................................... 76
8.5 Future Work ........................................................................................................................................ 77

9 References.............................................................................................................................................. 79
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array
1 Introduction

1.1 Motivation

The video is used widely in the human life for many purposes such as entertainment, communication, gathering intelligence, security, safety, control mechanisms. In some cases the images/video does not need to have a high quality however there are certain cases requiring high quality images or video stream. Therefore, large efforts were spent to create better quality images by increasing the resolution, size and color depth. Even the objects in the pictures are extracted in order to provide the user the opportunity of manipulating these objects.

Development of the pictures along with the development of high definition video streams, pushed the MPEG standards to further developments. The demand on the video decoders increased rapidly as well. Therefore the capabilities of the decoders are forces to increase. Consequently it becomes more difficult to meet all of the expectations with the conventional programming languages and techniques. Furthermore, the power of single processors becomes not sufficient for certain applications. Hence, multicore architectures and dataflow languages are started to be in use in order to meet the high performance demands. The structure of the decoder is suitable to be implemented on multicore architectures. The decoder can be divided into different blocks which perform different operations. These operations are possible to parallelize as well. Thus, the blocks can be divided further into smaller parts and each part can be mapped on a processor. At the end, the decoder can be build by using many interconnected cores which run in parallel.

1.2 Goals

Ambric 2045 chip is a massively parallel processor array consisting of 336 processors. The first goal is to implement an MPEG-4 simple profile decoder, which is already written by an actor language called CAL [16, 17, 18], on this chip. Different input streams are planned to be used in order to test the decoder on the ambric chip.

Performance of the decoder will be analyzed by measuring the execution times. Furthermore, the ambric chip will be compared with a general purpose (intel i7) processor using the CAL implementation of the decoder, by using the decoding times of two different input streams. The comparison will give an idea about the efficiency of the architecture for video decoding.

1.3 Problem Statement & Approach

Video decoding became more complex due to the increase of the image quality and several other factors. Single processors in the embedded systems are not powerful enough to meet the expectations of real-time applications. Thus multicore architectures became the focus for video decoding in the embedded systems. Since the latest standard of video coding is MPEG-4, it is used in new decoder implementations. In addition to decoding, general purpose computations are needed as well in the embedded systems. Using multicore architectures meet this expectation as well by using a number of the cores for general purposes.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

The MPEG-4 decoder is written in CAL actor language which is different than the programming language used for ambric architecture which is a java. Since the CAL is a dataflow language, its capabilities are larger than the capabilities of ajava and the architecture as well. When compared to CAL language, Ambric architecture has several constraints.

Due to being a parallel implementation, the CAL implementation is chosen in order not to spend time for parallelizing a sequential implementation.

While developing a decoder for the ambric architecture, the constraints must be taken into account. Especially, while converting from a dataflow language, one may run into problems due to the limitations.

Before starting the implementation, there are questions need to be answered:

“How to implement the schedules defined in the CAL language?”

"Is it possible to fit a CAL actor to a leaf object?"

“How to divide a CAL actor which does not fit into a leaf object?"

“How to check the input availability?”

“How to overcome the memory limitation?”

All questions will be answered in further chapters.

1.4 Contribution

In this thesis work, 3 of 4 blocks of the MPEG-4 simple profile decoder are converted from CAL language to the ambric languages. Several methods are developed in order to convert the CAL implementation. CAL actors are converted as leaf objects and only one actor did not fit into a leaf object (SRD processor). The solution is to divide the actor into two leaf objects.

In general, actions of the CAL language are converted as methods. Moreover the default method (run) of the ajava language is used as a scheduler to implement the schedule and the priorities in the CAL language. However, the simple actors are converted without using any scheduling. The actions are considered as sequential processes.

In order to check for availability or the value of an input (without consuming) a simple method is developed. Since the division operation is not supported, a division method is developed as well. In the CAL language sending boolean values through channels is supported, instead of using boolean values, 1 and 0 are used meaning true and false respectively. External DDR memory is used when an actor requires a large memory space. Additionally, a few of the actors have very complex scheduling and action priorities. Instead of dealing with a scheduler, the execution sequence of the actions can be found and arranged in an easier way by using if conditions and for loops.

1.5 Structure

This thesis report consists of seven main chapters. In chapter 2, the architecture of ambric processor array family, the programming model and the development & analysis tools are
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

described. In chapter 3, the fundamentals of video encoding and decoding are described as well as the main MPEG standards. Chapter 4 describes the CAL actor language with the reasons of its usage. Additionally reconfigurable video coding (RVC) is explained. In chapter 5, the methods developed for conversion purposes are described and answers are given for the problems described in the Problems chapter. Chapter 6 gives detailed information about the implementations of the actors, the occurred problems and the solutions. In chapter 7, performance results of the ambric chip are given. The chip is compared with a notebook using single processor and CAL implementation of the decoder. In chapter 8, the final conclusions about the conversion of the CAL language, performance and efficiency of the ambric architecture are given. Several future works, which aim to enhance the decoder and use the ambric chip more efficient, are proposed.
2 The Architecture of Ambric Processor Array Family

2.1 Introduction

Interconnecting hundreds of processors and memories with flexibility, high performance, low cost and low power has always been a challenge for the architectures with multi processors. Moreover, they have challenged with programming and validating a complex and irregular application on those processors efficiently, which is the main issue. With conventional multiprocessing techniques, keeping the processors busy, communicating with one another and synchronizing is not as easy and can be failure-prone.

Ambric chip (Am2045) has around 180 million silicon transistors and 336 32-bit processors. The memory is distributed to the processors with constant amount. The main issue of Ambric chip is Ambric register that is used through channels. The chip is asynchronous inside but the processors or the objects synchronize and communicate with one another through these configurable Ambric channels. In structural object programming, objects are strictly encapsulated software programs. They run concurrently on asynchronous array of processors and memories.

There are a few structures on Ambric chip such as compute unit (CU), ram unit (RU) and brics.

2.2 Ambric Compute Unit, RAM Unit, Brics

Compute unit consists of 2 SRD and 2 SR processors. Each compute unit has a ram unit as a pair which consists of 4 RAM banks. Each of the rams has 2KB of memory. The RAM banks are reached through the RU engines. There is a configurable network between RU engines and RAM banks. Thus the RU engines connect to the RAM banks dynamically through the configurable network. There are channels also between compute units. The interconnection between two CPUs in CU is dynamic under instruction control. When 2 compute units and 2 ram units come together, they create a bric. In total it makes 8 CPUs and 21KB of RAM. Brics have their own circuit-switched interconnection network with configurable switches. This network always runs at the highest clock rate and connects compute units through clock crossing registers. In total there are 45 brics on the AM2000 family Ambric chip.
2.3 Ambric Registers and Channels

This section deals with Ambric channels and registers which are the core of the Ambric chip.

2.3.1 Registers

Architecture of the massively parallel processor array (MPPA) is built on the Ambric registers. The capabilities of the architecture and the implementations depend on these registers. Instead of ordinary registers, there are only ambric registers throughout the Ambric board. They are rather different than the ordinary edge triggered registers however still there are similarities. Ambric registers are also clocked registers and they have data in and data out. In addition to data in and data out there are accept and valid signals as control signals to implement a hardware protocol for local forward and backward pressure. This protocol makes the system self-synchronized and asynchronous.

The operations in Ambric registers are self initiated. When the register is ready to get an input it asserts it’s accept signal upstream. On the other hand when the register has an output to send, it asserts its valid signal downstream. Thus, when two registers, which are connected to each other, sense that both signals are true, they realize that the transfer of the data can be done or the transfer of the previous data has already been done. As the result the registers become self-synchronized by using two signals. Since these dynamics are self contained, the synchronization becomes hidden.

Each register can hold two words. This happens in case of output stalls and requirement of incoming data. The register holds both the stalled output and the incoming data. Thus in a register chain with N registers, Nx2 data words can be held.
2.3.2 Channels

Ambric channels consist of Ambric registers. Channel is created by putting together few Ambric registers. The channels are scalable and the length depends on the number of registers in the channel. The channels are fully encapsulated and used for data and control token transfers between objects. They are ordered as first in first out and connected point to point. One other property of these channels which can be called as a drawback is being unidirectional. During implementations this property will be mentioned more.

Since each register has capacity of holding two data words, the channels can be used as small FIFOs as mentioned in previous section. Bigger FIFOs can be implemented by using RAMs. These FIFOs can be used in channels transparently, to control the data flow or implement storage in the application when it is necessary. FIFOs often replace random access memories due to being self synchronized.

Ambric channels are used throughout the chip, inside the processors, memories and interconnections.

2.3.3 Hardware Objects & Clocking

A hardware object in Ambric can be combinational logic, state machine or RAM. An object is interconnected by Ambric registers. Internal synchronization is acquired by using accept and valid signals. Composite objects and complete applications are obtained by connecting objects. These objects are connected by Ambric channels, by sending the data; the synchronization between objects is acquired. Since the objects run independently and they are synchronized only through the channel interfaces, this system is globally asynchronous and locally synchronous (GALS).

The synchronization protocol enables clock crossing registers. A clock crossing register works with different, but related, clocks on input and output. Thus, the objects that are connected to the ends of this register can run with their own speed. Objects can adjust their clock rates
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

dynamically during an application. For instance when an object feels that it is stalling then it reduces the clock rate in order not to consume much power. So each object runs at the lowest clock rate possible.

The processors are also hardware objects. They are interconnected by Ambric channels and called leaf objects in programming model. They synchronize with each other by using transfer events. Since they are connected with Ambric registers they communicate as registers. If a processor wants to send a message to another, first it checks whether the receiver processor (accept signal) is ready to get it. If it is ready, then the sender processor sends the message otherwise it stalls until the receiver gets ready. The same protocol is valid for a processor which wants to get a message.

![Figure - 2.3 Processors connected by an Ambric channel](image)

The receiver object has to stall until sender object send something through the channel. The channels can hold up to 2 words which is rather a small buffer. However, in this thesis approach, on some channels larger buffers were needed and therefore used.

RAMs are turned into hardware objects by using Memory Access Engines. Thus ram units can be used as buffers on the channels; detailed information will be given in Memory Architecture chapter.

2.3.4 Data Extent

The objects may encapsulate the similar data and send it as a structure to another object. These structures can be packets, vectors, lists etc... Ambric offers a general and efficient way of sending structural data. This can be implemented by a facility called Data Extent (DE). For a beginner user, DE can be an optional feature; however it is a cheap and powerful tool for the transfers of structural data between objects.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

![Diagram](image)

**Figure – 2.4** Structural data with Data Extent (DE) bit

Each word in ambric is 8 bytes plus DE bit. If the DE bit is not used, which means the data is not structured; the DE bit is set to zero. If the data is structured, the DE bit on all words is set to 1, but the one on the last word is set to zero. Thus, no header words are needed.

In this thesis project, the DE bits are used for external memory accesses. For the external memory accesses, TMS packets (a structured data type) are used. Thus DE bits are used in TMS packets.

For more information one can read [1].

### 2.3.4.1 TMS Packets

TMS is a standard DE-limited packet format which is used for memory requests, responses, commands and some other purposes. TMS packets can access any memory unit with no limit on distance. In our approach, TMS packets will be used to access external DDR memory.

While accessing a memory unit with TMS packets, firstly a header word must be sent to make either read or write request (the header word number can increase to 2 if longer than 16 bits addressing is needed). With the header, the type of request, the number of bytes those will be written/read and the address is sent. Then reading or writing can be applied.

For more information one can read *TMS packets* chapter in [1].

### 2.4 Processor Architecture

There are two types of processors in Ambric architecture, SRD and SR. These processors have some local instruction memory and they get the instructions through the channels. Processors execute the instructions streamed from local memory, by a program counter by using random access channels. Ambric processors do not have interrupt because there is no need since each processor is dedicated to own task.

#### 2.4.1 SR and SRD Processors

SR processors are 32 bit streaming RISC processors. SRD processors are also 32 bit streaming RISC processors but in addition to this they have DSP extension. So when compared, SRD has
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

more capabilities than SR. Thus SR processors are mainly used for small tasks, when SRD is not necessary, like forking or as splitters. SRD is more like extension of SR. Both processors can be programmed. The local memories are parity checked. Below, a comparison table between SRD and SR processors can be seen:

<table>
<thead>
<tr>
<th>SRD Processor</th>
<th>SR Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits wide instruction</td>
<td>16 bits wide instruction</td>
</tr>
<tr>
<td>Accepts inputs from 2 channels and feeds an output to a channel each cycle</td>
<td>Accepts 1 input and feeds 1 output each cycle</td>
</tr>
<tr>
<td>Three ALUs (two in series and a parallel) including multiply and barrel shift</td>
<td>One ALU</td>
</tr>
<tr>
<td>20 general purpose registers + previous result register &amp; 64 bit accumulator</td>
<td>8 general purpose registers &amp; 32 bit accumulator</td>
</tr>
<tr>
<td>256 (32bits) words - local memory + 2K more instructions from the RAM unit next to it</td>
<td>64 (32bits) words - local memory</td>
</tr>
<tr>
<td>Streaming and random access to the RAM unit through a dedicated RAM unit read/write engine</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 – Comparison between SRD and SR processors

![Figure 2.5 - Architecture of SR processors](image-url)
For detailed information read can refer to [1].

