Porting the digital radio mondiale receiver on the Ericsson M7400 platform

Liu, Y.

Award date:
2013

TU/e Technische Universiteit Eindhoven University of Technology

ERICSSON

Porting the Digital Radio Mondiale Receiver on the Ericsson M7400 platform

By

Yang Liu

Final Project Thesis
Eindhoven University of Technology
Department of Mathematics and Computer Science

Student:
Yang Liu (0804638)
y.liu.1@student.tue.nl

Supervisor:
Prof. Dr. ir. C.H.van Berkel
Eindhoven University of Technology
kees.van.berkel@ericsson.com

Tutor:
Dr. ir. Peter de Jager
Senior Software Engineer
Ericsson B.V.
peter.de.jager@ericsson.com

Abstract

Software Defined Radio (SDR) is a modern wireless communication technology. In SDR, the software is implemented on general-purpose processors to handle the communication tasks. Digital Radio Mondiale (DRM) is a new digital broadcasting standard using the SDR technology. This thesis concerns the development of an embedded DRM receiver on the Ericsson platform. The C++ code of the DRM receiver is from the Open Source PC software, Dream DRM receiver. The Ericsson platform M7400 is selected as the hardware to implement the embedded DRM receiver. The DRM code is ported on the MSS (Modem Subsystem) of the M7400 platform to achieve a DRM receiver program of the multi-core version. The MSS contains two ARM Cortex R4 processors and one EVP (Embedded Vector Processor) core. The EVP is a new generation DSP (Digital signal processor) of Ericsson.

The DRM code was firstly simplified and isolated in order to obtain a simple, independent, compatible program running on PC and the program only contained the main DRM receiving process. Before porting the DRM program on the target platform, an intermediate step of porting the code on the Cortex A8 Real-Time System Model(RTSM) was performed. The program was then optimized based on the profile report of the program on Cortex A8. After that it was ported on the Cortex R4 of the M7400 to achieve a single-core version which cannot reach the real-time requirement. The multi-core version of the program was analyzed with the help of the analysis tool, Pareon from Vector Fabrics. Finally we ported Viterbi decoder function on EVP and developed the DMA communication between the ARM and EVP core. Thus a embedded DRM receiver of the multi-core version was accomplished on the M7400. It provided a speed up of 1.7X comparing with the single core version on the Cortex R4 and it could deliver a real-time service.

This project also shows that the ARM and EVP's cooperating architecture on M7400 is suitable to process the SDR receiving task. And Pareon from Vector Fabrics is an appropriate tool in the analysis of the multi-core version.

Keywords: SDR DRM ARM EVP Multi-core Pareon DMA

Abbreviations

DRM	**Digital Radio Mondiale**
MSS	**Modem Sub System**
ARM	**Advanced RISC Machine**
EVP	**Embedded Vector Processor**
RTSM	**Real-Time System Model**
SDR	**Software Defined Radio**
OFDM	**Orthogonal Frequency-Division Multiplexing**
ACC	**Advanced Audio Coding**
QoS	**Quality of Service**
CR	**Cognitive Radio**
SoC	**System on Chip**
NoC	**Network on Chip**
MLC	**Multi Level Coding**
SDC	**Service Description Channel**
FAC	**Fast Access Channel**
MSC	**Main Service Channel**
RISC	**Reduced Instruction Set Computing**
CISC	**Complex Instruction Set Computing**
DMA	**Direct Memory Access**
VA	**Viterbi Algorithm**
HD	**Hamming Distance**
BM	**Branch Metric**
PM	**Path Metric**
MCAPI	**Multi-core Application Programming Interface**
AXI	**Advanced eXtensible Interface**
APB	**Advanced Peripheral Bus**

Contents

Chapter 1

Introduction

1.1 Background of the Project

As various wireless networks make up an increasing part of our daily lives, there is a big desire for the optimization in wireless communication systems. To begin with, modern communication technology standards define an extremely high transmission rate, which is a challenge to the hardware to support a high data rate while minimizing the power consumption of the platform. Therefore, the hardware is required to perform well, both in processing speed and power. Besides, another trend of the wireless communication is to provide a seamless service over various wireless networks in a single device [1]. However, the support of multiple protocols of various networks significantly increases the complexity of the communication task. This can be solved by using the software. The platform uses the general-purpose processor rather than the specific-purpose hardware. The software processing complex tasks of different services is implemented on the general-purpose processor to support the communication. In short, a fully optimized communication software and a well-performed platform are necessary for the implementation of the wireless communication.

Software defined radio (SDR) is a modern software-based wireless communication technology. It realizes modulation, encoding, filtering and other radio communication processes which are traditionally implemented by circuits, by means of the software on PC, cell phone or other computer systems [2]. The SDR supports multiple communication protocols including multi-band, multi-standard, multi-service and multi-channel. This software based implementation has some obvious advantages including ease of adaptation and flexibility over the traditional hardware based analogue radio [3]. Firstly, it is reconfigurable to provide a high adaptable service. The SDR can easily switch within modes according to the environment or user requirements [4]. In the transmitter end, it not only transmits the signal but also configures settings according to the environment. If the Cognitive radio (CR) is applied, which senses the environment and tracks changes,

the SDR automatically reacts on CR's finds to characterize all possible transmission channels, propagation paths as well as modulation methods, and finds the best mode to transmit. Otherwise, the user can configure the transmitter by himself, motivated by his own knowledge or detection of the environment. In the receiver end, it detects the transmission mode and corrects possible errors. What's more, some software tools are used to improve the quality of service (QoS) [5]. Secondly, in terms of the flexibility, the SDR can be easily redesigned for a new or changed protocol and the redesign cost is obviously less than the traditional analogue radio.

In 2001, the European Telecommunications Standards Institute (ETSI) defined an Orthogonal frequency-division multiplexing (OFDM) based SDR, known as Digital Radio Mondiale (DRM). The working frequency of DRM is the same as the analogue radio system and it is divided into 2 modes, DRM30 and DRM+ [6]. DRM30 mode is designed to utilize AM broadcast bands below 30 MHz and DRM+ mode works above 30 MHz, which is the FM band. The analogue radio system working on this frequency range has advantages of a large coverage area and relatively little interference caused by the environment [7]. But it also has some disadvantages, low flexibility, service quality limitations as well as sensitive to interferences from the long-distance propagation. DRM inherits the advantages of the traditional radio and implements the digital technology, various transmission modes and different bandwidths to overcome the analogue system's drawbacks.

DRM's main processes include OFDM modulation/demodulation, mapping/de-mapping, cell interleaving/de-interleaving and so on. DRM achieves these processing routines by software. It has 4 transmission modes, namely Mode A, Mode B, Mode C and Mode D. These 4 modes are various robustness modes which suit different channel conditions and the details can be seen in Table 1.1 from [7]. The ability to select from a range of transmission modes is a key and revolutionary feature of DRM. This allows the broadcasters to balance or exchange bit-rate capacity, signal robustness, transmission power and coverage [6]. The CR technology, as discussed above, has not been applied in DRM yet. Therefore, in response to any local changes in the environment, the DRM user can dynamically changes robustness transmission mode without disturbing the audience.

Table 1.1: DRM's 4 robustness transmission modes [7]

Robustness Mode	**Typical Propagation Conditions**
A	Gaussian channels, with minor fading
B	Time and frequency selective channels, with longer delay spread
C	As robustness mode B, but with higher Doppler spread
D	As robustness mode B, but with severe delay and Doppler spread

1.2 Introduction of the Dream DRM Receiver

In our project, the Dream DRM receiver is chosen as the implementation software target, which is an Open-Source software under the GNU General Public License (GPL). The DRM software is a C++ program running on the personal computer (PC). It is designed to run under Mac OSX, Microsoft Windows and Linux. The start time of the project was June 2001 by the Institute of Communication Technology, Darmstadt University of Technology.

This software project implements a working software receiver with the basic features of DRM. The receiver runs in two modes, DRM and Analogue. In the DRM mode, it receives the DRM signals and provides the service which can be an audio service, possiblly with associated data or a data service. The audio service supports Advanced Audio Coding (ACC) audio and AAC+ audio. The data service supports Electronic program guides (EPG), Multimedia Object Transfer (MOT) Slide Show, Broadcast Web Site and Journaline. In the analogue mode, it can only process the AM, FM analogue signals.

1.3 Introduction of the Ericsson M7400 platform

The Ericsson M7400 is chosen as the target platform to port the Dream DRM receiver. The M7400 is one of the productions of the Thorium Modem Solution which aims to provide modem platforms for smartphones, tablets and connected devices for the LTE (long-term evolutio), HSPA (High Speed Packet Access) market. It is chosen to port the DRM implantation on the Modem Sub System (MSS) of the platform. The MSS contains two ARM (Advanced RISC Machine) processors and one EVP (Embedded Vector processor) processor. The ARM core is expected to process tasks with high complexity but low data rate, for instance, protocol stack handling, user interfaces and application frameworks. High data rate tasks with generic parallel processing, especially vectorizable tasks are assigned to the EVP core. A good task scheduling on the platform, for software which means a good multi-core version, will significantly increase the software's performance and greatly decrease the platform power consumption.

1.4 Problem Description

This project aims to map the DRM Dream receiver on the Ericsson M7400 platform including partitioning C/C++ code into ARM and EVP. In addition, the interface needs to be developed to achieve the communication between processors.

Previous researches of the multi-core DRM receiver's implementation on embedded hardware platforms, provides us useful experiences in the project. The Hijdra project of NXP

contained a work of a DRM program mapping in the Hijdra architecture [8]. The DRM receiver was executed on TriMedia and the Viterbi decoder took up 10% of the whole execution time. In order to improve the performance, the Viterbi decoder was separated from the receiver and mapped on a Viterbi accelerator. A 5% improvement in the speed performance was achieved. Paper [9] also concerned the DRM program mapping. It achieved a run-time mapping of a DRM program to a heterogeneous System on Chip (SoC). The platform consisted of multiple tiles of different types (e.g. ARM, FPGA, DSP, FPGA) interconnected by a Network-on-Chip (NoC). Partitioning a DRM receiver into smaller independent blocks was done and tasks were mapped into different hardware tiles.

The project is started with the DRM implementation's source code downloaded from the Dream DRM's website and the System-C model of the target M7400 platform provided by Ericsson. Finally it is expected to achieve a embedded multi-core version of the DRM receiver implementation on M7400. The main scope of this project is to port the DRM receiver's main processing code on the M7400 platform. Comparing to the related works, an additional goal of this project is to evaluate the EVP and ARM's cooperating performance. The Vector Fabrics tool, Pareon is also involved in the analysis of the program's multi-core version. Vector Fabrics [10] is a company which specializes in developing tools for the design and implementation of multicore, multi-threaded applications and embedded systems.

The project consists of these steps. An analysis and isolation work of the DRM receiver program is done at the beginning. Then an intermediate step of porting the DRM code on an ARM virtual processor with a fast simulation speed, is achieved. On the basis of the profile report delivered in the intermediate step, some modifications must be done to the source code. Afterwards, the modified DRM receiver is ported on the ARM core of the target M7400 platform. Based on the profile report on the target platform, a research of the multi-core version of the DRM program is done with the help of the software Pareon. It is excepted that a similar conclusion as in [8] can be delivered that the Viterbi decoder should be moved to another processor. The Ericsson company provides an EVP version Viterbi decoder. Based on the provided EVP code, a modified Viterbi decoding program suiting the DRM receiver is created. After developing the communication between the EVP and ARM core with the DMA controller, the performance of the DRM receiver multi-core version is estimated and compared with the actual profile report running on the target platform.

1.5 Outline of the thesis

Chapter 1 provides an introduction to the project and relative concepts including the software and the target platform. In Chapter 2, an analysis of the target software, DRM receiver is achieved. A detailed background of the hardware including the ARM Cortex R4, EVP and on-chip communication is discussed in Chapter 3. The mapping work and

profile report on the ARM Cortex A8 and Cortex R4 are presented in Chapter 4. This is followed by the description of Multi Level Coding (MLC), Viterbi decoder working principles, analysis of the multi-core version by Pareon as well as implementation of Viterbi decoder on EVP in Chapter 5. The development of the communication can be seen in Chapter 6. The calculation process of the multi-core improving performance estimation and the actual profile report are in Chapter 7. Finally the study ends with conclusions and future works in Chapter 8.

Chapter 2

Analysis of the DRM Receiver

In this chapter, the outline of the DRM receiver is shown and the functionality of each module is described in 2.1. Subsequently, 2.2 presents the DRM isolation steps and discusses the data flow of the DRM main processing. Finally, the creation and usage of the benchmarks to check functional correctness and to analyze the real-time performance are given in 2.3.

2.1 DRM Receiver Outline

As the DRM receiver's main running target is the computer or the mobile device, low energy consumption and easy implementation are required for these platforms. Therefore, there should be some constraint on the DRM signal. The bandwidth of a DRM passband signal is less than 20 kHz so that there won't be a big pressure on the filtering process in the DRM receiver. What is more, the number of carriers used in the OFDM-modulation is relatively small (max 460). In the OFDM-modulation process of the transmitter, the encoded data are modulated into OFDM symbols. Each OFDM symbol is constituted by a set of carriers and these closely spaced orthogonal sub-carrier signals are used to carry data. Thus the small amount of carriers result in a less heavy computational load of de-modulation an OFDM symbol in the receiver [11].

The outline of the DRM receiver is given in Figure 2.1. There are 6 modules, namely RF-reception, A/D-converter, OFDM demodulation, De-mapping, Channel decoding and Source decoding.

The antenna and RF-reception can use the same modules as the analogue radio.

The A/D-converter converts the analog signal from the RF-reception into the digital signal.

The OFDM demodulation involves a series of complex processing steps including Sam-

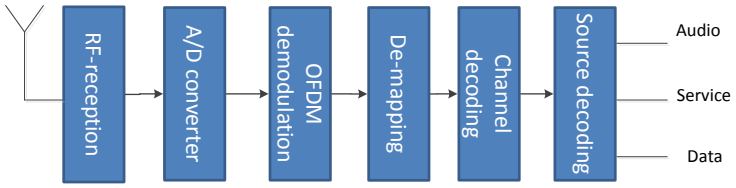


Figure 2.1: DRM receiver outline structure

ple Rate Correction, Synchronization, Channel Estimation and Demodulation. The detailed processing steps are as following [11], the first step is to correct the sample rate and then acquire a coarse frequency offset estimation without the detection of the robustness mode. Based on the received signal, the receiver detects the robustness mode and achieves the timing acquisition which means getting the position of OFDM signals. With the knowledge of the robustness mode and timing, the useful part of OFDM signals can be extracted and demodulated. The first OFDM symbol of each frame contains additional pilots for frame synchronization. After obtaining frequency pilots, the receiver can call the frequency offset tracking to achieve the frame synchronization and get the frame from the signal. Now the beginning of the frame is obtained, so channel estimation and timing tracking can be called. With timing tracking, the obtained frame in frame synchronization is corrected. Thus the receiver obtains all the transmitter parameters, channel parameter and uses them to obtain demodulated signals. The dataflow chart of the OFDM demodulation can be seen in Figure 2.2.

The De-mapping processing stage follows the OFDM demodulation and it divides demodulated signals into 3 channels, Main Service Channel (MSC), Fast Access Channel (FAC) and Service Description Channel (SDC), based on the cell mapping of the transmission frame. The detailed transmission frame structure can be seen in [7]. MSC is the channel of the multiplex data stream which occupies the major part of the transmission frame and it carries all the digital audio services, together with possible supporting and additional data services. FAC is the channel of the multiplex data stream containing the information that is necessary to find services and begin to decode the multiplex. SDC is the channel of the multiplex data stream which gives information to decode the services included in the multiplex and information to enable a receiver to find alternative sources of the same data [7].

A cell de-interleaving function is applied to MSC in the receiver. Because MSC is transferred in a higher protection level than the other two channels by locating an additional cell interleaving processing after the MSC channel encoding process in the transmitter. The cell interleaving routine is aimed at changing the possible burst error into the random error which can be corrected by the receiver's channel decoding routine. The burst error means error occurs in contiguous sequence of symbols while for random er-

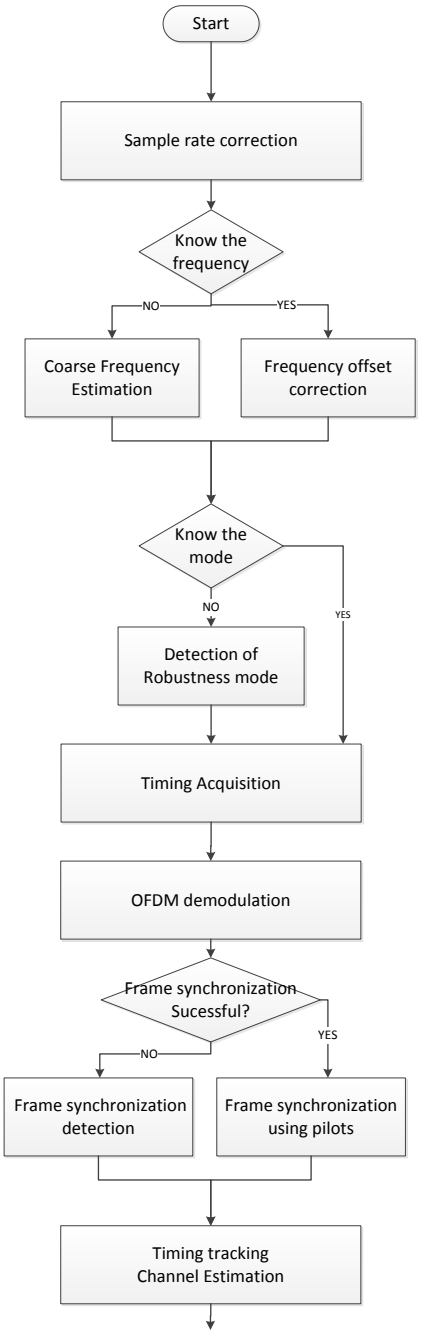


Figure 2.2: OFDM demodulation's dataflow chart

ror, error symbols are randomly located in the sequence. After interleaving, the adjacent code blocks scatter. So adjacent error blocks are also separated and the burst error is converted into the random error.

The Channel decoding processes MSC, FAC and SDC separately. The Channel decoding in DRM is multi level decoding including QAM de-mapping, Viterbi decoding and so on. In MSC, FAC and SDC, the decoding parameters are all different.

In the last stage Source decoding, the output bits of the Channel decoding are decoded into audio, data or other services.

2.2 Analysis and Isolation of the DRM receiver processing flow

All the work and result of the DRM on PC is achieved on the Linux OS of a workstation using an Intel i7 processor. Since the DRM receiver program has to be ported from the PC to an embedded system, it is not wise to port the whole original program. It is better to delete some functions unnecessary on the target platform, to simplify the main DRM process for the purpose of achieving a minimal version of DRM receiver. Furthermore, an isolation work of the code related to the main DRM processing flow is finished on the minimal DRM receiver in order to get a simple, independent and compatible DRM receiver program. The structure of the expected DRM receiver program can be seen in Figure 2.3.

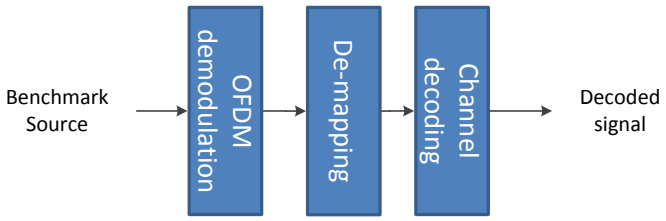


Figure 2.3: DRM Receiver expected minimal structure

These steps have been done to achieve a minimal version of DRM receiver. Firstly, the Dream DRM receiver is built on our own PC's Linux environment with supported external libraries and building tools. An analysis of external libraries is done and it comes to a conclusion that for the minimal version, the only external library needed is Libfftw, which is a library to provide the function of "Fastest Fourier Transform", and other libraries can be ignored. The details of the FFTW library can be referred to [12]. The relevant code of the Libfftw library is extracted and embedded into the main function so that the main program can call the required functions of FFTW easily.

Secondly, the main program is simplified to isolate the DRM receiver function. Many functions like GPS, DRMLogger, Graphical User Interface (GUI) and so on which run on PC, are not useful on the target platform. The original program has the code to support the function of the DRM transmitter, receiver and simulation of the whole process from the sender via the channel to the receiver. In the receiver part, there are also routines to support the analogue mode to receive and demodulate AM and FM signals. These functionalities are not relevant to the main scope of the project, so they are removed.

Finally, a further-simplification step on the main signal processing routine is implemented. The modules reading from the sound interface including audio file, sound card, as well as decoding source and playing the decoded audio are removed from the flow.

A full image of the DRM receiver processing flow after these simplification steps is presented in Figure 2.4.

In the part of OFDM demodulation, there are 6 stages, namely Input Resample, Frequency Synchronization Acquire, Time Synchronization, OFDM Demodulation, Synchronization using pilots and Channel estimation. OFDM cells are demodulated at the output of the OFDM demodulation. In the part of de-mapping, there is one processing stage, OFDM cell De-mapping. It separates the MSC, FAC and SDC off the carriers. The next part is Channel decoding, it decodes MSC, FAC and SDC separately. Depending on the different importances of information channels, the decoding function of various level complexity is applied to MSC, SDC and FAC separately. The decoding process of MSC is extremely complicated and costs much more time comparing with the routine of FAC and SDC. The detailed working principle of decoding is discussed in section 5.1. What's more, two more stages, De-multiplexer and De-interleave for cells are applied for MSC channel decoding. The decoded SDC and FAC contain the information on how to decode the MSC and how to find alternative sources of the same data, and give attributes to the services within the multiplex [7]. Therefore, SDC and FAC are imported into the Utilization Stage to change the setting of the receiver.

Since the DRM receiver has to process the continuous real-time input and output data stream, the program uses the input driven processing sequence which means the process routine is called when enough input data is delivered to the processing stage. There are buffers between processing stages and they act as the input or output of the stage. The processing stage checks its input buffer whether data is enough to call one routine. If so, it calls the processing function and exports the output data to the output buffer which also acts as the input of the next stage. Otherwise, the stage keeps on waiting. Two kinds of buffers are used in the program, namely single buffer and circular buffer. The single buffer is used when the input block size of the buffer equals to the output block size. However sometimes the buffer needs a different input size and output size. In this case, the circular buffer is applied.

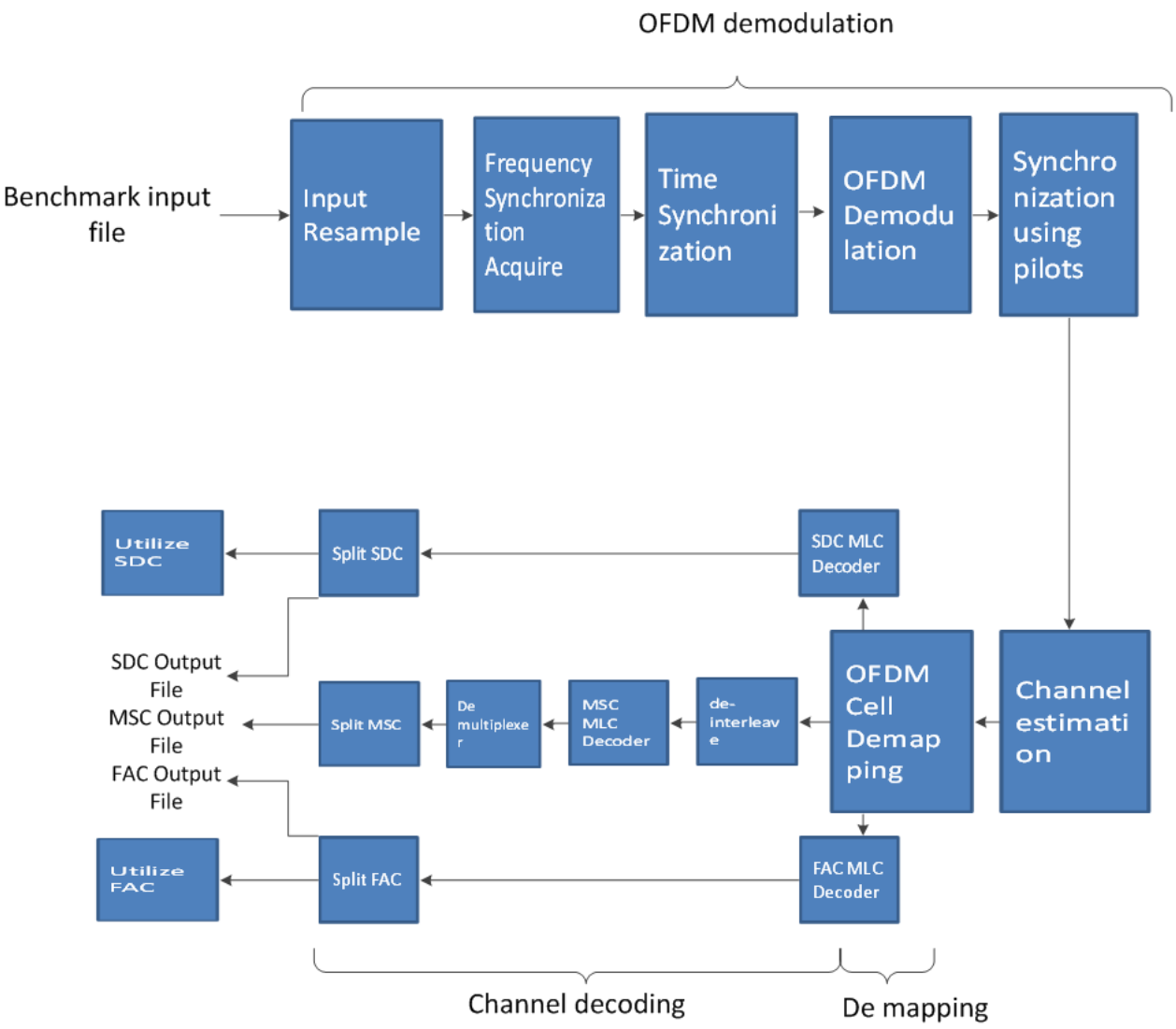


Figure 2.4: The minimal version of DRM receiver and its benchmark files

2.3 DRM receiver's benchmark

The DRM dream project website provides some benchmarks to help developers and users to check the functional correctness of the receiver. These benchmarks are antenna-received modulated digitized signals recording from the sound card interface of an integrated DRM receiver. The received signals are packed in the Wav file. Here the benchmarks containing audio services are chosen, because the audio service is the main service of the DRM and there is a strict processing time requirement on such service.

The received signals cannot be directly imported into DRM receiver processing stages.

They must be processed in the stage of Read Data before demodulation. In the Read Data stage, it reads the signal recording file in the buffer in the type of 16-bit int and checks the sample rate of the file from its header to see whether it is equal to the receiver's supporting rate 48000Hz. If unsupported, an additional resample has to be done. Then according to the sound file's audio channel setting, namely stereo and mono, an audio channel process is used to produce a two channel signal stream. If the channel is mono, it extends the mono channel into a stereo channel by copying the values of the single channel to another channel. If the channel is stereo, it just copies the data. At last, the receiver handles the stereo channel data based on the receiver's setting, the default setting is **mix channel**. For the setting of **mix channel**, it converts stereo channel data from 16-bit int type to 64-bit double type and then mix two channels by averaging their values. There are also other settings like left channel, right channel, I/Q input and so on. Thus all the pre-processing routines are finished and the achieving data are stored in buffer *DemodDataBuf*. The DRM main processing function reads the data in buffer *DemodDataBuf* and starts the main DRM processing.

At the output of the main DRM processing routine, information channels, MSC, FAC and SDC, are obtained. It utilizes the information in FAC and SDC to change parameters and demodulation methods in the receiver and it decodes the MSC information into the audio file and plays the audio. Using the GUI the information of the audio quality can be seen including signal-to-noise ratio (SNR) and weighted modulation error ratio (WMER) and delay. From paper [13], it can be concluded that audio dropouts detectable by non-professional listeners do not occur if the signal-to-noise ratio is greater than 17 dB.

Three benchmark of received signals in different robustness modes, are chosen to test the original receiver. The parameters of them can be seen in Table 2.1. These benchmark files are used to test the receiver and all SNRs are larger than 17 dB. The detailed result can be observed in the evaluation dialog window. The detailed result including SNR, WMER and delay can be seen in Table 2.2. The delays which indicate the time period between the launch of the software and the start of the audio service, are also acceptable.