2.5 Memory Architecture

In the architecture of Ambric, like any other objects, memories are encapsulated and can be read and written by using channels. There are access engines between processors and memories. Access engines work in 2 forms. In first form access engines set some region of memories as FIFOs, which is the most common way of memory usage, and connect them to the channels. In the second form, access engines enable random access read and write. The written data is sent over channel like the read data. Requests can be: read or write the data.

The memory is distributed on the chip as ram units. In addition to this each processor has a small internal memory to keep instruction data. In case of requirement, an SRD processor can use an external memory, which is in the RU, to keep the instruction data. Each compute unit has a ram unit as a neighbor. In a ram unit there are 4 ram banks and each ram bank has the size of 2Kb which means 512 words.
Many buffers are used in this thesis work. Additionally, very large buffers were needed by several processors; therefore external DDR memories are used as well. Detailed information will be given in further chapters.

For detailed information about the memory architecture, reader can refer to [1].

2.6 Programming Model

To reach the practical programming model, components shall be familiar, productive, and easy to use and the model should be scalable to any size.

To obtain the practical programming model, Ambric uses block diagram method because of its familiarity to parallelism. In Ambric, block diagrams are the objects and lines are the channels with defined behavior. Thus, the whole block diagram becomes the structure of objects and channels.

By having hierarchy with interfaces and strictly encapsulated objects, design reuse becomes an important issue. So scalability becomes available. The objects can be reused even by different designs, because they do not share anything with any other which they can harm. Instead of using shared memory for communication, these objects communicate with each other through
the point to point channels which also arrange the synchronization between objects. Synchronization is done during runtime, not during compilation time.

Figure 2.8 - The hierarchy between design, structure and implementation parts of the programming model

The design may consist of several objects which are built by using astruct language. The connection between the objects builds the structure of the whole design. These objects can be either composite or leaf objects. Each leaf object is binded with a java file which is coded by using ajava. These objects can use different java files depending on the binding.

The memory is distributed with limited amount which is a constraint. However, it is not something new for the developers.

2.6.1 Structural Object Programming Model

Objects are programmed by conventional sequential code in structural object programming model of Ambric. (The compiler [aDesigner] is built on eclipse and uses ajava and astruct languages.) Smallest programmed object is called leaf which can be either a processor or a memory. Leaves can be connected to each other by channels, so comes out composite objects which can include leaves and/or other composite objects. Objects run independently with their own speed without having any effect on the others. When an object feels a stall or that it is idle, it reduces the clock frequency by itself in order not to consume much power. On the other side, processors, memories and the channel hardware synchronize themselves transparently, dynamically and locally even they do not run with the same speed.

Channels provide hardware level interface. So connecting the objects becomes easy since they use only the channels for interaction. Also assembling the objects into composite objects becomes easy.
During programming, conventional techniques can be used. However, the processors miss some operations like division and modulus. The division operator can be used only in the constructor. In addition to this, there is no floating point, and no array implementation except in constructors, which are two other constraints.

One can think that Ambric has the same model of computation (MoC) as Communicating Sequential Processes (CSP). However that way of thinking is not correct. MoC of Ambric is inspired by CSP. (CSP is a formal language to describe interaction patterns in concurrent systems [3].) However Ambric still has differences with CSP. In CSP components are sequential processes and they run concurrently. The objects in Ambric correspond to the components in the CSP. In CSP message passing is done synchronously which means there is no buffer between processors and they wait until a synchronization point to do the transfer. However in Ambric objects send messages to each other asynchronously and channels arrange the synchronization and the registers have small buffers. Additionally, Ambric MoC has similarities with Kahn Process Networks (KPNs) [4] and it can be defined as a process network with channels which are bounded FIFOs. In the other hand in KPNs the channels are unbounded FIFOs which means that you can always push data through the channels. In KPNs, get operation blocks until an input is available like the Ambric registers stall until there is a valid input. Another common property is that the objects (processes) cannot check if there is an input on the channel. But sometimes this property becomes a constraint, especially while converting from CAL language. In the CAL language, the inputs can be controlled without consuming them; however it is not possible for ambric model. Thus some software solution is used in our approach. More details will be given in further chapters.

2.6.2 Programming and Analysis Tools & Languages

All implementations are done on a platform called aDesigner. aDesigner has been produced for Ambric. It has been built on Eclipse environment. Thus the programming language is built on
Implemen
tation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

java and called ajava. The syntax is the same. However, ajava has some limitations when compared to java. For instance all types of arithmetic operations are not supported in ajava. There is also a structure language called astruct, used to implement the structure of objects and so of the applications. Creation of objects and channel connections between these objects are done in structure files. In addition to java, assembly can also be used. However, for the beginning java is much easier and we use java in our implementations. The structure files and the program files are binded to each other by the programmer while implementing the structure. Thus each object knows which program to run.

The program that will be run by the processor is written in ajava. Since java is a common programming language, programming the object is not a hard work, except some drawbacks of ajava. This program is sent to all objects that are binded to that java file. With astruct language, objects, numbers and names of input and output channels, properties (variables) of the objects, triggers, tags and the binded java files can be defined. Thus the structure of the object is built using astruct language and the task that will be processed is defined using ajava. These two languages complete each other. In our implementations we use both tags and triggers. Thus we will go into further details about them. By connecting the objects with each other, composite objects are obtained. This operation is done with astruct language and while doing this operation, no binding with java file is used since there is no java code for composite objects. Defining the design of the application is also done by using astruct language. In design file, a sample of the highest level composite object is defined. If there are any input-output channels, they are connected to the proper inputs and outputs.

A structure file sample (which defines the channels, variables and java file of an object) can be seen in figure 2.10.

```java
package single;  //name of package that includes this object

interface calculator {

    inbound in;      //input channel declaration
    outbound out;   //output channel declaration

    /*the properties below, are used during object declarations for dynamism purposes*/
    property int A_row;
    property int B_row;
    property int B_col;
}

//binding the object with the java file
binding Jcalculator implements calculator{

    implementation "Calculator.java";  //The java file that will be run by this object
}
```

**Figure 2.10** – Structure file sample written in astruct

In the interface definition, input/output channels and properties of the object are defined. Properties are used to parameterize the object. The values of the properties are passed to the Java constructors at compile time. They are static for the life of the object. In the binding definition,
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

the structure (object) is binded to a java file. It means that, the object will run that java file. The object runs the run function of the java file as default. However this can be changed by using triggers.

Triggers are used to trigger the execution of actions. A trigger can execute an action for once or many times. For example this trigger executes myAction only once.

\[
\text{trigger } t = \text{myAction};
\]

The action can be triggered continuously by using regular expressions (not all kinds of expressions can be used). For example the following trigger will execute myAction repeatedly

\[
\text{trigger } t = \text{myAction}^*;
\]

By default (if no trigger is defined), run method of binded java file will be executed repeatedly.

Triggers can be used with tags as well. Objects can pass tokens between each other. For instance an object may want to trigger a specific action in another object. So, a token can be passed to the target object in order to trigger the specific action. In the object structures, actions can be defined in a way that, they get executed in case of receiving some pre-defined tokens (tags). An instance of usage of triggers with tags can be seen in the below figure:

```
Definition of tags:
package myActions;
tag {
    myTag, // Invoke action3
    Left, // Call myAction1
    Right // Call myAction2
};
```

**Figure 2.11 – Tag definitions**

The object, that receives the tags, must have trigger definitions such as;

\[
\text{trigger } t = (\text{myAction1{in:Left} | myAction2{in:Right}})^*;
\]

In this case, depending on the input tag (which will come on channel ‘in’ and can be either Left or Right), either myAction1 or myAction2 will be run. And this procedure will run repeatedly. Trigger definitions shall be unambiguous, so that the system knows which action to run.

For more information about tags and triggers reader can read [1].

Another structure file sample (which defines a composite object) can be seen below:

```
package matrixMult2;

import matrixMult2.Adder;
import matrixMult2.Multiplier;
import matrixMult2.Splitter;

interface MatrixObj {
```
Splitter, Adder, Multiplier are the names of the structures and splitter, mult1, mult2 and adder are the objects that are included in the composite structure. This composite structure has an input and an output channel which are defined in the interface part. In the binding definition of composite objects, the object does not get binded to any java file. Instead, objects and the channels between objects are defined.

A design file sample (still a structure file).

```java
package matrixMult2;
import matrixMult2.MatrixObj;
import astruct.pcie.Vio;

interface Root{};

binding CRoot implements Root{
    Vio vio = {numSinks = 1, numSources = 1};
    MatrixObj matrixObj;
    
    channel ch0 = {vio.out[0], matrixObj.in},
    ch1 = {matrixObj.out, vio.in[0]};
}

design Matrix {
    Root root;
}
```

Figure 2.13 – Design file sample

In the above design, Vio library defines inputs and outputs to the files in simulator. The MatrixObj object, which is a composite one, is connected to these inputs and outputs. The top level interface is defined in the design file. This top level interface is the application itself.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

aDesigner has two kinds of simulator: Java Virtual Machine (JVM)[1] and Instruction Set Simulator (ISS)[1]. JVM is more flexible than ISS. Unfortunately, performance analysis can only be done while using ISS. When JVM is selected as the simulator, all debugging features of java can be used and there is no limitation of java subset which is used to build designs for the hardware. In addition to the simulators, aDesigner creates machine code for hardware chip. So it can program the hardware device as well.

In aDesigner, as mentioned in processors chapter, there exist addition, subtraction, multiplication and barrel shift operations. There is no division operation and array definitions can be used only in constructors. These are two serious drawbacks. Another drawback is lack of floating points. Only integers are supported in aDesigner.

There is a graphical window in aDesigner which shows the graphical structure of the application, with objects and the channel connections, after running the design. Though sometimes becomes useless because of the difficulty of changing the graphical layout. But still helps to see the connections between objects.

![Figure 2.14 - A screenshot of graphical layout of a design, created by aDesigner (splitter, mult2, mult1 and adder are leaf objects)](image)

**Performance Analysis Tools in aDesigner and Analysis Methods:**

To analyze the performance, aDesigner has an attachment on the simulator as an analyzer. But this analyzer is dedicated more to analyze code efficiency than analyzing the application run time. It allows the programmer to analyze only single processors by giving some cycle counts as total cycle count, busy cycle count and stall cycle count. Thus, neither composite objects nor the applications can be analyzed with this analyzer. In order to get the run time for entire application, one have to combine the cycle counts of each processor when they run in serial. So it is needed to find out when they run in parallel and when they run in serial. In addition to the cycle counts, the analyzer gives also the data in the registers like inputs, outputs and program counter with the first and last cycle numbers of the data’s duration time. (Like between 45th and 97th cycles there is 0x10000 in input1.) That is the main source that can be used for performance analysis in parallel implementations.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

The analyzer ignores the channel stalls which means that it does not count the cycles when the processor waits for an input or when it waits to send an output. The length of a cycle (in terms of seconds) can change from a processor to another processor. Because the processors can run with different clock frequencies as mentioned before. So when a processor counts two cycles, another one can count only one cycle or four cycles for the same time, depends on the clock frequencies of the processors. When there is any negative value in any register, the performance analyzer shows it as 0x0fffffff. This may become a serious obstacle for analyzing methods.

Because of the drawbacks and constraints, the performance analyzer is not very reliable.

To get the performance analysis of a processor, the programmer needs to define the processor and a time interval (program counter’s start and end addresses). In general time interval is from the beginning (pc address = zero) till the end of the process. There is another method. In this second method, the programmer defines a cycle count (t) and gets the performance analysis of defined processors at each t cycles. However, the above performance analysis are not used in this thesis approach, instead, a newly developed method is used, which is developed by Ambric department in Nethra Company.

![Figure 2.15 - A screenshot of aDesigner/Performance Analysis part](image)

Using The Performance Harness Library

This library has been provided by the ambric department of Nethra Imaging. It is still under development. It includes several objects to be used for the performance measurements.
The PerfHarness (performance harness) library object can be used to measure the latency and throughput of a system. To avoid the IO transfer irregularities, Fifo preloaders or DDR preloaders are used. For small sizes of inputs fifo preloaders are useful however when the input size becomes larger, the DDR fifo preloaders must be used.

The PerfHarness object has 4 channels, 2 inputs and 2 outputs. One input and one output channel is connected to the system which is under measurement. The second input channel is used for receiving the input data. The second output channel is used to write out the chosen measurement results. The types of measurements are latency and timing.

The performance analysis (PerfHarness) library is developed to analyze designs which write the same amount of the output when the same amount of in the input is given. For example if the input is 1,000 words and the design produces 10,000 words, then when an input of 10,000 words is given, the design should still produce 10,000 words of output for each 1,000 words of input. Therefore the input of 10,000 words can be divided into 10 pieces of workloads of 1,000 words. However because of the nature of the decoders, the implemented design does not produce the same amount of the output when the input is divided into workloads. The reason of using this method is to ignore the first analysis result and get consistent results.

At the beginning, the results were varying due to the mentioned factors. Later a delay gate was added which waits long enough for the configuration of the chip to be done. Then sends the data through the system under measurement and at the end we get the reliable results. Another issue was the DDR controllers. We have been using both of the DDR controllers on our design and additionally one DDR controller is needed for the performance analysis. Changing the port of the controller, either on our design or on the performance harness library object fixes the problem. Further problems will be explained in coming chapters. For more information about the library, reader can refer to [19].
3 Video Encoding/Decoding & MPEG

3.1 Video Introduction

Video is a sequence of still images. In general it covers the technology of recording, capturing, processing, receiving/transmitting, reconstructing and storing the time sequence of the images representing scene. When a series of two dimensional bitmap digital images are used with a constant rate, they create the digital video. One two-dimensional bitmap image is called as the frame of the video. A bitmap image consists of an image which has two dimensions. Each element in the array is color information for an individual pixel. The speed of displaying frames is called frame rate. Generally frame rate is measured by the number of frames per second. If we call the height of the frame as H and weight of the frame as W, then the size of the frame is W x H. For instance, for an image which is 16 by 16 then the size is 16x16 = 256. The color information of a pixel can be represented by various amounts of bits. This gives the color depth of the pixel. The more bits the more variations of colors can be reproduced by the pixel. In a video, the color depth is constant for all images/pixels.

If we use an uncompressed video sequence which is 30 minutes long, has frame size of 1024x768, color depth of 24 bits and frame rate of 30, as an example, this example would have the following properties:

- Bits per frame = 24
- Pixel per frame = 1024 x 768 = 786,432
- Bits per frame = 786,432 x 24 = 18,874,368 = 17.31 Mbits
- Bit rate per second = 17.31 x 30 = 519.5 Mbits / sec
- 30 minutes = 1800 seconds
- Video size = 1800 x 519.5 = 935,115 Mbits

With compression the size can be reduced significantly. More information about compression will be given in Video Compression chapter.

3.2 Color Spaces

There are two different representations of color information of a pixel, RGB and YUV.

RGB is the most known and in the same time the most common color representation of computer graphics. RGB stands for red, green and blue respectively. In this representation, color of a pixel is created by the mixing red, green and blue colors each with a predefined proportion. This makes the compression of the data difficult due to the requirement of using data of each color.

Human visual system is more sensitive to brightness (luminance) than to color (chrominance). This situation gives the opportunity of using another representation which is more efficient for compression, YUV. YUV representation focuses on the brightness of the pixel.

The luminance information (Y) is extracted from the RGB information by the following formula:

\[ Y = krR + kgG + kbB \]
where \( kr, kg \) and \( kb \) are the weighting factors and in normal processing \( kr = 0.299, kg = 0.587, \)
\( kb = 0.114 \) \((kr+kb+kg = 1)\). By this process, the original image is converted into a black& white image.

The color values can be calculated by subtracting the luminance \( Y \) value from the RGB values

\[
\begin{align*}
Y &= 0.299R + 0.587G + 0.114B \\
Cr &= 0.713 (R - Y) = V \\
Cb &= 0.564 (B - Y) = U \\
Cg &= G - Y \text{ (not used, redundant)}
\end{align*}
\]

There are three different sampling modes using YUV information. 4:2:0 is the most common sampling mode/pattern. In 4:2:0 mode, \( Y \) value is sampled for every pixel. On both, horizontal and vertical directions, the \( Cb \) and \( Cr \) values are sampled as half of the luminance, which means every group of 4 luma samples have one \( Cb \) and one \( Cr \) sample.

For 4:2:2 mode, on the columns, \( Cb \) and \( Cr \) have the same number of samples however on the rows, they have half number of samples. Luma is sampled for all pixels.

For 4:4:4 mode, the resolution is the same for all \( Y \), \( Cb \) and \( Cr \) on both rows and columns.

![Figure 3.1 – Video Sampling Modes](image)

### 3.3 Video Compression

Video compression is known as video coding as well. With the increase of quality of images, the video sizes increased very fast. As a result, storing and transmitting the video data became more difficult and demanded more sources. Therefore, compression of video data began.

Video compression can be called as space saving representation of video data since an amount of space can be saved by compressing the data. Thus storing or transmitting the data gets easier for both analog and digital video. The purpose is to reduce the size of the data by compression. The compression techniques will be explained shortly.
3.3.1 Redundancy Reduce

The main goal of video compression is to convert the raw video into a smaller one, with smaller number of data, which can be reproduced without significant loss by using reverse operations. First step of video compression is reducing the redundant data on the video sequence. In total, there are 3 types of redundancies: spatial redundancy, temporal redundancy and frequency domain redundancy.

Spatial and Temporal Redundancy:

Pixel, picture partitions or symbols have very high probability of any correlation with the adjacent pixel, symbol or picture both partition in the same frame and across frames. For instance, a large partition of a picture which has the same or very similar color on every pixel can be called as a spatial redundancy [5]. This means that, to some extent the value of a pixel can be predicted by using the neighbor pixels. For a frame, instead of keeping the whole data of each pixel/picture partition/symbol, only the difference between adjacent pixels/picture partitions/symbols can be saved. Hence, inside one frame the spatial redundancy can be removed. This method is a part of predictive coding [4] and called as intra-prediction.

The same situation appears on the time domain as well. In a video sequence, the same pixels or the same objects may occur repeatedly in many frames. The consecutive frames are similar to each other except the situation when the entire scene (the content of the video) changes. This redundancy is known as temporal redundancy. Instead of keeping the data of each frame, only the small difference between consecutive frames can be kept. Hence the temporal redundancy can be removed. This method is a part of predictive coding and called as inter-prediction.

In a different way, the operations for temporal and spatial redundancy can be called as motion estimation and compensation. In this thesis work there will be a block (consisting of several processors) for motion compensation.

Frequency Domain Redundancy

Human visual system is not sensitive for very high frequencies. Therefore removing them in a frame (such as edges of objects) is not an obstacle for the human eye to recognize the objects. So the picture can be smoother by applying high frequency filter.

Discrete Cosine Transform (DCT) [7] is a common transformation used in video processing. By applying DCT to the picture, the frequencies which do not contain much information can be discarded. Thus a large amount of compression can be obtained. At the final phase of this process the picture can be reproduced without significant loss by using the reversed DCT. Moreover, quantization is another method which can be used for frequency domain redundancy. The whole operation of reducing frequency domain can be called as Transform Coding [5].

After the operations for redundancies, entropy coding can be applied to the video data in order to achieve further compression.
3.3.2 Motion Estimation

The basic goal of motion estimation is finding the similarities between the consecutive video frames and exploiting them.

Providing there is a common region in consecutive frames, instead of keeping/transmitting the whole data about the region, only the difference between the regions can be kept in order to save space. This difference is called residual as well. It is produced by subtracting the matching region from the current region.

In the decoder part, the region can be reconstructed by using the residual and the matched region. The residual is added to the matched region (prediction). With the increase of similarity, the number of zeros in residual increases as well. Thus, the temporal redundancy is reduced. Multiple frames before or after the current frame can be used as reference for the current frame.

In general, motion estimation and compensation are applied by using rectangular blocks of luminance and chrominance components. The most common size for a block is 16 x 16 for luminance and 8 x 8 for chrominance components. These blocks are called as macroblocks. In the current video coding standards (MPEG and ITU-T series) macroblock is the basic data unit for motion estimation and compensation. A 16 x 16 macroblock consists of one luminance
sample block, one 8 x 8 Cr sample block and one 8 x 8 Cb sample block. The smaller macroblock size the better performance for motion estimation and the more computation. In the new standards, macroblock of smaller sizes started to be used.

**Figure 3.3 – Macroblock (4:2:0)**

### 3.3.3 Motion Vectors

Motion vector is used for finding the place of matching regions on the reference frames. It is a pair of coordinate offsets. These offsets give the position of the best matching region when added to the position of the current region. Motion vectors are encoded and transmitted with the residual. On the decoder side, the matching regions on the reference frames are found by using the motion vectors. The current frame is reconstructed by adding the residual to the matching frames.

The accuracy of motion estimation and compensation depends on the granularity of the distance between pixels. When non-integer coordinates are used, which means high granularity, the encoder gets more choices while searching for the best matching region. Thus better matchings can be obtained. This operation results further reduce of the redundancy in the residual. In practice, integer pixel positions, half pixel positions (2 times more granularity) and quarter pixel positions (4 times more granularity) can be used.

### 3.3.5 Discrete Cosine Transform (DCT)

In the encoder, the step after the motion estimation is transformation, to achieve further reduction of the frequency redundancy.

Discrete Cosine Transform is a block based transform which is similar to Discrete Fourier Transform. Both methods transform a signal from the spatial domain to the frequency domain. DCT is applied to a block X as follow:

\[ Y = A X A^T \]

and inverse DCT (IDCT) is applied on Y as follow:
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

\[ X = A^T Y A \]

where A is the transform matrix of same size as X, Y is the result block in frequency domain. Further details about matrix A can be found in [6].

Y is the coefficient matrix of the block X in the DCT domain. \( Y(0,0) \) is the coefficient in the top left corner which is called as DC coefficient. The rest of the Y matrix is AC coefficients. DC coefficient is much greater than the AC coefficients. The reason is, in most cases most of the signal energy lies in the low frequencies (top left corner) of the DCT. Since the human visual system is not so sensitive for high frequencies, some parts of the AC coefficients can be ignored without any significant loss. AC coefficients are rather small and ignoring them does not cause any big distortion of the image. This operation is done by quantization method. In the mean time, the DC coefficient is kept well.

For detailed information about DCT, one can read [6].

### 3.3.6 Quantization

After the DCT transform, the next step is quantization. It is applied to the DCT coefficients to truncate the magnitudes. Consequently the number of bits which are used to represent the coefficients decreases.

This operation can be applied on both individual coefficients (Scalar Quantization) and on a group of coefficients (Vector Quantization). For vector quantization, matrices are used.

An example of quantization can be given as:

\[ Y = \text{round} \left( \frac{X}{QP} \right) \cdot QP \]

where X is the input, Y is the output and QP is the quantization constant.

For video coding, most often vector quantization is used. In this case QP becomes a quantization matrix. In an ordinary quantization matrix, the values which are closer to the top left corner (corresponding to the low frequency positions) are smaller than the ones which are closer to the bottom right corner (corresponding to the high frequency positions). The reason is simple, canceling the high frequency parts of the image in order to reduce the frequency redundancy.

### 3.3.7 Zig-Zag Scan

The next step after the quantization is zig-zag scan. After the quantization most of the AC coefficients become zero and the coefficients which are close to the top left corner are left non-zero. Zig-zag scan is used to get the zero values together by rearranging the positions of the coefficients. For this reason while transmitting or saving the data, the zero values can be represented with less number of bits.
The operation for reducing the number of bits is called run-length encoding. This method represents a string of data as pairs. First element of the pair is the number of zeros and the second one is the non-zero value following the zeros. A simple example can be given as:

The following data string \{0, 3, 0, 0, 0, 5, 0, 0, 2, 1, 0, 0, 0, 1, \ldots\} can be presented as:

\[(1,3), (3,5), (2,2), (0,1), (4,1), \ldots\]

### 3.3.8 Entropy Coding

Entropy coding is the last step of the video encoder. Entropy is the measure of the similarity in a given data set.

In entropy coding, each data value is given a unique code word. The value which appears the most gets the code word with least number of bits. The more often the value appears, the less number of bits is used for it. These values are found by using entropy. So the data set (in our case video data) is compressed once again.

The most common entropy coding methods are Huffman encoder and the arithmetic encoder. Nevertheless in the latest MPEG standards different methods are in use such as Context-based adaptive Variable Length Coding (CAVLC) and Context-based Arithmetic Coding (CABAC).