Table 2.1: Benchmarks for the DRM receiver

File name	**Bandwidth**	**sample rate**	**Robustness Mode**
received_signals_recording1	10kHz	48000Hz	Mode B
received_signals_recording2	10kHz	48000Hz	Mode A
received_signals_recording3	10kHz	48000Hz	Mode C

Our minimal version of the DRM receiver only contains the main DRM process. The interface with different source inputs, decoding MSC and playing the audio are excluded. The chosen benchmarks cannot be directly processed to do the functional check of the program. The new benchmarks are created for the main DRM signal processing program based on the original received signal benchmarks using the default setting in the Read

Table 2.2: Benchmark results for the DRM receiver

File name	SNR	WMER	Delay
received_signals_recording1	31.8db	28.5db	0.82ms
received_signals_recording2	17.4db	17.7db	0.58ms
received_signals_recording3	21.2db	19.9db	0.82ms

Data stage as discussed before. The creation process can be seen in Figure 2.5.

The original program with different benchmarks are launched and the data in *Demod-DataBuf* is stored, which is the output buffer of the stage Read Data, in order to obtain the input data of the main DRM processing. The data is stored and it can be used as the test input file of the DRM main signal processing program. The data of *SDCDecBuf*, *FACDecBuf* and *MSCDecBuf*, which are output buffers of the stage Channel Decoding, is also saved into files as the standard test result of the main DRM processing. Thus the new benchmarks are created.

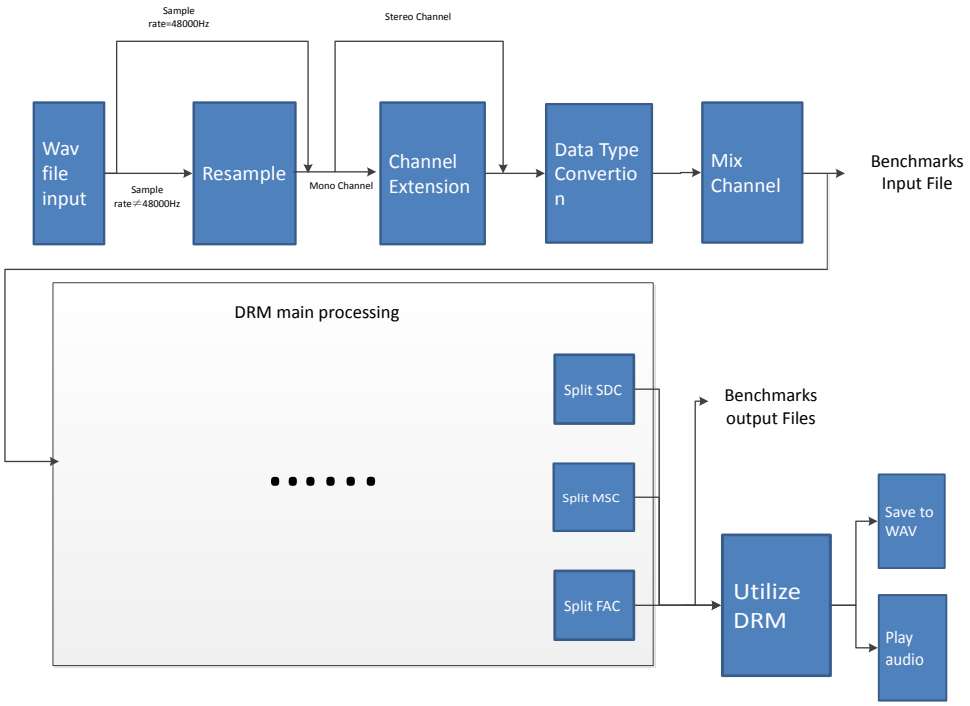


Figure 2.5: generation of the DRM main processing program's benchmark input and output files

Through an observation of the created benchmarks, it can be found that the data in the input file are in the type of 64-bit double and their ranges are from -32168.0 to 32167.0. The content is the digitized, time domain, modulated, 48kHz sample rate received radio

signal stream. In output files, there are 3 files for MSC, SDC and FAC separately and data in files are binary in the type of 8-bit char. The DRM sends the information in the unit of frame and MSC, SDC and FAC bits are mixed in the frame. The program firstly divides bits belonging to different information channels and then writes the divided bits of one frame into output files for MSC, SDC and FAC. The details of benchmark files can be seen in Table 2.3.

Table 2.3: Benchmarks for the DRM receiver's main processing program

File name	**type**	**data type**	**file size (byte)**	**frame number**	**bit number per frame**	**original benchmark file**
mytest1	input	64-bit double	28794880	-	-	received_signals_recording1
1MSC	output	8-bit char	4888800	175	6984	
1FAC	output	8-bit char	52416	182	72	
1SDC	output	8-bit char	158760	63	630	
mytest2	input	64-bit double	13051656	-	-	received_signals_recording2
2MSC	output	8-bit char	2834400	75	9448	
2FAC	output	8-bit char	22752	79	72	
2SDC	output	8-bit char	81852	29	630 to 798	
mytest3	input	64-bit double	22400368	-	-	received_signals_recording3
3MSC	output	8-bit char	506736	138	3672	
3FAC	output	8-bit char	40608	141	72	
3SDC	output	8-bit char	57792	49	282 to 630	

In order to evaluate the modified DRM processing program's demodulation and decoding quality, a tiny testing program is created to compare the MSC output with the benchmark MSC output, frame by frame to measure the bit error and error rate in each frame.

Besides functional correctness, it is also required to test the timing performance of the DRM program. The DRM receiver is a real time system and it is required to provide a

lasting audio service which means that the audio service will not be interrupted because of the unfinished processing of an audio frame. The DRM processing execution time should be less than its produced audio duration time so that a new audio stream is produced before the end of the current playing audio stream.

One transmission frame contains one MSC frame and other SDC, FAC frames and these SDC, FAC frames contain the information of how to obtain the MSC frame in the same transmission frame. Thus, only the whole transmission frame is processed, the receiver can obtain the MSC frame. Each MSC frame is in the same size for each benchmark. After decoding, it generates the audio of the same time period. Therefore, based on the audio stream duration and the MSC frame number, the audio stream duration per frame can be calculated, which can be seen in Table 2.4. It can concluded that audio stream duration per MSC frame is more than 400 ms. Considering the real time requirement, it means that the processing time of each MSC frame (equalling to the processing time of the whole transmission frame) should be less than 400 ms. However, for the first frame, there is no such requirement because the processing time of the first frame only results in a latency before the start of the service and will not influence the lasting audio.

Table 2.4: The audio stream duration per MSC frame

benchmark input	**audio stream duration**	**number of MSC frames**	**duration per MSC frame**
mytest1	73420 ms	175	420 ms
mytest2	30814 ms	75	411 ms
mytest3	55629 ms	138	403 ms

In conclusion this chapter mainly describes the software. The background of the DRM software is introduced and the detailed analysis, isolation work of the program and the creation of the benchmarks are discussed. The introduction of the hardware is presented in next chapter.

Chapter 3

Background of the M7400 platform hardware

The background of the target M7400 platform hardware is introduced here. Since it is chosen to port the implementation on the MSS system, the structure MSS system of the M7400 is presented in Figure 3.1. The ARM processor (section 3.1), the DSP processor EVP (section 3.2) as well as the on-chip communication (section 3.3) are presented respectively in this chapter.

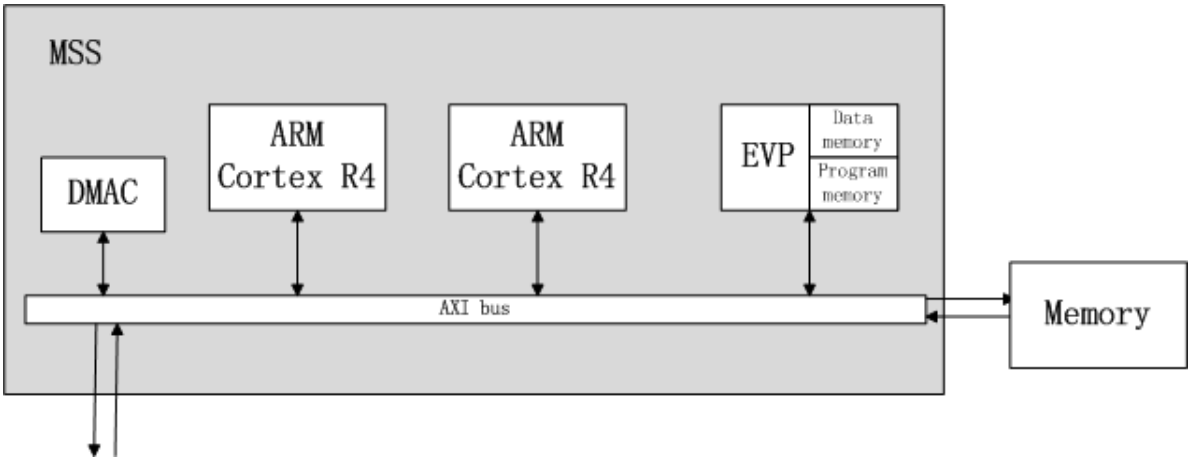


Figure 3.1: The MSS system of the M7400 platform

3.1 ARM

ARM architecture processors are a family of Reduced Instruction Set Computing (RISC) -based computer processors which are designed and licensed by the company ARM Hold-

ings. Now ARM architecture processors are widely used in the electronic market. The ARM based product's market share is more than 75% in mobile phone market, 25% in mobile computers and digital TVs, 50% in enterprise applications and less than 5% in Microcontrollers/Smartcards in 2009 according to [14].

The original DRM software is aimed at personal computer which contains the processor in X86 architecture rather than ARM. X86 is based on Complex Instruction Set Computing (CISC) which leads to the most essential difference comparing with ARM of RISC.

The different instruction sets result in various characteristics of ARM and X64 processors. On one hand, the ARM core has a simple hardware architecture, as a result the ARM core has an obvious advantage in energy performance. The power saving advantage makes the ARM core suitable in embedded systems like the smart phone or Tablet PC. On the other hand, the X86 processor especially Intel core's complex hardware has the strength of execution speed so it is always used in computers and servers. A performance comparison between typical ARM and X86 processors is provided in paper [15]and [16]. In the first paper, the ARM Cortex A8 and Atom N330 are used to test benchmarks . They are both aimed at embedded system, especially mobile processor markets. With integer benchmarks, the Cortex A8 is slightly slower than the Atom N330 in the benchmark performance per Mhz. However, the Cortex A8 performance is even 100 times slower than the Atom N330 in double precision floating point benchmarks. The Cortex A8 deliveries a big advantage over the Atom N330 in power consumption ranging from 1 to 8 times power saving in most integer and double precision floating point benchmarks. In paper [16], comparisons of Cortex A8 to Atom N450 and Cortex A9 to Intel i7 are made. Considering the execution time, the A8 is about 4 times slower than the Atom and the A9 is approximately 7 times slower than the i7. But the A8 only consumes one third power comparing with the Atom and A9 costs 20 times less power than i7.

In the target M7400 platform, the ARM core is Cortex R4. The Cortex R4 is aimed at deeply embedded system, which means the system is of big constraints in terms of memory, time and power consumption [17]. In a deeply embedded system the functionality or the behaviour normally is not altered very often and the end user is not able to modify, add or remove functionality to it.

The Cortex R4 was released in 2006 and is designed for semiconductor processes from the 90 nm node onward. The advanced semiconductor technology enables the Cortex R4 to achieve a high performance with a limited clock frequency and power consumption. Besides the semiconductor technology, there are also many other features enhancing the Cortex R4. It supports two instruction sets, ARM and Thumb-2. The ARM instruction set has comprehensive data-processing, control functions and a high performance. On the contrary, the Thumb-2 instruction set provides a higher code density, lower memory size and cost but sacrificing the performance. Consequently, the ARM core can alternately use two instruction sets to execution different parts. Critical parts like handling

interrupts use ARM instruction sets to ensure the performance and insignificance parts are executed by the Thumb-2 instruction to reduce the cost [18]. The Cortex-R4 is also highly configurable. The configuration includes clock frequency, cache, Tightly-Coupled Memory Interface and so on. Therefore the hardware can be minimized according to the running software to save the energy. The Cortex R4 also enables a quick response to an interrupt ranging from 20 cycles to 30 cycles [19].

The Cortex R4 provides another version called Cortex R4F, which includes a Floating Point Unit (FPU) extension. The Cortex R4's FPU is based on VFPv3-D16 (Vector Floating Point) architecture, which gives a full support of single-precision and double-precision arithmetic operations. It includes a register bank to support floating point operations. There are 2 views of the register bank, 16 double-precision 64-bit double word registers, D0-D15 and 32 single-precision 32-bit single word registers [20].

With FPU, the Cortex R4 processor's application field expands, for instance, automotive electronics that uses sophisticated control algorithms, accurate image processing method and other single precision floating point applications. What's more, the existing software written in C/C++ or other high level programming languages , including floating-point algorithms, can be re-targeted to the ARM platform with less cost of the speed. When required, the FPU can perform double-precision (64-bit) floating-point calculations at the expense of some calculation speed [19].

Without FPU, the Cortex R4 processor has to use the software to emulate floating point operations, which leads to a much higher time cost comparing with using FPU.

In summary, the Cortex-R4 processor delivers a high performance combined with cost and power efficiencies across a broad range of deeply-embedded applications. Cortex R4's main application markets includes imaging and printing device, automotive system control, storage device driver and wireless communication device [19].

3.2 EVP

The EVP is a next generation Digital Signal Processor (DSP), which is designed for high computation applications such as 3G, 3.5G and Multimedia. It processes data in parallel as in the traditional DSP. Its processing speed can be up to 30 GPOS (30×10^9 operations per second). Besides SIMD (Single Instruction Multi Data), it also supports VLIW (Very Long Instruction Word). The maximum VLIW-parallelism available equals 5 vector operations plus 4 scalar operations plus 3 address updates plus loop-control [21]. Some specific vector operations including Intra-Vector operation as well as Shuffle operation, are also delivered by the EVP.

The EVP architecture combining SIMD and VLIW can be seen in Figure 3.2 from paper [22]. The main word width is 16 bits and it also supports 8 and 32 bits. The bit supporting design is based on the characteristic of the wireless protocol which is

the main EVP application target. Since most wireless protocol's algorithms operate on variables with small values, 16 bits length is normally big enough.