### 3.2 MPEG

#### 3.2.1 Overview

“The Moving Picture Coding Experts Group (MPEG) was established in January 1988 with the mandate to develop standards for coded representation of moving pictures, audio and their combination.” [8]
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

Since the encoder is more complex than the decoder, the methodology of MPEG compression is considered as asymmetric. Whereas the encoder is the smart part of the methodology and the decoder just applies predefined actions. This situation is good for broadcasting. The number of complex encoders is less than the decoders. The standardization of ISO is for the decoder. The way of decoding the streaming data is standardized. On the other side, the encoders may vary which means different developers can develop different encoders. This may cause a competition between encoder designs, yet a decoder may work with the output of any encoder.

The MPEG standards consist of different parts. MPEG has standardized the following compression formats and ancillary standards:

3.2.2 MPEG-1

It is the first MPEG standard for audio and video compression and was developed to allow moving pictures and sound to be encoded into the bit rate of a Compact Disc. MPEG-1 is used on Video CD, SVCD and can be used for low-quality video on DVD Video. It was used in digital satellite/cable TV services before MPEG-2 appeared. To be able to meet the low bit requirement, MPEG-1 downsamples the images, as well as uses picture rates of only 24–30 Hz, resulting in a moderate quality. The resolutions that MPEG-1 provides are 352x240 for NTSC and 352x288 for PAL. It includes the popular MPEG-1 Audio Layer III (MP3) audio compression format. (NTSC is a color TV standard developed in the U.S. in 1953 by the National Television System Committee. PAL, short for Phase Alternate Line, is an analogue television encoding system used in broadcast television systems in many countries.)

3.2.3 MPEG-2

MPEG-2 was published by ISO/IEC in order to produce better data rate and video quality than MPEG-1 standard. The same coding technique as MPEG-1 is used however the resolution of picture is increased to 720x486.

MPEG-2 has a layered structure which is an important feature. Therefore MPEG-2 supports scaling. Systems with different scales can use the MPEG-2 standard. Accordingly the MPEG-2 is able to encode/decode a video stream with different qualities depending on the requirements. In addition, MPEG-2 is compatible with MPEG-1 standard so that MPEG-1 videos can be played on MPEG-2 players.

MPEG-2 is used widely for transmitting the digital television data over the air, on cable and for direct satellite broadcast TV systems. It is used as the format of the videos which are distributed on DVD and similar discs as well. It is the core of most digital television and DVD formats. However it does not specify them all. Institutions can manipulate the MPEG-2 standard for their own needs.

3.2.4 MPEG-4

MPEG-4 uses the same basics as MPEG-2. However to reach better compression than MPEG-2, MPEG-4 uses additional coding tools and involves higher complexity. Additionally, MPEG-4
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

gets closer to the computer graphics. In complex profiles of MPEG-4, the MPEG-4 may become a rendering processor and in the compressed video data, 3 dimensional objects and surface textures can be described. Consequently the MPEG-4 introduces objects into video compression area. A user may access an object and manipulate it.

An MPEG-4 decoder may decode a compressed video data with quality between 5Kbit/s to 5 Mbit/s. MPEG-4 has two main parts; MPEG-4 Part 2 (Simple and Advanced Simple Profile) and MPEG-4 AVC (MPEG-4 Part 10 or H.264). The decoder in this thesis is designed for simple profile of the MPEG-4 which is often used for the systems where low bit rate and low resolution are required such as networks with low bandwidths, cell phones, surveillance systems.
4 Reconfigurable Video Coding (RVC) & CAL Actor Language

The RVC framework is a new standard of ISO which is under the final stage of standardization. The framework aims to provide library components for different video codec specifications. The main idea is to use a library of standard coding algorithms in order to create a decoder of an existing standard or a totally different decoder to gain better results for application-specific constraints. Thus instead of using monolithic algorithms, the codecs can be specified at the level of library components.

CAL is a dataflow/actor based language and has been designed to model complex signal processing designs. The RVC framework is based on this new language. CAL is used to specify the standard library and to instantiate the RVC decoder model. CAL actor language uses actor programming and dataflow in order to provide concurrency for the algorithms.

4.1 RVC Framework Introduction

Due to the development of the video coding techniques, the solution complexities increase rapidly. Additionally, presenting large overlap between consecutive versions of the standards becomes complex as well. As a consequence, adding new coding tools to the standard involves modifying all of the components whereas only a few interface and tools are changed. With the wide variety of the coding tools, another problem occurs which is choosing the proper subset of coding tools for a specific application. These subsets are correspond to ‘profiles’ in MPEG. It is not possible to satisfy a variety of specific applications by using the average subsets. On the contrary, if there are too many profiles, interoperability cannot be guaranteed. Because of such drawbacks, development of RVC standard was needed. The main idea of the RVC standard is to make it possible to develop a decoder at a higher level of abstraction than the one which is developed by monolithic specifications based on C. Instead of using low leveling coding, higher level coding is the aim, such as abstract models based on the library components.

In the multicore era, parallelism and concurrency are the fundamental aspects. Thus CAL becomes the core of RVC since it provides the intrinsic concurrency by using the actor programming and notions of dataflow. Additionally, standard RVC framework is supported by another design framework including a simulation platform and some conversion tools. These tools are Cal2C [11, 12] providing direct conversion from CAL to C and Cal2HDL [13, 14] providing direct conversion from CAL to HDL.

4.2 ISO-MPEG Standardization

The old specifications of MPEG standards (such as MPEG-1, MPEG-2, MPEG-4, AVC [Advanced Video Coding], SVC [Scalable Video Coding]) lack flexibility. Because they have been implemented by using monolithic algorithms based on low level languages such as C and C++. Likewise since the algorithms are monolithic, it is not possible to take them from different standards and combine. Therefore achieving the requirements of specific applications is not enabled. On the other hand, there are tools defined for each profile level however not all of them are used in all applications. In most of the cases, the codecs are not used at their full potential. Additionally, in some applications the codecs need unnecessarily complex implementations.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

However, when a decoder is designed for a given standard, it has to meet the requirements of all applications. In order to overcome these problems, a profiled version of CAL language has been chosen for the RVC standard.

The MPEG standard library which is written in CAL, includes video technology from all MPEG standards. The video coding tools of this standard library can be used via RVC framework to implement different codecs by using different combinations of functional units (FUs, such as actors in programming language). For more information about RVC, one can read [15].

Implementing a new MPEG RVC decoder is based on CAL dataflow model of computation and there are 3 main steps to implement one: A decoder description, an abstract decoder model and the final decoder implementation. For more information about these steps, reader may refer to [9]. Thus developers may design new decoders by using a set of already-defined modular components which are from the RVC standard library. In addition to providing the flexibility required by the application, CAL based approaches become a start point of codec implementations for multicore platforms.

4.3 The CAL Actor Language

“CAL [16] is a domain-specific language that provides useful abstractions for dataflow programming with actors.”[9] CAL actor language was created at UC Berkeley as a part of Ptolemy project.

“A Cal actor is a computational entity with input ports, output ports, states and parameters.”[17] Several actors can be connected to each other and they can communicate via tokens along the unidirectional channels. When connected to each other, actors create a network and so models or applications. Actors are called to be fired when they are executed. During an execution, the actor consumes the inputs on the input port and writes output to the output ports.

Cal language is not designed to be a fully operational language. Instead it is designed to be a small language to help to provide the required infrastructure.

4.3.1 Actors

A simple example of CAL actors can be seen below:

```
actor Add() T A, T B => T Out:
    action [a], [b] => [sum]
    do
        sum := a + b;
    end
end
```

Figure 4.1 – A simple CAL actor
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

In the above example (figure 4.1), the actor is called Add and has two input ports (A and B) and one output port (Out). These ports are all type of T. The actor has only one action which accepts one input from each input ports and writes an output to the output port. The action is triggered if the conditions are satisfied. In the above case the condition is availability of inputs on both input ports. There is a need of at least one input on both input ports.

An actor may have a number of actions. In further chapters examples will be given.

### 4.3.2 Actions

An action is the smallest piece of the computation which the actor performs. Usually the actions are responses to the inputs. There are 3 things which can be performed by an action:

- consuming the input
- producing the output
- changing the state of the actor

If the actor has a number of actions, whenever it is fired, it needs to choose which action to run. Usually it depends on the availability of the inputs on the input ports and possibly on the further condition of value of input and state of the actor. If defined, scheduling and priorities also affect the choice of action. Scheduling and priorities will be described in a further chapter.

On top of an action (head part), the input ports and the inputs types to be used in the action are defined as well as the output ports and the output types. The input tokens, which will be used, and the output tokens, which will be written, are described as well. In the body of an action, the changes that will be done to the actor state, the computations to be done on variables, which can be in the output port expressions, are defined. In the previous example, [a], [b] means a is the input on the input port A and b is the input on the input port B. The same pattern could be presented by using port names, as “A:[a] B:[b]” For more information about action matching, port patterns & syntaxes reader can refer to [17].

### 4.3.2.1 Guards

The conditions which are used to define which action will be run are not only the input conditions. Guard expressions are used in order to impose extra conditions to run an action. The guard consists of boolean expressions. An example can be seen below:

```cal
actor Sorter [T] ([T -> Boolean] f)
T input => T output1, T output2 :
  action [a] => [b], []
  guard f(a) : end
  action [a] => [], [b]
  guard not f(a) : end
end
```

**Figure 4.2 - A CAL actor using guard expressions**
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

In the above example (figure 4.2), the Sorter actor has two actions, one input port and two output ports and gets a function f as a parameter. If the evaluation of function f is true then the first action is executed otherwise the second one. Since the action bodies are empty, they only read the input and forward it to a chosen output port (depending on the guard expressions). In another case, the guards can use local variables such as in the example below:

```pascal
actor FairMerge() [T] T A, T B = > T C :
    Integer i : = 0,
    action [a],[i] => [a]
    guard i = 0 do
        i = 1
    end
    action [], [b] => [b]
    guard i = 1 do
        i = 0
    end
end
```

Figure 4.3 - A CAL actor using guard expressions

In the FairMerge actor (given in figure 4.3), there are two input ports and one output port. Depending on the value of the local variable i, either first or second action is executed. Additionally if the types (T) are not the same, the behavior of the actor is not defined.

### 4.3.3 Action Scheduling and Priorities

The behavior of an actor can be constrained by terms of action firing sequence. Actions can be labeled and the labels can be used in order to define the importance of the actions or the firing sequence of them. The importance of actions is defined by priorities and the firing sequence is defined by the action schedule. The firing schedule can be imagined as a state machine. In the previous examples, labels for actions were not used. Thus they were anonymous.

The schedule can be defined by using two different syntaxes. Both examples can be seen below:

```pascal
actor FairMerge2 ()[T] T A, T B => T C :
    a1 : action [a], [] => [a]
    do ... end
    a2 : action [], [b] => []
    do ... end
    schedule
        (a1 a2)* | (a2 a1)*
    end
end
```

Figure 4.3 - A CAL actor using schedule
All operators of regular expressions can be used in the action schedules. Another example is as below:

```plaintext
actor PingPongMerge () Input1, Input2 => Output:
    A : action Input1 : [x] => [x] end
    B : action Input2 : [x] => [x] end

    schedule fsm s1 :
        s1 (A) => s2;
        s2 (B) => s1;
    end
end
```

**Figure 4.4** – A CAL actor using schedule

In PingPongMerge actor, the schedule is used to toggle between the actions A and B. s1 and s2 are the states used by the schedule.

Another method of defining firing sequence is using priorities among the labels of actions.

```plaintext
actor Foo() [T] T A, T B => T C :
    a1 : action [a], [b] => [a, b]
        do ... end
    a2 : action [a], [] => [a]
        do ... end
    a3 : action [],[b] => [b]
        do ... end
    priority
        a1 > a2,
        a1 > a3
end
```

**Figure 4.5** – A CAL actor using action priorities

In the actor Foo, without priorities, it is not defined which action will run when there are inputs on both input ports. By using the priorities, the ambiguity is resolved and a deterministic actor is created. The programmer may group the actions under the same higher level label such as:

```plaintext
s1.a1 : action ...
s1.a2 : action ...
a3 : action ...
```

In the schedule and priority clauses, s1 means all actions which have s1 as the top label. The priorities and schedule can be combined and used together.
4.3.4 Composition & Networks

To create a CAL application a number of CAL actors must be instantiated and connected which forms a CAL network.

A CAL network is similar to the composite objects of ambric architecture. They have their own input and output ports/channels and may include other CAL networks or actors. This leads to a hierarchical design approach.

While interconnecting the CAL actors and creating a network, the semantics are left to the framework where the actors are instantiated. This is similar to the design approaches in the ambric architecture. (In the ambric architecture the objects were instantiated either in the design file or in a composite object and the interconnection was done in the same framework.) An instant of a network description can be seen below:

```
network idctld (MEM_SZ) X0, X1, ROW => Y0, Y1, Y2, Y3:
var
    COEFF_SZ = 13;
    SCALE_SZ = 30;
    ACC_SZ = 24;
entities
    scale = Scale(isz = MEM_SZ, osz = SCALE_SZ, csz = COEFF_SZ);
    combine = Combine(isz = SCALE_SZ, osz = ACC_SZ);
    shufflefly = ShuffleFly(sz = ACC_SZ);
    shuffle = Shuffle(sz = ACC_SZ);
    final = Final(isz = ACC_SZ, osz = MEM_SZ);
structure
    X0 => scale.X0;
    X1 => scale.X1;
    ROW => combine.ROW;
    scale.Y0 => combine.X0;
    scale.Y1 => combine.X1;
    scale.Y2 => combine.X2;
    scale.Y3 => combine.X3;
    combine.Y0 => shufflefly.X0;
    combine.Y1 => shufflefly.X1;
    shufflefly.Y0 => shuffle.X0;
    shufflefly.Y1 => shuffle.X1;
    shufflefly.Y2 => shuffle.X2;
    shufflefly.Y3 => shuffle.X3;
```
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

```
shuffle.Y0 --> final.X0;
shuffle.Y1 --> final.X1;
shuffle.Y2 --> final.X2;
shuffle.Y3 --> final.X3;

final.Y0 --> Y0;
final.Y1 --> Y1;
final.Y2 --> Y2;
final.Y3 --> Y3;
```

**end**

**Figure 4.6** – A network description written in CAL language

The actors are instantiated under entities key word and they are connected under structure key word. The input and output ports of the network are defined on the top line where network key word is used.

Previously the interconnections were defined by textual languages and later this text was converted to FNL and vice versa. Along with the CAL language, a graphical framework so called Graphiti editor can be used in order to display, edit and save the network. The former textual formats are supported by this editor. An example can be seen below:

**Figure 4.7** – A sample network view on Graphiti editor (decoder is another network itself)

For further information about composition and networks reader can refer to [16, 17, 18].

### 4.3.5 Tools

The tools which are used for this thesis project are the CAL plug-ins for Eclipse environment. – Open Dataflow environment for CAL editing and the Graphiti editor for editing the network. Additionally Actors Tools have been used for performance analysis. For more information about Actors Tools, reader can visit [http://sourceforge.net/projects/opendf/develop](http://sourceforge.net/projects/opendf/develop). One of the plug-in
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

tools for eclipse environment is a simulator for CAL language however it does not support debugging, besides the simulations are very slow when compared to the run-time of the projects which are built by Actors Tools. Additionally console display can be used by this simulator and during the thesis for debugging CAL project; console display has been used even though it is not very useful.
5 Conversion of CAL to ajava & astruct

There are both similarities and differences between CAL language and ajava & astruct languages. CAL language covers both the structure and the program codes however in ambric architecture they are divided into ajava and astruct languages. In order to build the structure (interconnections of the processors and ram units, instantiation and replacement of objects, etc…) the astruct language is used. On the other side, for programming purposes, the ajava language (could be imagined as a subset of java) is used.

Since there are time dependencies on the MPEG-4 decoder project (written by using CAL language) ambric architecture is not free of the problems. These problems are solved by using buffers on the channels. For instance DDRModel actor executes actions depending on the arrival time of the inputs. The details will be given in the implementations chapter.

5.1 Guards

As mentioned in the previous chapters, guard expressions are used in order to impose extra conditions to run an action. This condition may use a local variable or an input value on an input port. It happens that they can be used together. In the CAL language, if the guard condition is not satisfied, the action does not run. Nonetheless in this thesis work, the action runs and checks the condition. Should the condition is not satisfied then returns. Otherwise the run continues.

In the CAL language a simple action, which uses a guard condition, may look like:

```text
bool preferRead := true;
select.read.prefer: action RA:[a] =>
guard preferRead
  do
    address := a;
    burstSize := BURSTSIZE;
    preferRead := false;
  end
```

**Figure 5.1 – Usage of guard expressions in CAL**

As seen, the preferRead is a local variable used in the guard expression. If the variable is true then the action is executed, whether it is false then the action is not executed. The same action in ajava looks like:

```java
int select_read_prefer(int position, InputStream<Integer> RA){
  if(!preferRead)
    return position;
  address = RA.readInt();
  burstSize = BURSTSIZE;
  preferRead = false;
  return 2;
}
```

**Figure 5.2 – Usage of guard expressions in ajava**
If the preferRead is false then the action returns/ends before running the main action body. The position variable and the return value are used for scheduling purposes and will be explained in further chapters. More complex guard example which uses the input value together with local variables can be seen in figure 5.3 (in the CAL language).

```
start: action BTYPE: [ cmd ] ==> START: [ -2 ]
guard
  bitand( cmd, NEWVOP ) != 0
  do
    ...
  end
```

**Figure 5.3** - Guard expression using input and local variable

The guard uses both the input from port BTYPE and a variable NEWVOP in the expression. In the CAL language, unless the condition is satisfied the input is read but not consumed. However in the ambric architecture, reading the input without consuming, is not possible. Thus it has to be done on the software level. In this approach the input is read and if the guard condition is not satisfied, which means the input shall not be consumed, the input is saved in a variable and a flag is set. The variable is called undo_var and the flag is called undo_flag. Each time before reading the input port the undo_flag is checked. In case if the flag is set, instead of reading the input port, undo_var is read. Unless the condition is satisfied again, the undo_flag remains unmodified. Providing the condition is satisfied then the undo_flag is set to false which means the next time the input port will be read.

When start action is converted to java, it looks like:

```java
int start(int position, InputStream<Integer> BTYPE,
          OutputStream<Integer> START){
    int cmd;
    if(undo_flag){
        cmd = undo_var;
        undo_flag = false;
    } else
        cmd = BTYPE.readInt();
    if(!((cmd & NEWVOP) != 0)){
        undo_flag = true;
        undo_var = cmd;
        return position;
    }
    ...
    //the body of the action
}
```

**Figure 5.4** – Start action, given in figure 5.3, in java
The first *if* condition is used to check whether the input port will be read or there is already a read but not consumed input saved in the undo_var variable. The second *if* condition is the guard condition from the original CAL action.

### 5.2 Scheduling and Priorities

As mentioned in CAL description chapters, in the CAL language a schedule table can be defined for actions. Additionally, the actions may have priorities. However in ambric architecture there are neither scheduling nor priority definitions. If the schedule table and the priority definitions are very simple, the body of actions can be converted directly into the run method in the ajava, with the sequence depending on the schedule table and priority definitions. (In the aDesigner, the run method is the default method for java binding and it is executed repeatedly.) From the other hand, providing the schedule table and the priority definitions are complex, all actions are converted individually into java methods. Afterwards, a while loop is implemented in the run method in order to run the methods in the sequence defined in the schedule table and priority definitions. The run method becomes a scheduler.

When the run method is used as a scheduler, each java method gets a position number depending on the states of the schedule table and priority definitions. The methods on the same priority group get the same position number. When the run method executes, the first method (action) on the schedule table runs. The position of the method is sent as an argument while calling for the method. If there is any guard condition and if it is not satisfied, the method returns self position which means it did not run. Considering there is any method which has a lower priority then it is called. This operation goes on until a method has a satisfied guard condition or the method with lowest priority is called. The method with the lowest priority cannot have any guard condition because unless the guard condition is satisfied, the next action (method) which is called is undefined. Scheduling works when the actions execute. Provided they have guard conditions the priorities are used. They are used in order to define the execution sequence of the actions when the guard conditions are not satisfied. In this approach, considering the guard condition is satisfied, the method runs the converted action body and at the end of the execution returns the position value of the next method, which will be executed, back to the run method (scheduler). The scheduler checks the position value and runs the method which has the same position value and the highest priority. A schedule table and priority definitions in the CAL language are given below:

```
schedule fsm read :
read        ( start       ) --> getw;
read        ( read.inter_ac ) --> inter;
read        ( read.intra    ) --> intra;
read        ( read.other    ) --> advance;
getw        ( skip         ) --> geth;
geth        ( skip         ) --> read;
intra       ( getdc.intra   ) --> sat;
inter       ( getdc.inter   ) --> sat;
sat         ( sat          ) --> advance;
```
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

```
   advance (advance) --> read;
end

priority
   start > read;
   read.inter_ac > read.other > read.intra;
end
```

**Figure 5.5** – Schedule and priority tables in CAL

The schedule table is divided into 3 columns: the left one contains the labels and states, the middle one contains the action names and the right one contains the schedule labels as well. Moreover, the right column shows which label is the next one in the execution sequence after the action on the left side of the arrow. In the above example, `read` is a higher level label to group `inter_ac`, `other` and `intra` actions together. Since `start` and `read` actions are in the same group of priority, they get the same position number in the ajava. The converted version can be seen below:

```java
int position = 1;
    while(true){
        if(position == 1){
            position = start(position, BTYPE, START);
            if(position == 1)
                position = read_inter_ac(position, BTYPE, START, PTR, SIGNED, QUANT);
            if(position == 1)
                position = read_other(position, BTYPE, START);
            if(position == 1)
                position = read_intra(position, BTYPE, A, B, C, PTR, START, SIGNED, QUANT);
        }
        if(position == 2){
            position = skip(position, BTYPE);
        }
        if(position == 3){
            position = skip(position, BTYPE);
        }
        if(position == 4){
            position = getdc_intra(position, IN);
        }
        if(position == 5){
            position = getdc_inter(position, IN);
        }
        if(position == 6){
            position = sat(position, OUT);
        }
        if(position == 7){
            position = advance(position);
        }
    }
}
```

**Figure 5.6** – Scheduler in ajava
Along with the positions, the input and output ports are passed to the methods as arguments.

At first start action runs and checks the guard condition. If the guard condition is not satisfied start action returns own position, which is the same as read.inter_ac action, immediately and the read.inter_ac action starts to run and this operation goes on as previously explained. If the guard condition in the start action is satisfied, instead of returning own position, it returns ‘2’ which is the position value of skip action. Then the scheduler reads the return value and runs the skip action.

In the following example there is another situation and this situation will be called as ‘different situation’ in the rest of this report;

```java
while (true) {
    if (position == 1) {
        position = tc(position);
        if (position == 1)
            position = data_out(position, DI, DO);
        if (position == 3)
            position = data_inp(DI);
    }
    if (position == 2) {
        position = addr(AI, AO);
    }
}
```

**Figure 5.8** – Solution of a ‘different situation’ in java
When the schedule depends on the availability of inputs then the triggers and tags must be used in astruct and ajava. By using the triggers, the correct action can be triggered depending on the input. More information about triggers and tags can be found in [1].

5.3 Structure (Network to Composite Object)

Building a design or a network of processors and rams is similar in CAL language and astruct. In both languages, the objects/actors are instantiated and interconnected on the same level/file. When a number of objects/actors are connected via channels, they are called composite objects in the ambric architecture and networks in the CAL language. In the astruct language, the syntax of instantiation of the objects is the same as the one in java language. Nevertheless in the CAL language it differs. An example can be seen below:

In the CAL:
```java
searchwin = SearchWindow();
```
In the astruct:
```java
SearchWindow searchwindow;
```
In both languages, parameters can be used while defining an object such as:

In the CAL:
```java
searchwin = SearchWindow(SEARCHWIN_IN_MB = 5);
```
In the astruct:
```java
SearchWindow searchwindow = {SEARCHWIN_IN_MB = 5};
```