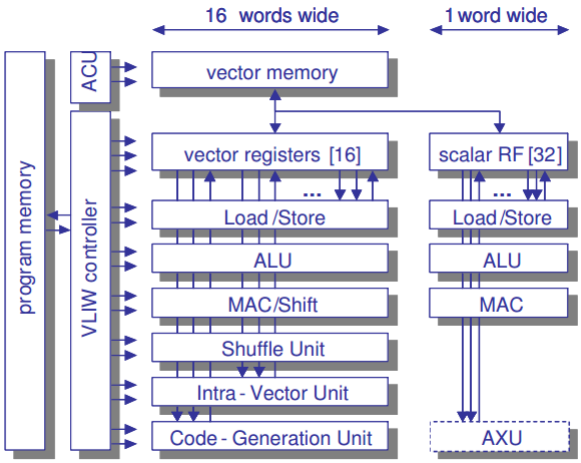


Figure 3.2: The EVP architecture [22]

The main computation units in the EVP are divided into 3 groups, namely Scalar Data Computation Unit (SDCU), Vector Data Computation Unit (VDCU) and Address Computation Unit (ACU). VDCU contains Vector Load Store Unit (VLSU), Vector Arithmetic Logical Unit (VALU), Vector Mask Arithmetic Logical Unit (VMAU), Vector Multiply ACcumulate Unit (VMAC), Vector Shuffle Unit (VFU) and Intra Vector Unit (IVU). With the support of VDCU, EVP can process operations on all elements of the vector in parallel or operations within the elements of a single vector. In the SDCU, there are Scalar Load Store Unit (SLSU), Scalar Arithmetic Logical Unit (SALU), Predicate Arithmetic Logical Unit (PALU) and Scalar Multiply ACcumulate Unit (SMAC). SDCU provides the hardware to compute the single variable and it can work in parallel with VDCU.

From the introduction of the EVP hardware, it can be seen that the EVP is good at high-performance generic parallel processing. However it is not optimized for complex tasks, for instance, real-time controlling, protocol handling and so on. It usually works in the multi-processor system to co-operate with other processors. A typical multi-core system design and its task allocation can be seen in Figure 3.3. It can be concluded from the figure that high complexity tasks are executed in the ARM and DSP. The low complexity tasks with big bandwidth is assigned to the EVP and the hardware accelerator.

The EVP also performs well in the power consumption. Paper [22] shows that the EVP of 90nm running at 300 MHz generates a power of 1mW/MHz including a typical memory configuration. The EVP power performance can be compared with the Signal processing On Demand Architecture (SODA) which is also a DSP designed for the SDR

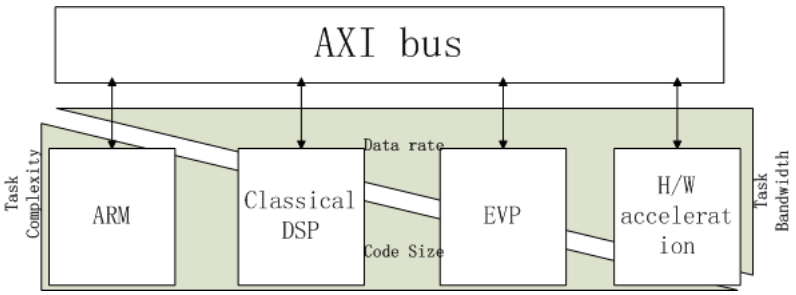


Figure 3.3: The task assignment of EVP-involved multi-core system

application. At 180nm, the power of SODA is 3W which is predicted to reduce to 250 mW if 65 nm technology is applied [23]. Normally, a typical hand held wireless device has a total power budget of 100 mW $\sim$ 300 mW [1]. Consequently, the energy performance of EVP is in the same level as other SDR processors and it is suitable to be applied on the handle device platform without a large power consumption.

3.3 On-Chip Communication

On-chip communication is an important element in the multi-core system since the components of the system need to communicate with each other during the running of the system. There are usually two requirements on the communication. The first is that the communication must ensure a correctly and reliably data transfer. This is an essential demand for communication. Another requirement is the latency guarantee which implies that a data unit must travel through the communication architecture and reach its destination within a finite time determined by a latency bound [24]. The latency is influenced by many factors including bandwidth, interconnect topology, commination protocol and so on.

The NoC (Network on Chip) of the target M7400 platform is bus. It is a simple architecture and all components are connected to a shared bus. There is a Direct Memory Access (DMA) in the bus to control all kinds of data transfers including memory to memory, peripheral to memory , memory to peripheral and peripheral to peripheral. Other master components write to DMA's slave port to access to the DMA's configuration registers and launch a transfer. After that, the master component is free and can involve in other tasks. While DMA's master port will connect to the source and the destination to control the transfer. The transfer initialized by DMA is the burst transfer which sends multi-data in a burst with requesting only once for the access token, thus it can achieve a very high throughput comparing with other transfer modes.

There are some existing bus-based communication architecture standards which define data transfer modes, protocols, bus architectures as well as component interfaces. The application of the architecture standard will significantly speed up SoC integration and

promote IP reuse over several designs [24]. Some popular architectures are listed here, ARM Microcontroller Bus Architecture (AMBA) 2.0, 3.0, IBM CoreConnect, STMicroelectronics STBus and so on. The architecture applied in the target platform is Advanced eXensible Interface (AXI) bus which is introduced in the AMBA 3.0 bus architecture.

The AXI bus is a high performance bus which is designed to support the connection of high bandwidth, high frequency components without using complex bridges. It is backward compatible to AMBA 2.0 AHB and APB interfaces. AXI provides a burst-based, pipelined data transfer bus, Five separate channels are defined: read address, read data, write address, write data, and write response and bandwidth ranges from 8 to 1024 bits.

The working principle of AXI is shown in Figure 3.4 and 3.5. While reading, the response information from the slave is received on the read data channel. While writing, the response information from the slave is received on the write response channel. In the burst mode, AXI requires the address of only the first data item in the burst to be transmitted and the following addresses of the burst do not need to be transferred but calculated by the slave interface itself based on the burst's first address. Thus the address channel is freed and can be assigned to other transfer tasks. This is an improvement of AMBA 3.0 comparing with AMBA 2.0. In the AMBA 2.0 AHB bus, every data's address has to be transferred. In summary, the AMBA 3.0 AXI bus in the platform can deliver a high throughput, reliable communication to the system.

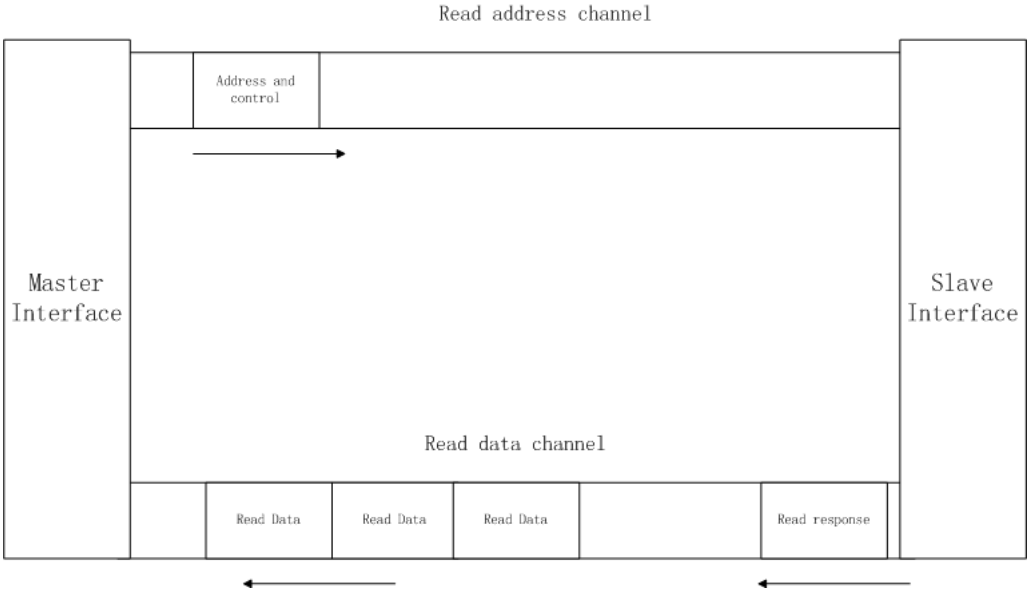


Figure 3.4: AMBA AXI bus channel reading working principle

In this chapter, the hardware of the M7400 platform is introduced. The work of porting the software on the hardware is discussed in next chapter.

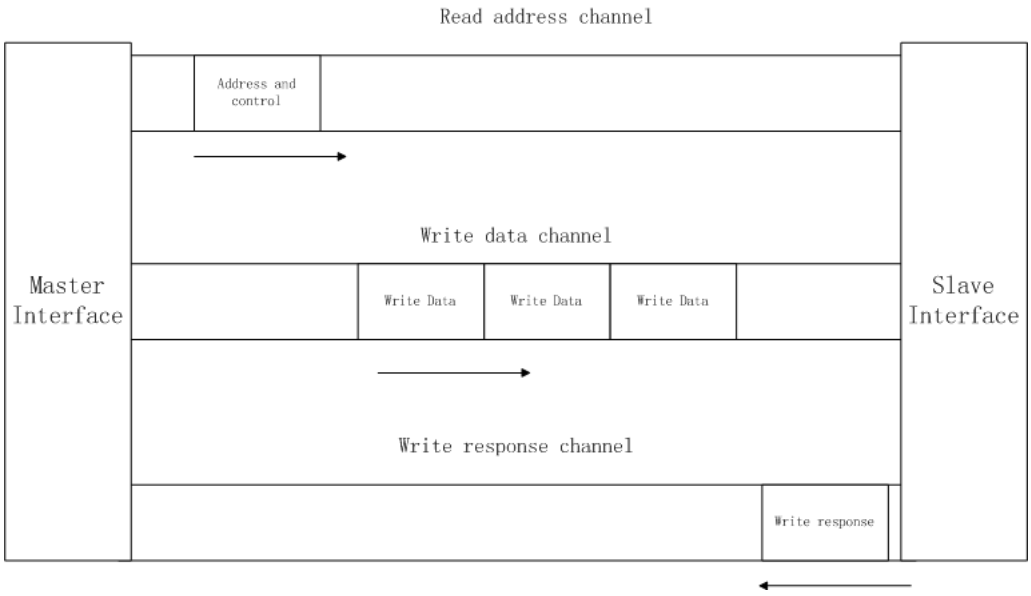


Figure 3.5: AMBA AXI bus channel writing working principle

Chapter 4

Porting the DRM Receiver program on ARM

This chapter discusses the porting work on ARM. An intermediate step of porting the Cortex A8 model is shown in 4.1. The comparison between the Cortex A8 processor and the Cortex R4 processor is achieved in 4.2. Then based on the profile report on Cortex A8 and the comparison conclusion, the DRM receiver needs to be optimized and ported on the Cortex R4 in 4.3.

4.1 Porting the DRM receiver program on Cortex A8

Before porting on the target platform, a intermediate step of porting on an easy transplant ARM core, is necessary for the reason that the software changes its running environment from X86 to ARM. On the X86 processor, the compile tool is gcc while the program has to be compiled by armcc on ARM. Different compile tools and different processor architectures may cause unexpected errors of the program, therefore a verification step to confirm the functional correct of the program on the ARM core is a need. As mentioned in 3.1, the processing ability of ARM architecture is obviously weaker than that of the X86 architecture. The execution speed on the ARM may be significantly slower than on the PC, even cannot reach the real-time requirement. As a result a measurement on the processing time is necessary.

The ARM Development Studio 5 (DS-5) supports debugging on Cortex-A8 Real-Time System Model (RTSM). The RTSM helps users to debug the software on the ARM without the requirement for actual hardware. The model has already been installed the Linux as the operation tool. The DS-5 tool will automatically link the application image without user's own boot file. The model's running speed is also satisfactory. The RTSM's simulation time (the real world time) is a few minutes in running DRM program of processing 11 frames. However for the same workload, the System-C model of the

target platform takes more than two hours in actual time. Therefore, it is not wise to directly debug on the the target platform's System-C model. In RTSM, the absolute timing accuracy is sacrificed to achieve the fast simulated execution speed. The model can be used for confirming software functionality, but the accuracy of cycle counts, low-level component interactions, or other hardware-specific behaviors are not reliable [25]. Nevertheless, it is still a good reference to consider the coarse execution time. In short, the Cortex-A8 RTSM is a suitable platform to process a verification routine.

The DRM receiver is directly mapped on the Cortex A8 model without any modification or optimized compile option. 3 benchmark inputs mytest1, mytest2 and mytest3 are imported into the program to do the functional check. All output results are 100% matched with the standard benchmark outputs.

The function $gettimeofday$ can get the current time of Linux expressed in seconds and microseconds. Normally the time is transformed into millisecond to support a millisecond accuracy measurement. The function can be allocated at the beginning and the ending of the target part to get the running time of the target. This is used to measure all the time result in this section.

The Linux time function is used to measure the running time of ARM DRM receiver and the result is compared with the receiver's profile report on PC which is also achieved by this function, $gettimeofday$. The comparison data can be seen in Table 4.1. It can be seen that the ARM version is as large as 300 times slower than the PC version. The ARM version is extremely far away from the real-time service. For instance, in benchmark1, the processing time of each MSC frame is 3447 ms in average, which is about 10X larger than 400 ms. As mentioned in 3.1, the ARM core's ability in processing floating point is significantly weaker than the Intel PC and the receiver program is written based on the floating point algorithm where most variables are in the type of double precision floating point. The floating point results in the long execution time on the ARM core.

Table 4.1: The execution time comparison of the DRM receiver on PC and Cortex R4 model

benchmark	**PC (X86) execution time (ms)**	**ARM Cortex A8 execution time (ms)**
mytest1	2432	589460
mytest2	1181	290907
mytest3	1325	349277

A study of the ARM core has been made to determine the optimization strategy. A program computing the cumulative sum from 1 to 99999 is used as a benchmark to test the simulation speed of the model dealing with different types of data. The result can be seen in the 2nd column of Table 4.2. It shows that processing floating point takes 100 times longer time than integer.

The default compile setting for the floating point is *mfloat-abi=soft* which means the compiler generates output containing floating point software library calling for floating-

Table 4.2: The cumulative sum testing program result on the Cortex A8 model

data type	**execution time with floating-point library (ms)**	**execution time with floating-point hardware (ms)**
int	1.6	1.6
float	93.9	5.6
double	117.8	5.7

point operations. From the result it can be concludes that the processing of the floating point is extremely slow using the floating point function call.

There are 3 options for compiler to process floating point *soft*, *softfp* as well as *hard*. *soft* calls the library. *softfp* allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. *hard* allows generation of floating-point instructions and uses FPU-specific calling conventions [26].

soft is slow and but is used when the ARM processor does not support floating point hardware operations. Both *softfp* and *hard* generate hardware floating-point instructions which make floating-point instructions efficient. But when using *hard*, all the programs and libraries are required to be compiled using this option. *softfp* is chosen to enhance the compatibility to benefit further extension development. Using the compile option *mfloat-abi=softfp*, the execution speed of the testing program can be seen in the 3rd column of Table 4.2. The execution speed for the floating point is significantly increased when the floating point hardware operation is applied instead of calling library functions. What's more, operations on the single precision floating point variable (float) and double precision floating point variable (double) take almost the same time.

In order to obtain a detailed concept of the Cortex A8's performance and FPU's enhancement, a comparison is made, between the Cortex A8 model using FPU and the PC in the ability in processing integer and floating point data. The benchmark, **Dhrystone** from [27] is chosen to test on the Intel i7 PC workstation and the Cortex A8 RTSM respectively to study their abilities of processing integer. The results are provided in the second column of Table 4.3. The VAX MIPS result is obtained by dividing the number of Dhrystone routines per second by 1757. Similarly, the **Whetstone** from [28] is used to test the floating point processing and the single precision and double precision result can be seen in 3rd and 4th column of the table. It uses Million Instructions executed Per Second (MIPS) to measure the performance. From the result, it can be seen that the Intel i7 has an extremely big advantage in processing data with integer, float and double types. It also can be concluded that the processing speed in float and double type is almost the same in the Cortex A8 FPU.

Based on the discussion of the floating point before, the DRM receiver program is recomplied using the option *mfloat-abi=softfp*. The generating floating point hardware operations are executed by the FPU of Cortex A8, VFPLite which is in the Vector

Table 4.3: The Dhrystone and Whetstone benchmark result on Cortex A8 RTSM and Intel i7 PC

	Dhrystone (VAX MIPS)	**Single-precision Whetstone (MIPS)**	**Double-precision Whetstone (MIPS)**
Intel i7	9517.6	2000.0	5000.0
Cortex A8	142.3	148.8	153.4

Floating Point v3 (VFPv3) architecture. The processing time of 3 benchmarks can be seen in the 2nd column of Table 4.4.

Table 4.4: The execution time comparison of the DRM receiver on Cortex R4 model with different compile options

benchmark	**using vfpv3 (ms)**	**using vfpv3 and NEON (ms)**
mytest1	80425	75469
mytest2	37712	35497
mytest3	47267	43752

There are still some spaces to further improve the performance of the program on the Cortex A8 core because the Cortex A8 implements the NEON technology. It is a Single Instruction Multiple Data (SIMD) extension which provides standardized acceleration for media and signal processing applications. It supports data types including integer and single precision floating point. The code is re-compiled with option *mfloat-abi=softfp* and *-mfpu=neon* and *-ffast-math*. *-ffast-math* is used to speed up the program by sacrificing the math precision. And NEON cooperates with VFP to enhance the performance. VFP can be used for "normal" (non-vector) floating-point calculations. Also, NEON does not support double-precision floating point so only VFP instructions can be used for that. The timing performance results can be seen in in the 3rd column of Table 4.4.

In the practical use of the benchmark, processing the whole benchmark input takes too much time especially on the model of the target platform. In the following work, only part of the benchmark is imported so that the process of generating the first 11 frames is analyzed and each frame contains a MSC frame.

A function correctness check of the receiver also has been done to the ARM version using the vfpv3 and NEON. The error rate testing program is implemented to evaluate the receiver's output. The error rate of 3 benchmarks can be seen in Figure 4.1. When using hardware floating point operations to process floating point calculations, the calculation accuracy is different from using software floating point library. In the receiving process, there is a synchronization process at the beginning of the demodulation routine and the synchronization algorithm is written in floating point. The algorithm is sensitive to the accuracy so it results in a different demodulated result at first few frames. However, with the process of the demodulation, the errors are corrected and the error rate in the MSC frame is decreased.

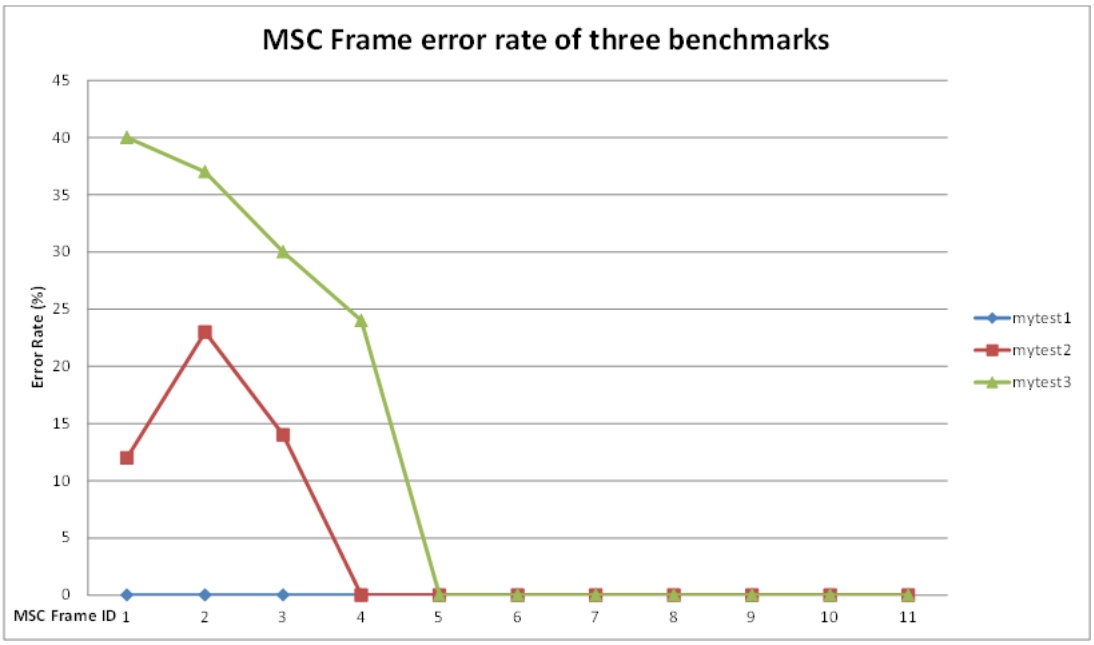
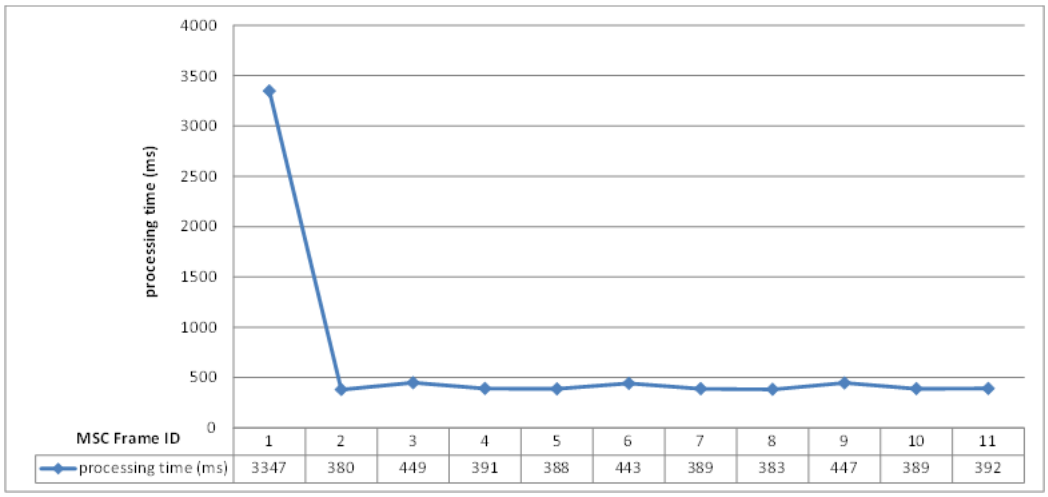


Figure 4.1: MSC frame's error rate of three benchmarks

Focusing on the timing performance, the time interval of generating each MSC frame is measured in Figure 4.2. The first MSC frame takes a much longer time comparing with the following frames because the synchronization routine as well as some initialization functions have to be called in generating the first frame which cost a large amount of time. After that generating each frame costs an approximately same time. Besides, another regular pattern is also discovered. Except the first frame, the processing time is relatively larger in Frame 3, 6, 9 since the workload is bigger in generating these frames. A structure of the transmitted frame can be seen in Figure 4.3 [7]. A transmission super frame contains 3 transmission frames. Processing the first transmission frame will generate the first MSC frame in the transmission super frame and the workload includs demodulation and decoding of one MSC frame, one SDC frame and one FAC frame. However, in the 2nd and 3rd transmission frame, the computation load only includes one MSC frame and one FAC frame. Therefore, generating the first MSC frame of the transmission super frame takes a longer time.

From the real-time view, it cannot provide a lasting service because the processing time of some MSC frames, is longer than 400 ms though it is very close to the real-time requirement.

A profile report is achieved to observe the processing stages' distribution in the whole DRM receiver's processing benchmark1 to generate 11 MSC frames. Since the function distribution in generating the first frame differs from the following process, they are discussed separately. The profile report of generating the first frame is in Figure 4.4. From



MSC Frame ID	1	2	3	4	5	6	7	8	9	10	11
processing time (ms)	3347	380	449	391	388	443	389	383	447	389	392

Figure 4.2: MSC frame's processing time on Cortex A8

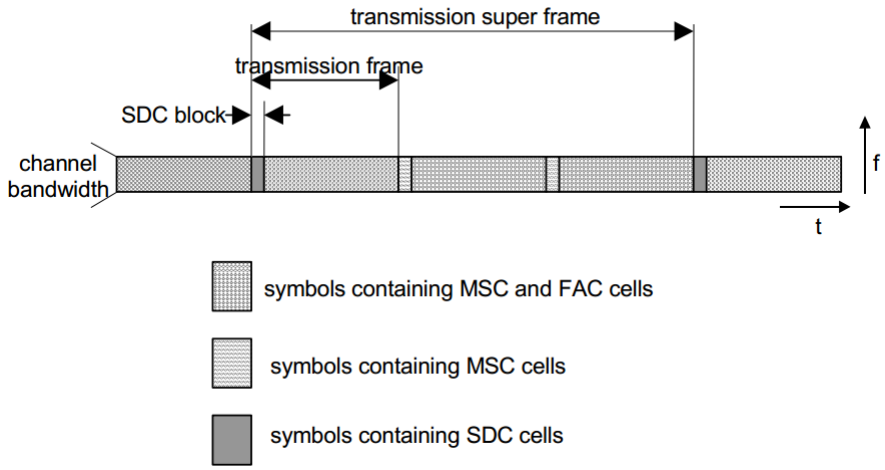


Figure 4.3: The frame structure of the transmitted signals [7]

the pie chart, it can be seen that the synchronization routine, Time Synchronization, takes more than half of the running time. The profile report of generating the 2nd to 11th frame is in Figure 4.5 where the distribution is rather different from Figure 4.4. The channel decoding function, MLC decoder takes 55% of the execution time. Among the MLC decoder, the processing function, Viterbi decoder occupies 91% of the decoder and the Viterbi decoder holds 50% of the whole processing time. Although synchronization takes a lot of time in the first frame, it only results in the latency of the start of the service which we are not interested in. Attention should be paid to the Viterbi decoder which is considered as a hot point needing to be optimized.

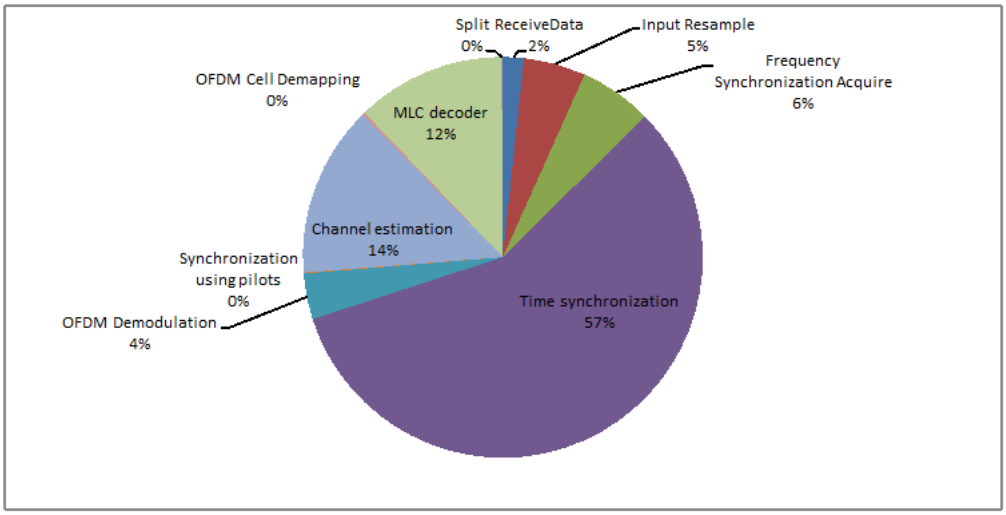


Figure 4.4: Profile report of processing the first frame on Cortex A8

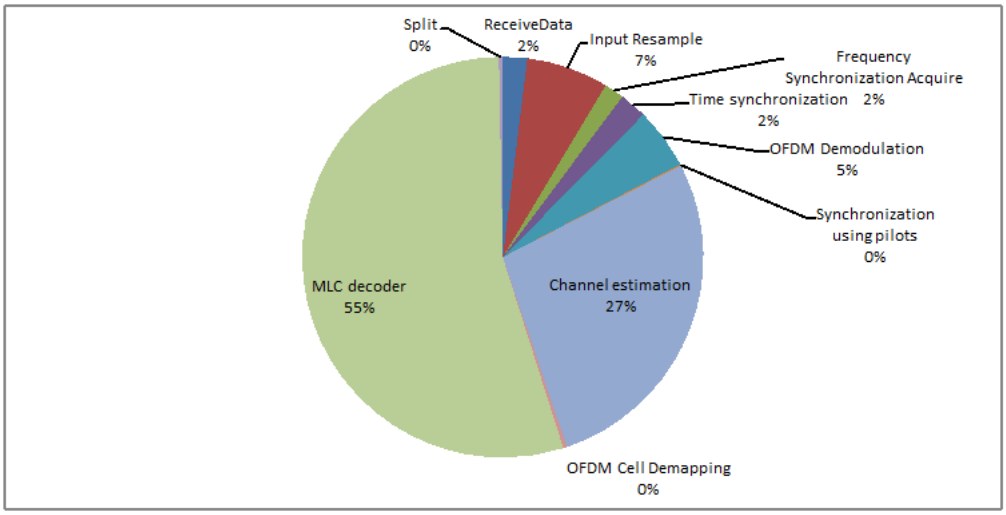


Figure 4.5: Profile report of processing the 2nd to 11th frame on Cortex A8

4.2 Comparison of Cortex A8 and Cortex R4

After the intermediate step, the processor Cortex R4 on M7400 must be compared with the Cortex A8 fast model to see whether a similar timing performance can be delivered. The ARM Cortex A8 core is designed for user applications with full-featured OS and it results in a much higher requirement on hardware comparing with Cortex R4, planned for deeply embedded systems with RTOS as shown in 3.1.