In the CAL, the actors are instantiated under entities label and the interconnections are defined under structure label. In the astruct input and output ports of objects or designs are defined in interfaces and the objects are instantiated along with the definitions of the interconnections in the binding methods. An example of a network in the CAL language can be seen below:

```java
network motion3 () MV, BTYPE, TEX, MCD ==> MBD, MBA, MCA, VID:
var
FLAG_SZ = 4;
entities
memorymanager = MemoryManager(FLAG_SZ = FLAG_SZ);
mbpack = MBPacker();
searchwin = SearchWindow(FLAG_SZ = FLAG_SZ);
unpack = Unpack();
interpolate = Interpolate();
add = Add();
structure
MV --> searchwin.MV;
MV --> unpack.MV;
BTYPE --> memorymanager.BTYPE;
BTYPE --> searchwin.BTYPE;
BTYPE --> unpack.BTYPE;
BTYPE --> add.BTYPE;
TEX --> add.TEX;
memorymanager.WA --> mbpack.AI;
```
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

memorymanager.RA --> MCA;
mbpack.DO --> MBD;
mbpack.AO --> MBA;
MCD --> searchwin.DI;

searchwin.DO --> unpack.DI;
searchwin.FLAGS --> interpolate.halfpel;

unpack.DO --> interpolate.RD;
interpolate.MOT --> add.MOT;

add.VID --> mbpack.DI;
add.VID --> VID;

end

Figure 5.9 – A CAL network sample

The same above network is a composite object in ambric architecture and it is defined as below:

interface Motion3 {
    inbound BTYPE;
    inbound MCD;
    inbound MV;
    inbound TEX;

    outbound MBA;
    outbound MBD;
    outbound MCA;
    outbound VID;
}

binding JMOTION3 implements Motion3{
    MemoryManager memorymanager;
    SearchWindow searchwindow;
    SWBuffer swbuffer;
    Unpack unpack;
    Interpolate interpolate;
    Add add;
    MBPacker mbpacker;
    Splitter4 splitter4;
    Splitter2 splitter2;

    channel ch0 = {BTYPE, splitter4.in},
    ch1 = {splitter4.out_1, memorymanager.BTYPE},
    ch2 = {memorymanager.RA, MCA},
    ch3 = {memorymanager.WA, mbpacker.AI},
    ch4 = {splitter4.out_2, add.BTYPE},
    ch5 = {TEX, add.TEX},
    ch6 = {interpolate.MOT, add.MOT},
    ch7 = {splitter4.out_3, unpack.BTYPE},
    ch8 = {searchwindow.DO, unpack.DI},
    ch9 = {MV, splitter2.in},
    ch10 = {splitter2.out_1, unpack.MV},
    ch11 = {unpack.DO, interpolate.RD},
    }
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

ch12 = {splitter2.out_2, searchwindow.MV},
ch13 = {splitter4.out_4, searchwindow.BTYPE},
ch14 = {MCD, searchwindow.DI},
ch15 = {searchwindow.FLAGS, interpolate.HALFPEL},
ch16 = {add.VID1, mbpacker.DI},
ch17 = {add.VID2, VID},
ch18 = {mbpacker.AO, MBA},
ch19 = {mbpacker.DO, MBD},
ch20 = {searchwindow.buffer_addr, swbuffer.address},
ch21 = {searchwindow.buffer_data, swbuffer.data_in},
ch22 = {swbuffer.data_out, searchwindow.buffer_in};

Figure 5.10 – Conversion of the CAL network in figure 5.9 to a composite object

A drawback of the ambric architecture is; the number of input and output channels of a leaf object is bounded. Therefore in some points when the input or output ports of the actors are too many, splitters and combiners must be used while converting to the astruct. There are primitive objects defined in astruct libraries which can be used as splitters and they are called DupForks. DupForks have one input and two output channels and they are mapped directly to the internal hardware forks on the chip, which copy the input to both outputs in less than one cycle (they are on the channels between processors, essentially taking zero cycles). When a CAL actor has an output port which is connect to several different input ports, then the DupForks must be used to split the same data to several input ports. Due to the fact in the ambric architecture the same channel cannot be connected to several ports. Additionally in some cases, CAL actors may have unnecessarily high number of input and output ports which can be combined together. Such combinations have been done during this thesis approach.

The Xilinx model of the MPEG-4 SP decoder has four main blocks which are called as parser, acdc, idc2td and motion. This structure has been kept while converting the CAL language codes to the astruct language. In the CAL implementation, parser block has 5 actors, acdc block has 7 actors and idct2d block has 12 actors, motion block has 6 actors. There are several actors in order to display the output however only DDRModel actor has been converted. The output is not displayed. In total 31 actors are used and converted so far. In the ambric implementation, parser block has 11 processors, acdc block has 7 processors, idct2d block has 12 processors and motion block has 7 processors. With the DDRModel processor, the total number of processors used on ambric chip is 38. The parser block which includes 11 processors has been implemented in [20] and then combined with the other blocks which are implemented during this thesis work.

5.4 Memory

On both structures, buffers can be used on the channels between objects. However the memory is limited on the ambric chip. Providing the required memory is larger than the memory on the chip, the external DDR2 memory can be used via DDR2 controllers. There are two DDR2 controllers and each controller has four ports which can be shared among different objects. To be able to use the DDR2 controllers, DDR libraries must be used. The controllers are instantiated in the same binding where the objects, which use them, are instantiated. Port number 0 must be
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

used as default because it is used for configuration of the controller. For more information about the DDR2 controller, reader can refer to [1].

5.5 Variables

In the CAL language, the variables may have different sizes however in the ambric architecture they have fixed sizes. Floating points and division operation are not supported in the ambric architecture. Additionally in the CAL language, boolean values are transferred through channels which is not possible in the ambric architecture. Only integer values can be used on the channels. Therefore, boolean true and false values of the CAL design are represented as ones and zeros respectively in the ambric design. The variables of CAL design which are computed by using math libraries are defined statically in our ambric design.
6 Implementations

In this chapter, detailed information about the objects and designs, which have been implemented in this thesis work, will be given.

The CAL implementation of the MPEG-4 decoder consists of four main blocks and one additional actor. These blocks are called as parser, acdc, idct2d and motion moreover, they have been converted into composite objects. Each block has a number of actors and these actors have been converted as leaf objects.

The parser block has been implemented in [20], thus no information about the parser will be given.

The outline of the decoder is given below:

![MPEG-4 Decoder Diagram]

**Figure 6.1** – The overview of the MPEG-4 decoder implemented in this thesis work
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

In some of the actors the input ports are checked for the input availability. Only BTYPE and START ports are checked and in an actor there can be either one of them. In the actors, which do have neither BTYPE nor START ports, there is no check for input availability. Since at most one input port is checked, only one undo variable and one undo flag are used in the actors which have the input availability check.

The CAL implementation of the decoder has the following graphical layout (from the Graphiti Editor)

![Graphical layout of the MPEG-4 decoder](image)

**Figure 6.2** – The blocks of the MPEG-4 decoder

In the ambric implementation each block is mapped to a composite object. Several splitters are used between the composite objects due to the restrictions of the ambric architecture.

Each actor is described in the paper regarding three notions - number of actions, the matter of conversion of the action bodies and the schedule implementation.

### 6.1 ACDC Block

As given in figure 6.3, the ACDC block consist of 7 actors; DCSplit, Sequence, DCPred, ZZAddr, ZigZag, ACPred, Dequant. The names of the actors give an idea about what is done within them. No further details will be given about the algorithms used in the actors. However, the implementation/conversion of the actors will be explained with details.

The overview of the block (from the Graphiti Editor) is given in figure 6.3:
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

6.1.1 DCSplit

This actor consists of 2 actions; dc and ac. The run sequences of these actions are defined by using priorities. dc action has a higher priority and a guard condition, if the guard condition is not satisfied the ac action runs. The action bodies have been converted directly to ajava and run sequence is arranged by using an if condition based on the guard condition of the actions. No position value is used. The if condition can be seen below:

```java
if (count == 0)
    dc(IN, DC);
else
    ac(IN, AC);
```

![Figure 6.3 – ACDC block of the MPEG-4 decoder](image)

6.1.2 Sequence

The sequence actor is composed of 11 actions. 5 of these actions are grouped under a high level label. This actor includes a schedule and action priorities. Since the scheduling is complex, the method defined under “Scheduling and Priorities” has been used for conversion with a small difference. The actions, which are in the same group, use a common variable for the guard conditions. In ajava, instead of using the position values of the actions, the same variable, which is used in CAL language, is used within the if conditions to define the sequence of these actions. The run sequence definition of the actions is shown below:

```java
if (comp == 0)
    position = predict_b0(position, A, B, C);
else if (comp == 1)
    position = predict_b1(position, A, B, C);
else if (comp == 2)
```

![Figure 6.4 – If condition used in DCSplit](image)
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

```plaintext
position = predict_b2(position, A, B, C);
else if (comp == 3)
  position = predict_b3(position, A, B, C);
else
  position = predict_b45(position, A, B, C);
```

**Figure 6.5** – The definition of run sequence of the actions in the Sequence actor

The actions still return a position value in order to define which action will run as next.

### 6.1.3 DCPred

This actor consists of 9 actions and has schedule and action priorities. The action bodies are converted without any modification and the schedule is implemented by using the usual method from “Scheduling and Priorities”. However, the division operation is used in an action. Since this operation is not supported in ajava, previously implemented fixed point division method is used.

This actor has 5 output ports however one of the output ports is connected to 3 different actors, which is not possible in the ambric architecture. Thus, this output port is connected to a splitter object which uses 2 DupFork[1] objects in the origin and splits the input to 3 different outputs.

In the actions, there are no guard conditions, which means the availability of the input is never controlled. Therefore undo variable and undo flag are not needed in the ajava.

### 6.1.4 ZigZagAddr

The ZigZagAddr consists of four actions and similarly to the previous case- it posses schedule and the action priorities as well. The action bodies are converted without any modification. The schedule is implemented by using the methods from “Scheduling and Priorities”. There exists a ‘different situation’ as defined in “Scheduling and Priorities”.

### 6.1.5 ZigZag

This actor is composed of 6 actions. This actor has schedule and action priorities and the action bodies are converted without any modification. The schedule is implemented by using the methods from “Scheduling and Priorities” as in previously described actor. There are 2 ‘different situation’ points and these points are caused by the same actions due to being in 2 different states of the schedule. The next actions in the schedule vary depending on the state of the schedule. Therefore, while returning a position value, the current state (position value) is checked. This applies only to the actions which appear on the different states.

### 6.1.6 ACPred

The ACPred actor has 5 actions, schedule and action priorities. The action bodies are converted without any modification except one action which will be mentioned in the next paragraph. The schedule is implemented by using the methods from “Scheduling and Priorities”.

59
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

This actor has a list which consists of 15744 integer values. The list is converted to an array of integers in Java. However, 15744 words mean 62976 bytes which exceeds the memory size of a processor in the ambric architecture. (An SRD processor has 1kb of internal memory and can use 4 ram units with total memory size of 8kb.) Since the memory requirement was too high, the array is kept in the external DDR2 memory. The action which uses the array accesses the external memory via TMS packets. Since the memory is off chip, the access takes longer time than a usual internal memory access.

Writing to the external memory:

```java
  toMemory.writeDE1(WRITE_HDR | (4 << 16) | addr*4);
  toMemory.writeDE0(pred);
```

Reading from the external memory:

```java
  toMemory.writeDE0(READ_HDR | (4 << 16) | addr*4);
  pred = pred + fromMemory.readInt();
```

6.1.7 Dequant

This actor is a small one consists of 3 actions. Moreover, it has schedule and action priorities. The action bodies are converted without any modification. The schedule is implemented by using the methods from “Scheduling and Priorities”.

6.2 IDCT2D Block

The IDCT2D block consists of 7 actors and a block which consist of 5 actors. Therefore, in total there are 12 actors in the IDCT2D block. An overview of this block from the Graphiti Editor is given in figure 6.6.

As seen in the figure 6.6, there are many channels between the actors. After the conversion, the channels of the same groups are combined into one channel and the unnecessary channels are removed. The new overview of the block (from the aDesigner) is given in figure 6.7.
The number of channels in the entire IDCT2D block is reduced from 37 to 17.

6.2.1 RowSort

This actor is built up with 10 actions. It includes schedule and action priorities as well. The actions are converted without any modification. However two actions are similar and they are the only ones which have priorities. In order to avoid the priorities and keep the schedule simple, these two actions are combined. The schedule becomes simple without the priorities and the actions run in a simple sequence. Since this schedule became very simple, in the run method, the actions are called sequentially; nothing spectacular is implemented for scheduling, such as in figure 6.8.

In the actions, there are no guard conditions, which means the availability of the input is never controlled. Therefore undo variable and undo flag are not needed in the ajava.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

```
a0(ROW);
a1(ROW);
a2(ROW);
a3(ROW);
a4(ROW, Y);
a5(ROW);
a6(ROW, Y);
a7(ROW, Y);
a8(Y);
```

**Figure 6.8** – Conversion of RowSort actor

### 6.2.2 FairMerge

This actor consists of 4 actions and has schedule and action priorities. There are 2 main actions (row and col) and the other two are copies of the main actions with lower priorities. Each main action is supposed to run 4 times in a row, however at the first run, if the required input type is not available, the copy of the other main action is executed. In the end, the other main action runs for 4 times in a row. Since the type of in the input influences the execution of the actions, triggers and tags are needed. Nonetheless, the execution sequence of the actions is fixed and known, thus the copy actions are not implemented and the main actions are executed in the known sequence without using any schedule table. The execution sequence definition is very complex in the CAL implementation; however after the conversion it is very simple. The definition of execution sequence is given below:

```
for(int i=0; i<8; i++)
    row(R, ROWOUT1, ROWOUT2, Y);
for(int i=0; i<8; i++)
    col(C, ROWOUT1, ROWOUT2, Y);
```

**Figure 6.9** – Definition of execution sequence of actions in the FairMerge

The actions only read the inputs and forward them to the output ports. Position variable, undo variable and undo flag are not needed.