Considering the working frequency, the frequency of Cortex A8 core is higher than the

Cortex R4 processor. Normally, it works at the frequency of 800 Mhz. On the other hand, the frequency of the Cortex R4 processor on the M7400 platform is 416 Mhz.

For the floating point unit, the Cortex A8 contains a VFPLite co-processor which is an implementation of the ARM Vector Floating Point v3 (VFPv3) architecture with 32 double-precision registers [29]. In the Cortex R4 processor on the M7400 real hardware platform, there is no FPU available. Therefore, the floating point library is applied when processing floating point data types. It can be estimated that if the DRM receiver is ported on the Cortex R4 without FPU, the execution time should be close to, or even slower than the processing time of the Cortex A8 using the floating point library shown in Table 4.1. The execution speed on the Cortex R4 is unacceptable and it obviously cannot achieve a real-time service. However, in the provided System-C platform model, the Cortex R4 processor is configurable. It can enable a FPU in the virtual hardware. After configuration, there can be a VFPv3-D16 in the same architecture as VFPLite but only with 16 double-precision registers [20].

Since most part of the DRM code operates on floating point types, attention should be paid in Cortex R4's speed in processing floating point types. The single precision and double precision Whetstone benchmarks are used as before in section 4.1. And the result is compared with that of the Cortex A8 shown in Table 4.5. The processing ability of single and double floating point in Cortex R4 is weaker than Cortex A8. And unlike Cortex A8, the Cortex R4's ability in processing single-precision is obviously stronger than double-precision. Therefore, a data type conversion from single-precision to double-precision, is necessary for the program implemented on Cortex R4.

Table 4.5: The Whetstone benchmark result on Cortex A8 RTSM and Cortex R4 model

	Single-precision Whetstone (MIPS)	**Double-precision Whetstone (MIPS)**
Cortex R4	147.9	123.8
Cortex A8	153.4	148.8

What's more, the Cortex A8 architecture contains a NEON co-processor which further enhances the processing ability. This SIMD technology is not applied in Cortex R4.

A signal processing kernel speed testing is presented in [30] using Certified BDTI DSP Kernel Benchmarks. It shows that the Cortex A8 working at 450 Mhz delivers an almost three times the performance of the Cortex R4 of 300 Mhz.

As far as the comparison achievement is concerned, the receiver mapping on Cortex R4 cannot provide a similar timing performance as on Cortex A8. Therefore, some modifications must be done to the program to speed up. The optimization of the program is presented in next section 4.3.1.

4.3 Porting the DRM receiver program on Cortex R4

4.3.1 Modifications of the DRM receiver program

On the basis of the conclusion in 4.2, the code must be modified to speed up the receiver.

As mentioned in 4.1, the widely used double precision floating type is the essential reason of slowing down the ARM execution. There are two options for data type conversion. First is to transfer double precision floating point to integer, which is not realistic because it would have a lot of work including debugging overflow and changing to an integer fftw library. Another option is to change double precision floating point into single precision floating point. This would also enhance the performance based on the processor's result shown in Table 4.5. Furthermore, the problems caused by converting into integer do not appear. Thus the latter option is chosen in this study.

The time expensive part, Viterbi decoder function also needs to be optimized. The Viterbi decoder is re-written in integer and a scaling processing is done to the soft-decision inputs. The working principle and modification details of the Viterbi decoder is discussed in 5.2.

4.3.2 Profile results on Cortex R4

For porting on the Cortex A8 model, the DS-5 software will automatically link the ARM image to the processor and begin the execution. However, the provided platform model is written in System-C and is launched in the software, Synopsys Virtual Prototype Analyzer G-2012.06-SP2. The boot file has to be written using the ARM Linker to load the application image to the processor's memory. The details of the ARM linker configuration is presented in Appendix A.

Since the Cortex A8 model contains the Linux operation system, it is easy to control the input and output of the data stream to read/write from/to files with the help of the operation system. However, the Cortex R4 System-C model is built without any operation system. Semihosting [31] is implemented here to communicate input/output requests from application code to the host running a debugger which is the Synopsys software here. The requests includes keyboard input, screen output, and disk I/O. When a request occurs, the application invokes the appropriate Software Interrupt (SWI) and then the debug agent handles the SWI exception and communicate with the host. After that, the host responses to the debug agent's communication information to export/import the data stream.

The modified DRM receiver program is successfully ported on the Cortex R4 core of the platform model based on the configuration as discussed before. As mentioned in 4.2, the core model setting has to be changed to enable VFU and the compile option

–fpu=vfpv3_d16 is used to configure the compiler to generate floating point hardware operations for vfpv3_d16.

A functional check of the modified code is achieved as it is done in Cortex A8 model in section 4.1. And the results can be seen in Figure 4.6. Comparing with Figure 4.1, the error rate is slightly increased, but still acceptable.

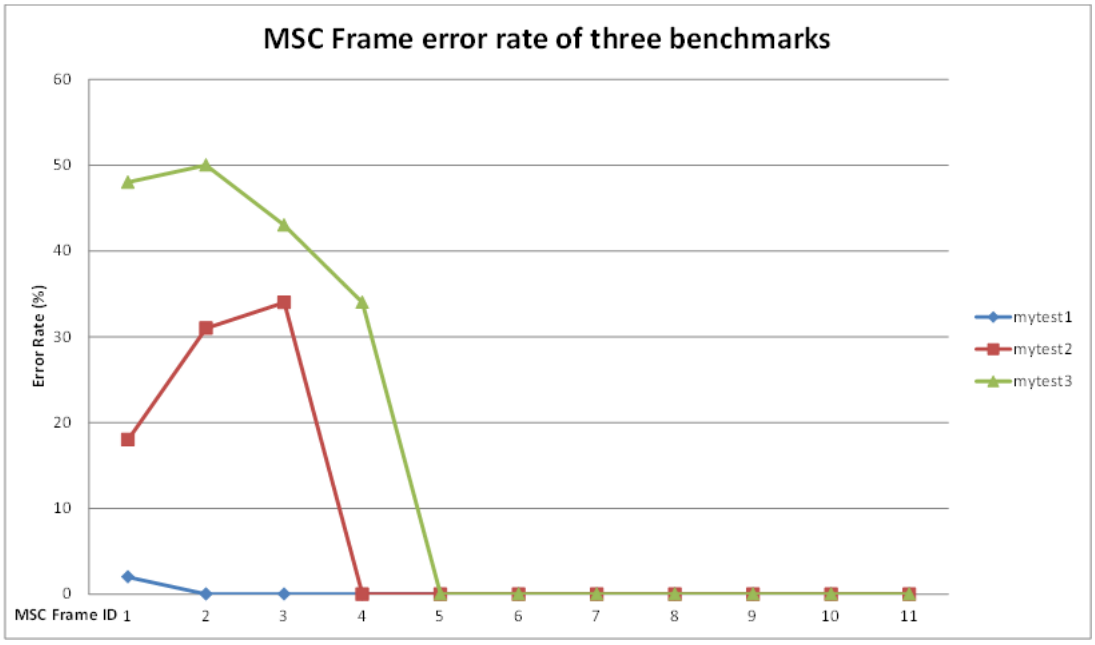


Figure 4.6: The modified program running on Cortex R4's MSC frame error rate of three benchmarks

The software, Virtual Prototype Analyzer automatically provides a function traced profile report. It is used to measure the time in all timing results of the M7400 platform.

A similar timing performance measurement is finished as in Cortex A8 model using the benchmark 1. The time interval of generating each MSC frame is measured in Figure 4.7. Comparing with Figure 4.2, the processing time is larger than that of the Cortex A8. Though some modifications are already applied to the source code to speed up, it still cannot achieve a real time service since the processing time of each frame is above 400 ms.

Profile reports based on benchmark 1 are achieved as it is done on Cortex A8 for the research on the function distribution on generating 1st MSC frame and generating 2nd frame to 11th frame respectively and the results are presented in Figure 4.8 and Figure 4.9. The hot spot Viterbi decoder in Cortex A8 version is still the most time expensive part which takes 78% of the MLC decoding time and 45% of the whole running time in the period of processing 2nd to 11th frame. Therefore, in order to increase the performance,

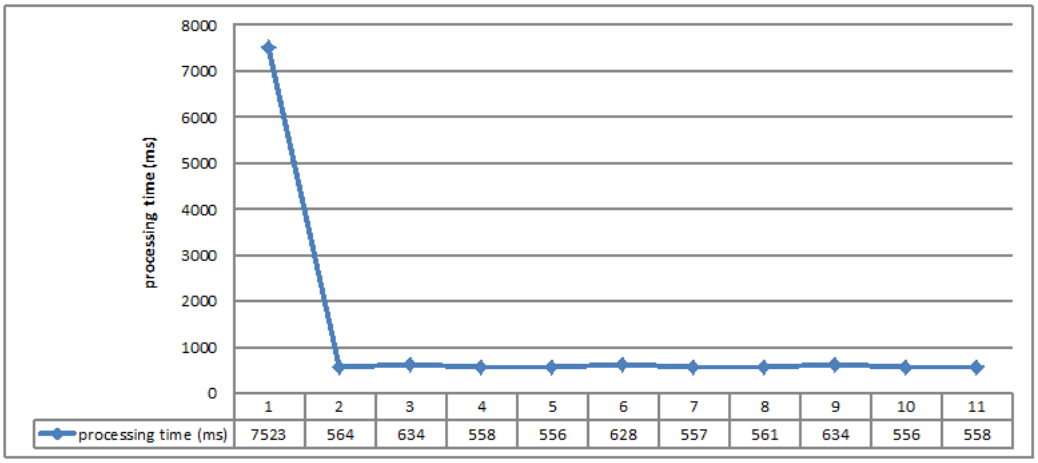


Figure 4.7: MSC frame's processing time on Cortex R4

attention should be paid to Viterbi decoder to further decrease the execution time.

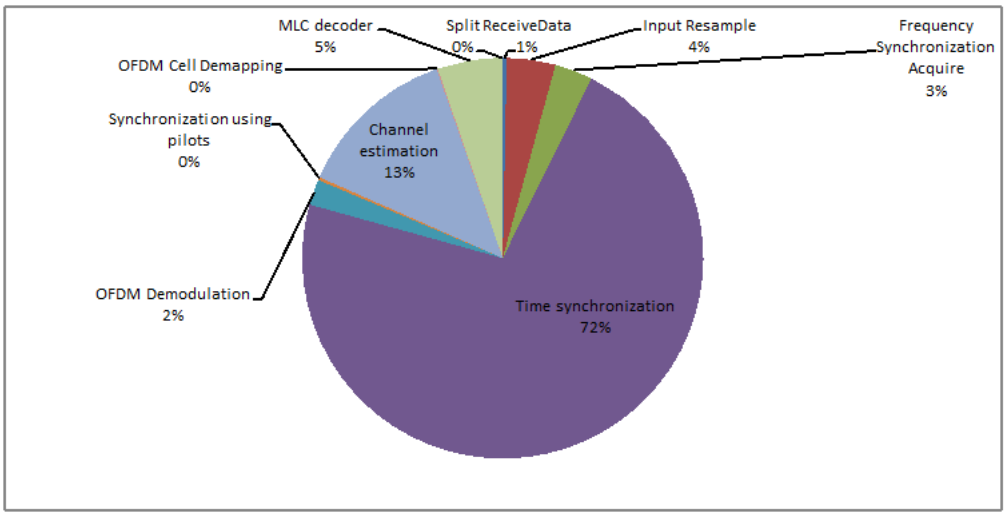


Figure 4.8: Profile report of processing the first frame on Cortex R4

In this chapter, an intermediate step of porting the DRM program on the Cortex A8 is done at first. Then the program is modified in order to speed up the execution. However, the details of the optimization on the Viterbi decoder is not shown in this chapter. Finally it shows the profile report of the optimized DRM program on the Cortex R4 model. The optimized single core version still cannot reach the real-time requirement in the Cortex R4. Therefore, a multi-core version should be achieved in the following chapters. Furthermore, in the next chapter, the detailed modification to the Viterbi decoder of the single-core version is also discussed in section 5.2 after the

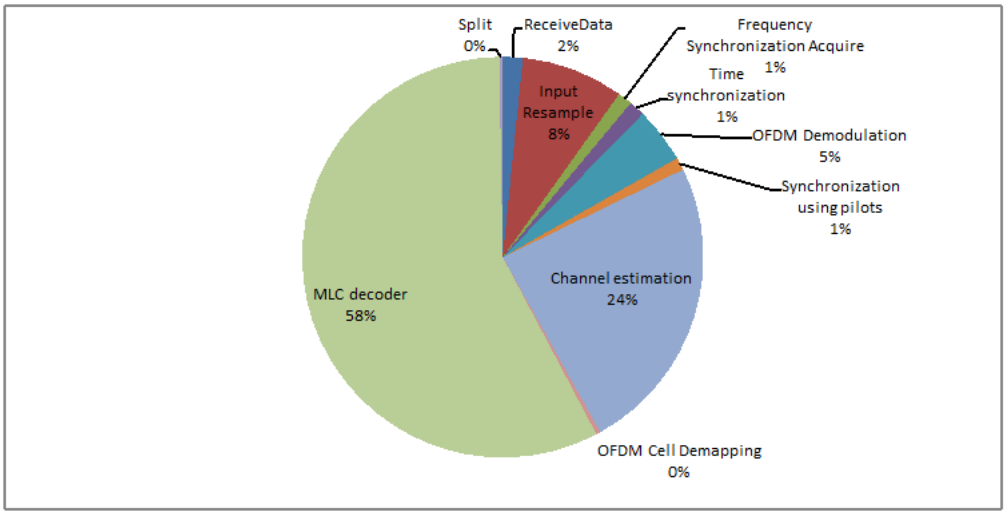


Figure 4.9: Profile report of processing the 2nd to 11th frame on Cortex R4

introduction the working principle of Viterbi decoder in section 5.1.