In the CAL implementation, one of the output ports is used to send boolean values, however in the ambric architecture only integers can be send via channels. Thus, instead of boolean values true and false, 1 and 0 is used respectively.

### 6.2.3 DownSample

This actor is very small, consists of 2 actions and has schedule. One may notice that the only thing that is done by this actor is reading the input 2 times and writing only the second input to the output port. In the CAL implementation the first action reads the input and the second action gets called. The second action reads the input and writes it to the output port. For this simple task no schedule is necessary in aJava. In the run method the following codes are executed:
6.2.4 Separate

This actor is composed of 2 actions. There is neither schedule nor priorities. The execution sequence is defined by guard conditions which check the input value of an input port. In the CAL implementation the value is either true or false and on the ambric implementation it is either 1 or 0. Before conversion this actor has 8 output ports however after the conversion and the combination of the channels of the same group, it has only 2 output ports and which causes no problems.

The bodies of the actions are combined in the run method and with an if condition the sequence is defined.

6.2.5 Transpose

This actor consists of 3 actions. There is neither schedule nor priorities. Each action has a guard condition and all guard conditions use the same variable. Therefore no schedule was needed. Since the execution sequence depends on a variable, it is easy to find it.

The execution sequence is fixed and known. Thus instead of using the run method as a scheduler, the action bodies are combined in the run method and for loops are used in order to define the execution sequence of the action bodies.

6.2.6 ReTranspose

This actor is similar to the ‘Transpose’ actor. The number of the actions, execution sequences and the conversion methods are the same.

6.2.7 Clip

The clip actor consists of 5 actions. 4 of these actions are grouped under the same label. All actions have priorities and no schedule is defined. The execution sequence is defined by the priorities and guard conditions. Instead of defining the run method as a scheduler, the action bodies are combined in the run method and if conditions are used in order to define the execution sequence.

This actor has 2 input ports and one of them is used for boolean values. However in the ambric implementation only integers are used as mentioned earlier.
6.2.8 IDCT1D Block

The IDCT1D block is inside the IDCT2D block and consists of 5 actors. The overview of this block from the Gprahiti Editor is given below:

![Figure 6.10 – The IDCT1D block of the MPEG-4 decoder](image)

The overview of this block after conversion is given below:

![Figure 6.11 – The IDCT1D block after the conversion](image)

6.2.8.1 Scale

This actor consists of 1 action and has 2 input ports and 4 output ports. The body of this action is converted directly into the run method. The input ports are reduced to 1 port and the output ports are reduced to 1 port as well. All output values are sent in a defined order on the same channel and on the other end the inputs are read in the same order.

6.2.8.2 Combine

This actor is composed of 2 actions; row and col. There is no schedule but priority. row action has higher priority and a guard condition. The guard checks the input value of one of the input...
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

ports. Either row or col is executed depending on the value of the input. Therefore in the ajava, bodies of the actions are combined in the run method and with and if condition one of them is executed depending on the input value. In the CAL implementation the input value is either true or false and in the ambric implementation it is either 1 or 0.

This actor’s input ports are connected to the output ports of the Scale actor. The number of input ports is decreased from 4 to 1 as mentioned previously. Additionally, the number of the output ports is decreased from 2 to 1.

6.2.8.3 ShuffleFly

This actor consists of 2 actions and has a schedule which is very simple; the actions run sequentially. There are 2 input port and 4 output ports. The first action reads one input from each input port and the second action reads one input from each input port and writes one output to each output port. In the ajava there is no need to implement two actions. The bodies of the actions are converted into the run method sequentially.

This actor’s input ports are connected to the output ports of the Combine actor. The number of input ports is decreased from 2 to 1. Additionally, the number of the output ports is decreased from 4 to 1.

6.2.8.4 Shuffle

Three actions and schedule compose the shuffle actor. The schedule of this actor is based on the simple run sequence of the actions. Thus there was no need to implement any schedule in ajava. The bodies of the actions are converted sequentially into the run method.

The task of the actor is given in the name of the actor. The inputs are shuffled and forwarded to the output port. The numbers of input and output ports are both decreased from 4 to 1.

6.2.8.5 Final

This actor consists of 1 action and has 4 input and 4 output ports. The action body is converted into the run method. The numbers of the ports are decreased from 4 to 1.

6.3 Motion Block

The Motion block is the last block of the decoder and consists of 6 actors. As given in the name, motion compensation is performed in this block. An overview of this block is given below:
In the ambric implementation there are 7 objects. One of the actors is divided into two objects due to the lack of memory.

### 6.3.1 MemoryManager

This actor consists of 9 actions along with the schedule and action priorities. 3 of the actions are grouped under a label and 4 of the other actions are grouped under another label. In the ajava, the schedule is implemented with the methods given under “Scheduling and Priorities”.

This actor is connected to the DDRModel actor which needs to be triggered. Thus 3 actions, which send outputs to the DDRModel, use tags. Before sending data, these actions write a pre-defined tag to the channel connected to the DDRModel in order to trigger an action in the DDRModel. An example is given below:

```java
RA.writeTag(RA.intOfTag("readTag"));
RA.writeInt(address(last_frame, this_mby, this_mbx ));
```

In the CAL implementation, complex mathematical operations are used (such as log2_ceil) in order to appoint values to some variables. Since such operations are not supported in the ajava, the values of the variables are appointed as constants.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

6.3.2 SearchWindow

This actor consists of 18 actions. Some of these actions are grouped under different labels. In the ajava the schedule is implemented by using the methods given under “Scheduling and Priorities”.

The SearchWindow actor does not fit into an SRD processor due to insufficient memory. There are two arrays consist of 512 words and 1024 words respectively. When combined with the other variables and instruction data, the entire data does not fit into 4 memory banks (each memory bank has capacity of 512 words, the SRD processor has capacity of 256 words) plus the internal memory of SRD processor. The size of the instructions is higher than 256 words. Thus one memory bank is used for instruction data and 3 memory banks are left. Two arrays with total size of 1536 words fill 3 memory banks and thus, no free space remains for the other variables. The number of the variables is highly decreased however there still exist several variables which cannot be removed. Therefore, the array, which consists of 1024 words, is placed in another SRD processor. This processor is used as a buffer and connected with two channels to the main processor. When an action needs to reach this array, at first it sends a tag to specify whether a read or a write operation will be performed. Furthermore the index of the array is sent to the processor, which contains the array. On the other side, a read or a write action gets triggered. If the write action gets triggered, the first input is interpreted as the index and the next one as the data. If the read action gets triggered, the data at the given index of the array is returned.

In two points there are ‘different situations’.

6.3.3 Unpack

The unpack actor consists of 11 actions and has schedule and action priorities. There are two groups of actions consist of 2 and 4 actions respectively. The actions are converted without any modification. In the ajava the schedule is implemented by using the methods given under “Scheduling and Priorities”.

6.3.4 Interpolate

As an actor, Interpolate is composed of 4 actions and has schedule and action priorities. The actions are converted without any modification. In the ajava the schedule is implemented by using the methods given under “Scheduling and Priorities”. One ‘different situation’ exists in this actor.

6.3.5 Add

Eight actions build up this actor along with schedule and action priorities. Similarly to the previous actor, the actions are converted without modification. In the ajava the schedule is implemented by using the methods given under “Scheduling and Priorities”.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

There are 3 input ports and one output port in this actor. The output port is connected to 2 different actors. Thus in the ambric implementation 2 output ports are used, one for each actor. The same output gets written to both of the output ports.

6.3.6 MBPacker

MBPacker actor consists of 4 actions and has schedule and action priorities. The actions are converted without any modification. In the ajava the schedule is implemented by using the methods given under “Scheduling and Priorities”. There is one 'different situation'.

This actor is connected to the DDRModel which needs to be triggered. Therefore the action, which sends data to the DDRModel actor, sends the tag as first in order to trigger the actor. Furthermore the data is sent.

6.4 DDRModel

This actor consists of 7 actions and contrary to the previous cases, it has very complex schedule and action priorities. The reason of the complexity is the priorities and the guard conditions. There are 2 main actions; read and write. The other actions are implemented in order to define the execution sequence depending on a local variable and input availability. Only these two main actions are implemented via ajava. The input availability check is fixed via tags and triggers. Since there are only two actions, the local variable is not used. Either read or write action is triggered depending on the tag value. The trigger definition is given below:

\[
\text{trigger } t = (\text{read}\{\text{RA:readTag}\} \mid \text{write}\{\text{WA:writeTag}\})*;
\]

In the origin, this actor in fact has a timing-dependent schedule (it depends on the availability/arrival time of inputs). This makes blocking reads inappropriate. The background is that the model was developed as part of an FPGA project at Xilinx. The DDRModel actor models a DDR memory controller that was included as a separate IP in the final design. It is thus neither "software friendly" nor a reasonable way of implementing the frame buffer in software.

DDRModel (and the actual memory controller) works in read and write bursts of 96x4=384 words. This happens to be the size of a macroblock in YUV format. It toggles between the states in which it prefers a read (write) burst in an attempt to balance the two kinds of bursts fairly. When preferring a read burst it carries out such a burst if input (on RA, read address port) is available and considers a write burst otherwise (vice versa). To carry out a write burst uninterrupted, 96x4 words must be present at the WD (write data) port and an address must be present at WA port. Therefore the channel, which is connected to the WD port, must have a buffer of at least 96 words in order not to block the sender actor.

In the ajava implementation, RA (read address) and WA (write address) ports are checked for the tags. RA channel triggers the read action and WA channel triggers the write action. Therefore the execution sequence of the actions depends on the timings of the inputs. The rest is the same as in the CAL implementation. The read action reads bursts of 96 words from the memory and the write action writes bursts of 96 to the memory.
Since this actor models a DDR memory controller, it implements an array of 524268 words. This size is very large for an SRD processor and 4 memory banks. Therefore the external DDR memory is used. Reading and writing the bursts of 96 words are performed on the external DDR memory.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

7 Performance Analysis & Comparison

In this thesis work, in order to analyze the performance of the design, perfHarness library, which is provided by the ambric department of Nethra Imaging, is used. The reader should keep in mind that this library is still under development. The performance results of the ambric chip are confirmed by the technical lead of the software tools team of the company. For comparison purposes, the performance results of the CAL design has been provided by Ericsson.

The problem of using the PerfHarness library has been explained in the previous chapters as well as the matter of dividing the input into workloads.

The performance analysis result may have variation. Whereas some portions of the design are beginning to run even as the chip configuration process is completing. This is one of the interesting side effects of using asynchronous channel communications on the Am2045 chip - one end of the chip is up and running part of the design while the other is completing the configuration. Any attempt to write or read data on channels, which are still under configuration, simply gets blocked. The first performance result variation is therefore related to the variations in the host/ambric card communications during configuration. Since the input could not be divided into workloads, first performance result needed to be used. In order to reduce the configuration-dependent time a delay object is placed in front of the PerfHarness library objects. Thus the input can be delayed until the configuration is finished on entire chip. However there are still small variations on the results which are related to the configuration time. These variations are small enough to be ignored.

If the same design configuration file is re-loaded, it will place identical contents on all processors, memories, etc on the chip - there is no variation there, in order to provide reliable run-to-run results and debug support. The perfHarness measurements on subsequent data input workloads are very consistent (when identical or similar data sets are used; encoders and decoders can have variable frame-by-frame performance based on the ability of the encoder to locate data it can compress via motion vectors, for example).

7.1 Performance Results

For performance measurements two different input streams have been used. First stream has 300 frames with the resolution of 176x144 (QCIF) and the second stream has 384 frames with the resolution of 720x480. Additionally, 38 SRD processors are used for the implementation of Xilinx model of MPEG-4 SP decoder. 11 of these processors are used in parser block and implemented in [20]. For the performance analysis, the parser block has been combined with the blocks implemented in this thesis work. Therefore the results are of an entire decoder of MPEG-4 simple profile.

The performance analysis library provides the cycles counts therefore the cycles counts will be given along with the time in seconds and frames per second (fps) value.

Since the ambric chip runs at 300 Mhz, the numbers of seconds has been calculated by dividing the cycle counts to 300,000,000.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

The performance is analyzed by using 20 individual runs.