Chapter 5

Analysis and Realization of the multi-core version

According to the conclusion in section 4.3.2, the Viterbi decoder slows down the whole processing speed. Therefore, in the multi-core version, this part is a candidate to port on EVP. However it still needs to be considered if its related functions also have to be ported on another core. Thus a discussion of MLC and its belonging Viterbi decoder's working principles must be done in section 5.1. Then upon the Viterbi decoding's working principle, section 5.2 shows the detailed modification of the Viterbi decoder of the ARM single core version as discussed in section 4.3.1. The tool Pareon is used in analysis of the multi-core version of the program and it will be discussed in section 5.3. Based on the analysis result, section 5.4 presents the realization of the ported code on EVP.

5.1 MLC and Viterbi Decoder's working principles

5.1.1 MLC encoder and decoder

In the transmitter end, the MLC encoding is applied in the channel encoding process. In MLC, the source bit stream is partitioned into different streams. Each stream goes through the encoding processes respectively and then converges together to do QAM (quadrature amplitude modulation).

The processing flow is shown in figure 5.1. The source information is encoded into bit stream before importing into the MLC routine. In the MLC encoding processing stage, the Energy Dispersal is the first processing function. The purpose of this function is to avoid the transmission of signal patterns which might result in an unwanted regularity in the transmitted signal [7]. A scrambler is used here to convert the bit stream into a seemingly random bit stream of the same length by using a pseudo-random binary sequence (PRBS), thus avoiding long sequences of bits of the same value.

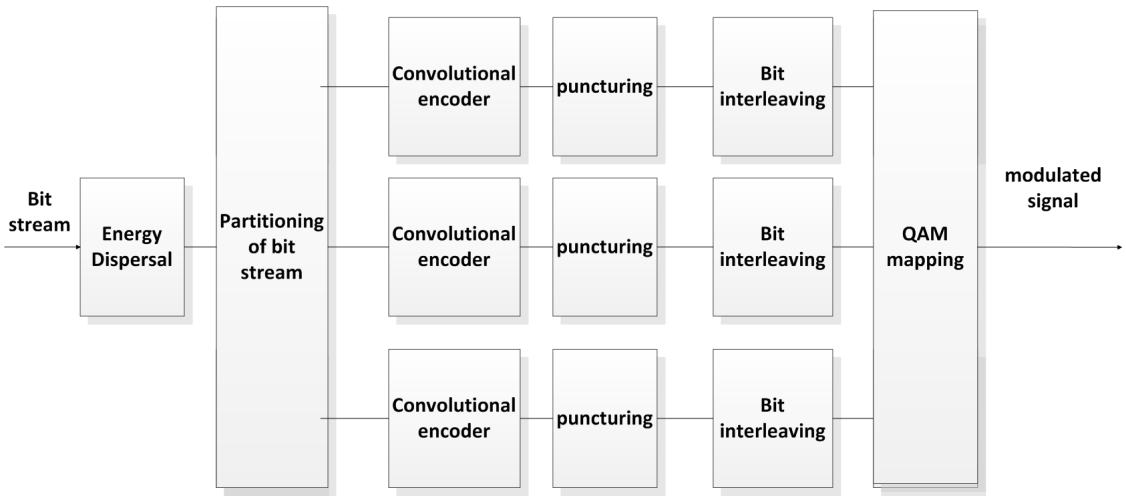


Figure 5.1: The processing flow of 3 level MLC encoder in the DRM transmitter

Then according to the robustness mode setting of the transmission, the bit stream is divided into n levels in the Partitioning function. n ranges from 1 to 3.

Each partitioned bit stream is encoded separately. The convolutional coding with a original code rate 1/4 and constraint length 7 is applied. The codeword is defined as the equation 5.1. The structure of convolutional encoder can be seen in figure 5.2 from [7]. Every 1 bit inputs into the encoder, it will generate 4 bits output. Therefore, the code rate is 1/4. There are 6 shift registers in the decoder, as a result 7 bits in the encoder are involved in calculating the output bits. So the constraint length of the encoder is 7.

$$\begin{aligned} b_{0,i} &= a_i \bigoplus a_{i-2} \bigoplus a_{i-3} \bigoplus a_{i-5} \bigoplus a_{i-6} \\ b_{1,i} &= a_i \bigoplus a_{i-1} \bigoplus a_{i-2} \bigoplus a_{i-3} \bigoplus a_{i-6} \\ b_{2,i} &= a_i \bigoplus a_{i-1} \bigoplus a_{i-4} \bigoplus a_{i-6} \\ b_{3,i} &= a_i \bigoplus a_{i-2} \bigoplus a_{i-3} \bigoplus a_{i-5} \bigoplus a_{i-6} \end{aligned} \tag{5.1}$$

However, if the original encoder's code rate is directly put into use, a large amount of output bits are generated and it gives a significantly large burden on the transmitter and receiver. Therefore, a puncturing processing to output bits is necessary in order to decrease the code rate. Various puncturing patterns are stored and applied to the convolutional output bits to remove some of the parity bits without influencing the error-correction ability.

Bit-wise interleaving shall be applied for some of the streams. The punctured bits are interleaved according to the corresponding interleaving table.

Finally, the divided bit streams are converged in the QAM function. In QAM, the constellation points are usually arranged in a square grid with equal vertical and horizontal

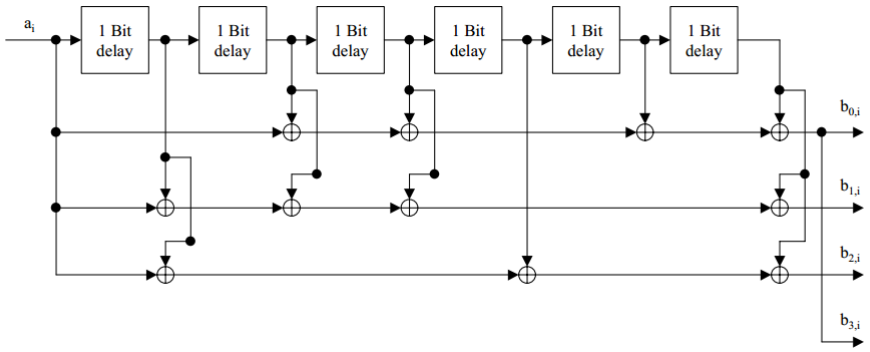


Figure 5.2: The structure of convolutional encoder in the DRM transmitter [7]

spacing and the number of points in the grid is usually in a power of 2. In this module, the number of point is decided by the bit stream level n. The point number equals 4^n, therefore 4-QAM, 16-QAM and 64-QAM are implemented here. In each mapping, 2 bits are extracted from each divided stream to act as the vertical and horizontal coordinates. Since the stream number n matches the QAM size, the extraction bits of n streams are suitable to process a mapping at the constellation which can be seen in Figure 5.3.

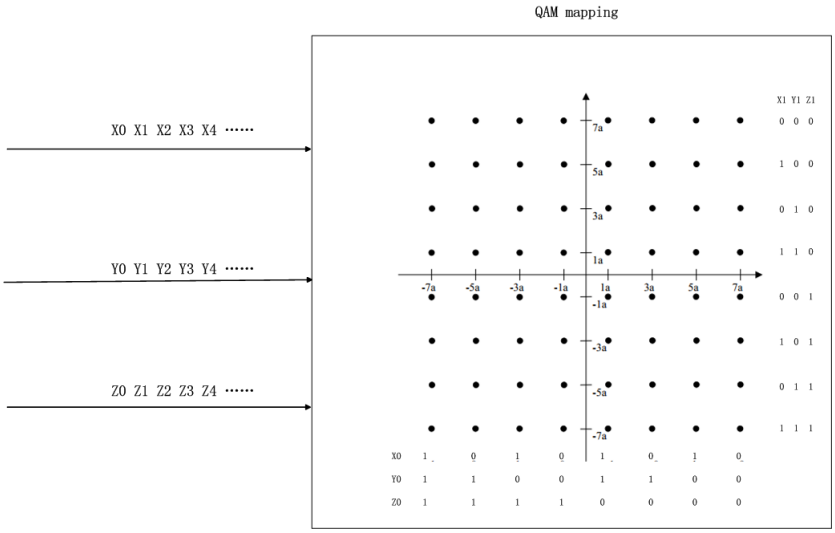


Figure 5.3: The multi level QAM mapping working principle

Based on the channel encoding process, a corresponding channel decoder is built in the receiver end. The MLC decoding is also named multistage decoding (MSD). The decoding is an iterative process which means the lower level decoding result will be of help to decoding the higher level bits as seen in Figure 5.4.

The data flow structure of MLC decoding can be seen in Figure 5.5. One decoding routine

includes QAM de-mapping, Bit de-interleaving, De-puncturing and Viterbi Decoder. After decoding ,the decoded bits are imported into the encoding routine which consists of the same processing functions as in the encoder of the transmitter. The encoded bits are iteratively imported back to the QAM de-mapping function and work as the determined bits to help estimate the bits of the next level.

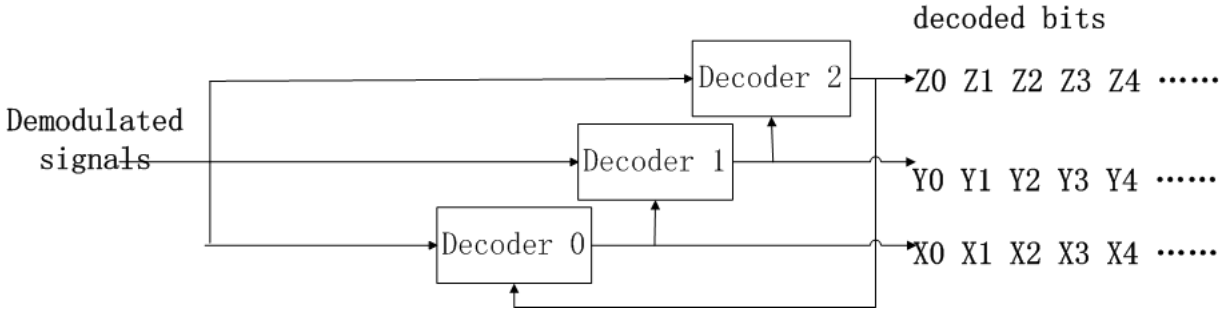


Figure 5.4: The iterative decoding process of MLC decoder

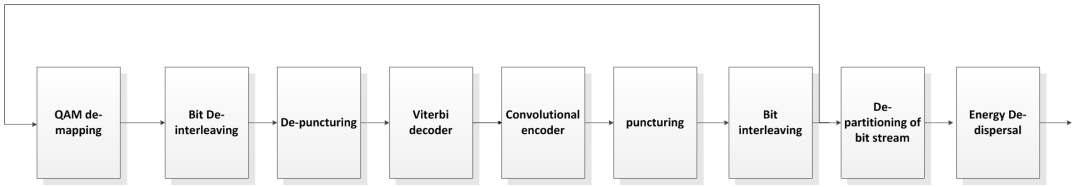


Figure 5.5: The processing flow of MLC decoder in the DRM receiver

There is an iterative number parameter that effects the decoding times of each output bit. In the DRM receiver, the default setting of this parameter in the MLC decoder is 2 which is applicable to 2 levels and 3 levels MLC decoder. For instance, in 2 level MLC decoder, 2 data streams X and Y are decoded. The X stream is firstly decoded and then the X result helps in decoding the Y stream. This is the first iteration. In the second iteration, the Y stream result from the last iteration, helps the re-decoding work of X, and the final result of X stream is obtained. The final result of X stream helps decode the Y stream. However, as to the 1 level MLC decoding, since it only contains one bit stream, even if it iteratively inputs into the decoder, it cannot be helpful in the 4-QAM de-mapping. The output bit of the 1 level MLC decoder is decoded only once.

5.1.2 Viterbi Decoder

Among all the processing functions in the MLC decoder, the most time-expensive part is Viterbi Decoder. A detailed working principle of the Viterbi decoder is discussed below.

There are two parts in the decoder, namely trellis construction and trace back. In the

decoder, Viterbi Algorithm (VA) is used to find a maximum likelihood (ML) estimation of a transmitted code sequence c from the received sequence r by maximizing the conditional probability $P(r|c)$.

In the trellis construction, the Branch Metric (BM) is used to present the conditional probability of one encoding branch based on the received encoded bit sequence. Two methods can be used to present the received encoded bit, namely Hard-Decision and Soft-Decision. In the Hard-Decision, each received encoded bit is presented by 1-bit quantized decoder input. Then the BM of a branch is obtained by calculating the Hamming Distance between the decoder inputs and the encoding bits of the branch. Hamming Distance stands for the number of differing bits between two bit sequence. The smaller BM indicates that the the decoder inputs is more closed to the encoding bits of the branch, thus the possibility of this branch is higher. While for Soft-Decision, it is basically the same as the Hard-Decision, but it improves the decoding quality by using multi-bit quantized inputs to present each received encoded bit.

The Soft-Decision is applied in the Viterbi decoder of DRM. The multi-bit quantized input is calculated by multiplying the distance of the QAM constellation which is gotten from QAM de-mapping function and the magnitude of the channel achieved by the channel estimation stage. Since puncturing is used, the puncturing pattern should also be considered. A table containing the puncturing pattern for each decoding bit is created before the MLC decoding routine. There are 6 types of puncturing patterns, namely 0000, 1111, 0111, 0011, 0001, 0101. 0 stands for un-punctured bit and 1 is for punctured bit. For example, 0000 pattern means all 4 bits are not punctured, thus 4 inputs are involved in calculating BM. From this analogy, if the puncturing pattern is 0001, 3 inputs are used to decide BM. Thus based on the multi-bit quantized input and the puncturing pattern, the difference between the received bits and every possible branch's encoder bits can be measured out.

Path Metric (PM) is the sum of the corresponding BMs. If two branches merge at a state, the smallest one is chosen to survive. A whole process of trellis construction can be seen in Figure 5.6. It is a decoder with the code rate 1/3, constraint length 3 and the Hard-Decision is applied. The states of 00, 01, 10, 11 present the bits store in the encoder register. Each state has two branches. The solid line branch is for the encoder input bit 0 and the dotted line branch shows the encoder input bit 1. The bits on the branch presents the corresponding encoder output. The received bits' (decoder input bits) HB with every branch are computed to obtain the BMs shown in red italic. PM is located in every state shown in black bold. If two branches merge at a state, the smallest one is chosen to survive as it is in state 00 in step 3. If choosing the branch from state 00, the resulting PM is 5. However, if choosing the branch from state 10, PM is 1. Therefore, the latter path is selected and PM is calculated as 1. The path decision is stored in the decision bit.