The results for the first bit stream (300 frames – 176x144 resolutions) are given in the following table:

<table>
<thead>
<tr>
<th></th>
<th>Cycle count</th>
<th>Run time in seconds</th>
<th>Frames per second (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>166678284</td>
<td>0,55659428</td>
<td>539,96236246</td>
</tr>
<tr>
<td>Best</td>
<td>165581028</td>
<td>0,55193676</td>
<td>543,54053170</td>
</tr>
<tr>
<td>Average</td>
<td>166170375</td>
<td>0,55390125</td>
<td>541,612787477</td>
</tr>
</tbody>
</table>

Table 7.1 Performance results of the first bit stream (176x144 - 300 frames)

The variation of the results is because of the configuration-dependent time. Therefore, it is reasonable to use the best result as the main result of the analysis which is 543,54 fps. Additionally the difference between the best and worst results is around 0.3%. For the second bit stream, the best result will be used as the main result as well.

The measurements are done by using the performance harness library which is provided by Nethra Imaging (the owner company of Ambric chips). Since the library is still under development, it does not work perfect with all kinds of designs. Thus, even though the library is modified two times, there is still a very small configuration-dependent time within the results.

The performance results for the second bit stream (384 frames - 720x480 resolutions) are given in the following table:

<table>
<thead>
<tr>
<th></th>
<th>Cycle count</th>
<th>Run time in seconds</th>
<th>Frames per second (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td>391357316</td>
<td>1,30452438</td>
<td>294,36015599</td>
</tr>
<tr>
<td>Best</td>
<td>357713712</td>
<td>1,19237904</td>
<td>322,04524494</td>
</tr>
<tr>
<td>Average</td>
<td>370964576</td>
<td>1,23654858</td>
<td>310,54178235</td>
</tr>
</tbody>
</table>

Table 7.2 Performance results of the second bit stream (720x480 - 384 frames)

After 20 runs, the best result for the second stream is 322 fps.

The decoder implemented with CAL language is directly converted to the ajava and astruct languages. No optimization is done. If assembly was used instead of ajava, the performance results would be better. Moreover if further optimizations were done, the performance could be enhanced one more time.

7.2 Comparison

In this chapter, at first the performance results of the decoders written in ajava/astruct and CAL languages are compared. Furthermore, the performance result of the decoder on the ambric chip
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

is analyzed with two different video streams. As mentioned in the previous chapters, the decoder is the Xilinx model of the MPEG-4 simple profile decoder.

7.2.1 Ambric vs Intel Processor using the CAL implementation

In the CAL implementation in total 31 actors are used for the performance analysis. The CAL implementation has been executed on a 2 GHz general purpose, Intel Core i7 processor. It is a quadcore however the decoder is executed single-core threaded on CPU0. On the other hand, 38 SRD processors are used for the ambric implementation. Additionally, the ambric chip runs at the clock speed of max 300 MHz.

The CAL implementation is compiled with CAL Actors tools however it is out of scope of this thesis work, thus no details will be given about CAL Actors tools.

The comparison parameters are the frame rate, clock cycles, run times, number of processors, processor speed. For the first video stream (176x144), the comparison between intel i7 processor (using CAL implementation) and ambric is given in the table 7.4.

<table>
<thead>
<tr>
<th></th>
<th>Ambric (ajava &amp; astruct)</th>
<th>CAL Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>Clock speed</td>
<td>300 MHz</td>
<td>2 GHz</td>
</tr>
<tr>
<td>Cycle counts</td>
<td>165581028</td>
<td>4801287271</td>
</tr>
<tr>
<td>Run-time (seconds)</td>
<td>0.5519</td>
<td>2.4067</td>
</tr>
<tr>
<td>Frame rate</td>
<td>543.5405317 fps</td>
<td>124.652013 fps</td>
</tr>
</tbody>
</table>

Table 7.4 The comparison of the decoders written in ajava & astruct and CAL languages with 300 frames of resolution 176x144

The ambric chip uses 38 processors running at 300 MHz and decodes the first input stream 4.46 times faster than a 2 GHz intel i7 processor using the CAL implementation. For the ambric implementation, the program code of the entire decoder is written in java and no optimization has been applied. The run-time can be decreased by using assembly and applying further optimizations, such as dividing an object into few processors which works slower than the rest of the decoder.

For the second video stream, which has 384 frames with the resolution of 720x480, the comparison between intel processor (using the CAL implementation) and ambric is given in the table 7.5.
The ambric chip decodes the second input stream 32,76 times faster than the 2 GHz intel i7 processor using the CAL implementation. It obviously shows that the ambric implementation shows better performance with larger input streams when the CAL implementation on a single processor shows the same performance. There are several reasons for this performance difference such as the structure of the decoder or the characteristic of the input stream. These reasons will be explained with details in the next sub-chapter (7.2.2).

<table>
<thead>
<tr>
<th></th>
<th>Ambric (ajava &amp; astruct)</th>
<th>CAL Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>Clock speed</td>
<td>300 Mhz</td>
<td>2 GHz</td>
</tr>
<tr>
<td>Cycle counts</td>
<td>357713712</td>
<td>77924239380</td>
</tr>
<tr>
<td>Run-time (seconds)</td>
<td>1,1924</td>
<td>39,058</td>
</tr>
<tr>
<td>Frame rate</td>
<td>322,04524494 fps</td>
<td>9,831533 fps</td>
</tr>
</tbody>
</table>

Table 7.5 The comparison of the decoders written in ajava & astruct and CAL languages with 384 frames of resolution 720x480

7.2.2 Comparison of the Results for Two Different Input Streams

The comparison parameters are cycle counts, run times and frame rates. The first video stream is 300 frames with the resolution of 176x144 and the second video stream is 384 frames with the resolution of 720x480. The comparison is given in the table 7.3.

<table>
<thead>
<tr>
<th></th>
<th>300 frames – 176x144</th>
<th>384 frames – 720x480</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle counts</td>
<td>165581028</td>
<td>357713712</td>
</tr>
<tr>
<td>Run-time (seconds)</td>
<td>0,55193676</td>
<td>1,19237904</td>
</tr>
<tr>
<td>Frame rate</td>
<td>543,5405317</td>
<td>322,04524494</td>
</tr>
</tbody>
</table>

Table 7.3 The comparison of the decoder on ambric chip using two different video streams
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

When the input streams are compared:

The difference in resolution (720x480 over 176x144) gives a factor of 13.63 in terms of data size (pixel numbers).
The difference in bit rate:
3130121 bytes / 384 frames = 8 kb per frame versus 250342/300 frames = 834 bytes per frame
The ratio of bit rates gives a factor of 9.76.
The difference between frame rates (543.5 fps over 322 fps) gives a factor of 1.69.

The size of the frames differ with a factor of 13.63 however the frame rates differ with a factor of 1.69. The speedup can be calculated by dividing the frame size factor to the frame rate factor (13.63/1.69 = 8). The main reason of the frame rate difference is the characteristics of the input streams. Hence, the results show the reaction of the ambric architecture to different input streams.

With the second input stream, the MPEG decoder shows approximately 8 times better performance. The reason of this performance increase is the structure of the decoder. In some parts of the decoder, the input is buffered. Thus even if there is no input left, some parts of the decoder go on running for a while to consume the buffered inputs. The time to consume the buffered inputs is almost the same regardless of the input stream size. Hence the time for filling the buffers may have large influence on the run-time when the input stream size is small.

The characteristic of the input stream has a significant influence on the performance as well. The first input stream is created by using mostly motion vectors which means a high reduction in the spatial domain is done. However, the second input stream is compressed mostly in frequency domain and while decoding, the idct block is the one which works more than the others. Thus, it can be said that the ambric architecture performs inverse DCT fast and this situation influences the performance of the MPEG decoder.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

8 Conclusions

The goal of this thesis work was to implement an MPEG-4 decoder on the ambric architecture and analyze the performance of the ambric chip in terms of speed and efficiency.

During the thesis work, acdc, idct2d and motion blocks of the Xilinx model of MPEG-4 SP decoder have been converted from CAL actor language to ajava and astruct languages in order to run the decoder on the ambric chip. Furthermore, these blocks have been combined with the parser block which has been developed earlier in [20] in order to complete the decoder design.

8.1 Similarities between CAL and Ambric Languages

During the conversion, each actor on the CAL language has been converted to a leaf object (processor) on the ambric implementation. The networks of the actors on the CAL base have been converted to composite objects on the ambric implementation. In the reality, actors and leaf objects have the same structure and this theory applies to networks and composite objects as well. Shortly, the CAL actors can be mapped to leaf objects and actor networks can be mapped to composite objects in the ambric architecture. In total 26 actors have been converted into 27 leaf objects during this thesis work. One actor has been converted into 2 leaf objects because of lack of memory of an SRD processor. One SRD processor has 1kb internal memory and can use up to 2kb of neighbor memory banks however, the memory was still not sufficient. Four CAL networks have been converted into four composites objects.

Additionally, on both sides, the channels are unidirectional.

8.2 Differences between CAL and Ambric Languages

The major differences between CAL language and ajava are the syntaxes, scheduling tables, action priorities and reading the inputs without consuming them. In the ambric architecture there is no support for scheduling tables and action priorities. However there are triggers and tags in order to trigger any action with pre-defined input values. With the help of triggers and tags the schedule tables and action priorities are possible to implement manually. For some schedule tables and action priorities, triggers and tags are not necessary. Additionally, in the ambric architecture, when an input is read, it is consumed directly. There is no way to check for the input or to read without consuming. However this can also be implemented on the software level and we have already used it in this thesis.

The minor differences between CAL language and ajava are the definition of the variable sizes, support for boolean values on the channels, support for mathematical operations. In ambric architecture the variables has fixed sizes however in the CAL language the sizes are defined by the developer. In the ambric architecture only integers can be sent through channels however in the CAL language boolean values are also sent. In the ambric architecture complicated mathematical operations are not supported such division, square root, ceiling etc... However they can be implemented on the software level such as the division which is implemented in this thesis work.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

8.3 Performance

There are 336 processors on the ambric chip and 168 of these processors are SRD processors. However, only 38 SRD processors have been used for the decoder implementation.

While using a small input stream (300 frames with 176x144 resolution), the ambric chip shows a better performance when compared to the CAL implementation on the single 2 GHz intel i7 processor. (38 processors with 300 MHz clock speed decodes the same input stream 4.46 times faster than a single processor with 2 GHz clock speed.)

While using a larger input streams (384 frames with 720x480 resolution), the ambric chip shows much better performance when compared to the CAL implementation on the single 2 GHz intel i7 processor. The ambric chip shows a performance of 322 fps when the single 2 GHz intel i7 processor decodes only at a rate of 9.83 fps. 322 fps is more than sufficient for any real-time application. The frame rate ratio gives approximately a factor of 32.8.

The decoder on the ambric chip has been tested with the largest 720x480 frame size. However, theoretically with larger frame sizes, the decoder has the capability to achieve the real-time requirements.

When the simulators are used for both CAL and ambric languages, the decoding takes very long time. The time differs when different processors are used for the simulation. The simulation times are not given in this thesis work since they are not presenting the real run-times but the simulations times.

8.4 Development Process & Tools

In order to convert the CAL language to ambric languages, the developer has to know the functionality of the CAL language. (It is assumed that the developer already knows the ambric languages.) It is not necessary to know the algorithm of the code which is under conversion. However, in order to apply optimizations, the algorithm must be known by the developer.

The lack of debug support for the CAL simulator on Eclipse platform is a big obstacle for the developer. Thus the comparison of the CAL results and the ambric results becomes very difficult to perform. The CAL simulator on Eclipse platform may perform some additional functions such as printing to the console or writing to a file. These functions have been used in order to compare the output results of the CAL and ambric languages.

Complex scheduling and priorities may need extra work on the ambric implementations such as using the triggers and tags.

The new performance analysis library for the ambric platform is much better than the previous performance measurement tools. The analysis takes shorter time however the library is not compatible with all design types and input stream sizes. The reason is; the library is still under development.
8.5 Future Work

The future works can be defined as below:

- The entire decoder is written in java. Instead, assembly language can be used (at least for the critical parts of the decoder) in order to gain performance increase.

- The slow parts of the decoder can be found, divided into portions for parallelization and distributed on several processors in order not to slow down the rest of the decoder.

- Some of the actors can be divided into several parts in order to gain more parallelism. For instance the idct blocks can be implemented with more processors. Additionally, opportunities of parallelization can be found in motion block. The key thing here is to sort out the data dependencies and (partial) action schedules, thus independent jobs can be identified. This way, the efficiency of the chip usage can also increase because of using a larger number of processors.

- By using parameters (such as macro block size), dynamism can be added to the decoder and with some further optimizations, the decoder could be used for other profiles.

- Since the ambric chip has enough number of processors, three decoders could be loaded on the chip and run in parallel. Even with some further modifications, three decoders could be used to decode one input stream in order to gain approximately 3X shorter run-time.
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array
Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array

9 References


[8] MPEG Home page http://MPEG.chiariglione.org/ About MPEG


Implementation and Evaluation of MPEG-4 Simple Profile Decoder on a Massively Parallel Processor Array