Therefore, in every trellis construction step, all the BMs are calculated and added with the corresponding PM. The confluent branches are compared to remain the smaller one

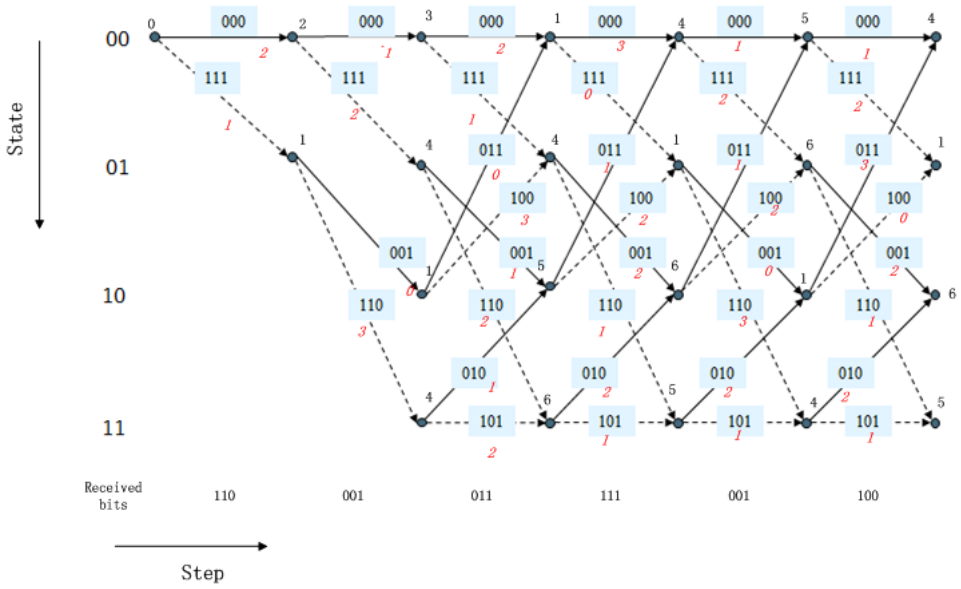


Figure 5.6: The processing of the trellis construction

and the decision bit is stored. This process results in an high computational load. In the DRM Viterbi decoder, the constraint length is 7. Based on this, a computation load is listed in Table 5.1. The Viterbi decoder's trellis step number n is equal to the number of the decoding output bits. In the implementation, normally the trellis step can be up to 4000, which leads to a significantly large number of calculation.

Table 5.1: The computation load of the n-step Viterbi decoder of DRM

Addition operation	**Comparison operation**	**Store operation**
$128 \times n$	$64 \times n$	$64 \times n$

Since the PM is a result of thousands times of repeated additions, the value may be extremely large which needs a large register to store. However, if the hardware is limited, some methods must be done to prevent the overflow. Four normalization methods are listed in [32], Reset Metric Normalization, Variable Shift Metric Normalization, Fixed Shift Metric Normalization, and Modulo Metric Normalization. Normalization is not easy to implement and it will slow down the processing speed due to its periodic checking overflow. But it has an obvious advantage that it will not decrease the decoding accuracy because it is the difference values between PMs that decides the decoding output rather than the absolute values. An alternative option is scaling. If the maximum number of steps is already known, based on the max steps n and register width, a scaling operation can be done to the BMs to ensure that even if the maximum BM value is repeatedly added n times, the accumulate value should not overflow in the register. The advantage of scaling is that the implementation is easy and it won't influence the decoding speed

because the scaling is processed before the Viterbi decoding. However, the scaling will influence the accuracy of the decoding, since the scaling of BM will decrease or increase the difference values between PMs.

In the trace back part, the most likely sequence is reconstructed by going back in steps and the processing can be seen in Figure 5.7. There are two methods in trace back. The first approach is sliding window. The timing relation between the Trellis construction and the trace back can be seen in Figure 5.8. One time trace back of a constant length D_{back} occurs periodically and the period is a certain number $D_{forward}$ trellis construction steps. In trace back, the smallest PM is chosen as the starting point and construct a path backward in the length of D_{back} based on the decision bits. At every turn of trace back, D_{update} decoding bits are exported ($D_{update} = D_{forward}$). Furthermore, in practical implementation, the trellis construction can execute in parallel with the trace back.

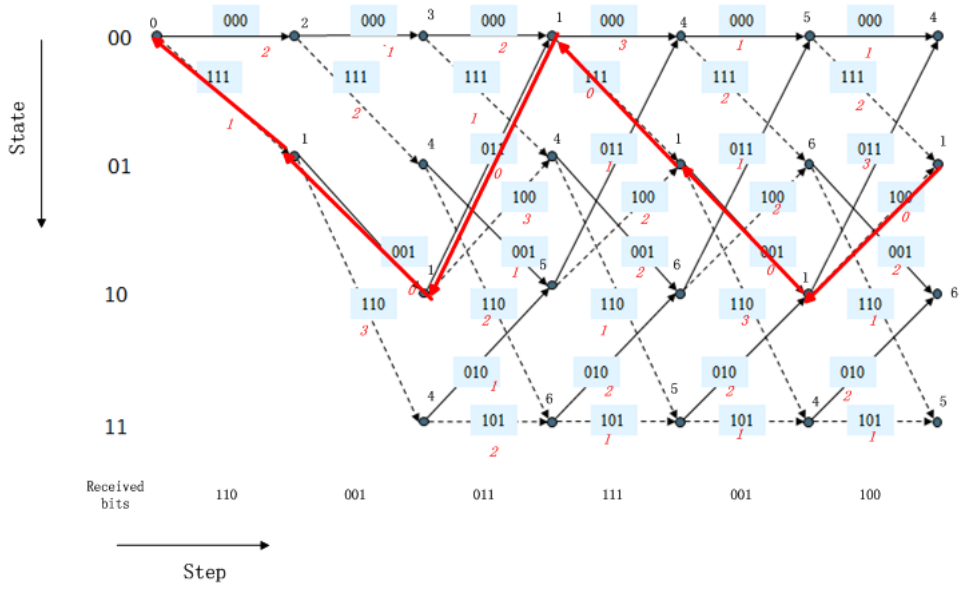


Figure 5.7: The processing of the trace back

A second approach uses frames. With frames, the dependency chain is the length of the frame [33]. In the encoding, the input bits of a frame is ended with a sequence of 0. Trace back only happens once when the trellis construction of the whole frame ends. A trace back from the ending to the beginning of trellis steps is done to decode the whole bits in one time. And the starting point is chosen in the state with all 0, since the ending of the frame is a sequence of 0.

Comparing these two approaches, the sliding window method uses less memory. The decision bits of the already decoded bits do not need to be stored in the memory. Another advantage is that it can process faster because of the parallel execution of trellis

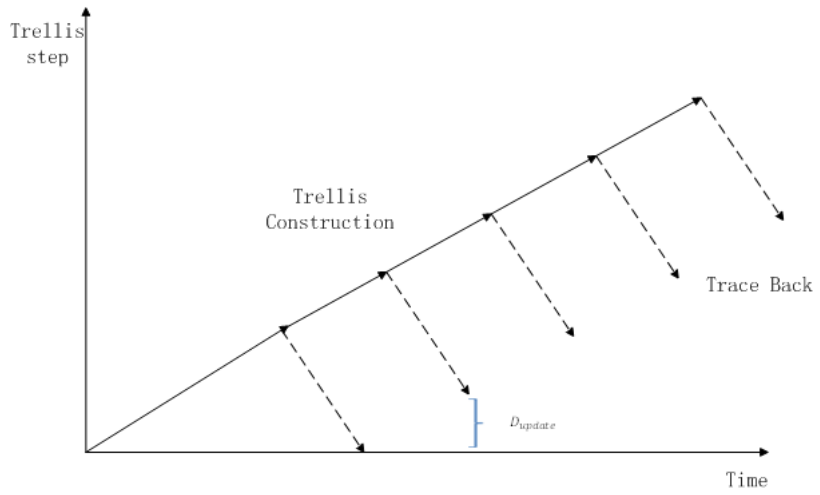


Figure 5.8: The timing relation between Trellis construction and trace back of sliding window

construction and trace back. However, the speed advantage is not obvious because the time cost of the trace back is greatly smaller then that of the trellis construction. While the frame based trace back method delivers a higher error correction decoding because the multi-time trace back will inevitably cause error. In the sliding window, the trace back path is built based on the current constructed trellis but if the unestablished trellis are taken into consideration, a rather different path may be built.

5.2 Modification on the Viterbi decoder

Section 4.3.1 presents that the variables of the whole program are converted from 64-bit double to 32-bit float at first. Then the Viterbi decoder function is optimized in the ARM Cortex R4 single-core version. However, the detailed optimization steps in Viterbi decoder are not shown. In this section, we work on the 32-bit float version program and

the data type conversion and scaling process are discussed combining with the working principle shown in section 5.1.

To decrease the execution time of the Viterbi decoder on Cortex R4, the first step is to rewrite the function from 32-bit float to 32-bit int. As mentioned in 5.1, the relative values of PMs decide the decoding results, therefore a type conversion on BMs from single precision floating point type to integer will not significantly decrease the decoding accuracy. The conversion is achieved by using $V_{int} = floor(V_{float} + 0.5)$ to round the decimals to the nearest integral number. A result can be seen in Figure 5.9 testing on the PC version of the DRM receiver to observe the type conversion's influence on the receiver's MSC outputs.

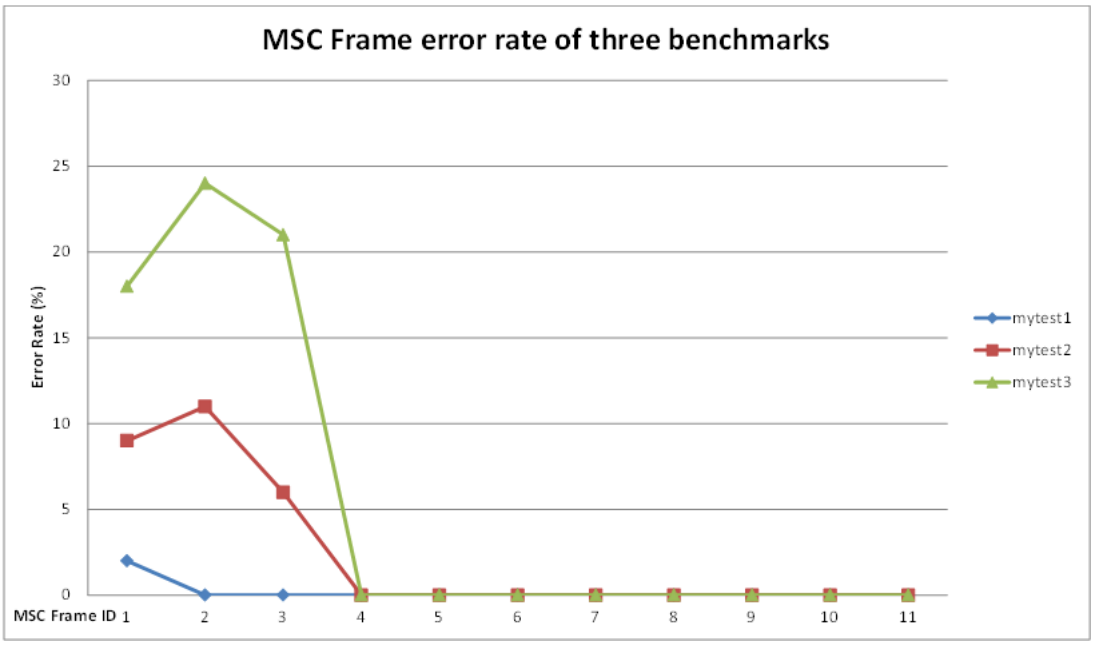


Figure 5.9: MSC frame error rate tested by three benchmarks of the single precision floating type version program with integer Viterbi Decoder

Based on the integer Viterbi decoder version, a scaling to the soft-decisions is further achieved to prepare porting the Viterbi decoder code on EVP. When it is implemented on EVP, an overflow problem may occur. The range of BM should be limited to prevent its accumulation result, namely PM over the range. In the integer conversion process, the BMs are converted from the 32-bit float to 32-bit int. However, the soft-decision inputs are still in 32-bit float to guarantee the precision of the decoder. It is chosen to scale the soft-decision input before de-puncturing processing. Since BM is calculated by soft-decision inputs, the value of BM can be controlled when the value of the soft-decision is restricted.

Before de-puncturing, the soft-decision inputs for a single Viterbi decoding routine are

searched to find the max value I_{max}. If it is bigger than the S_{max}, the scaling factor is calculated by $S_{factor} = I_{max}/S_{max}$ and then the scaling factor is used to scale the input value by $I_{scale} = I/S_{factor}$. Otherwise, inputs are unprocessed. Since scaling will decrease the accuracy as shown in 5.1, it is necessary to find a proper scaling degree. A similar testing on the MSC error rate in different S_{max} is finished and the result can be seen in Table 5.2. The used benchmark is mytest2 because it is in the lowest protection, Robustness Mode A. If this benchmark can be decoded with low error rate, others can also be processed successfully. It can be seen when S_{max} is decreased to 2, the error rate is greatly increased and the error extents to the latter frames. The calculation result for EVP shown in 5.4 presents that when S_{max} is 3, the PM will not overflow in a 16-bit register. Therefore, S_{max} is chosen as 3.

Table 5.2: The single precision floating type version, integer Viterbi decoding DRM receiver's MSC error rate in different S_{max}

MSC Frame ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
not applied (error rate %)	9	11	6	0	0	0	0	0	0	0	0
S_{max} 200 (error rate %)	9	11	6	0	0	0	0	0	0	0	0
S_{max} 100 (error rate %)	9	12	6	0	0	0	0	0	0	0	0
S_{max} 50 (error rate %)	8	11	4	0	0	0	0	0	0	0	0
S_{max} 10 (error rate %)	15	20	12	0	0	0	0	0	0	0	0
S_{max} 5 (error rate %)	18	31	21	0	0	0	0	0	0	0	0
S_{max} 3 (error rate %)	18	31	33	0	0	0	0	0	0	0	0
S_{max} 2 (error rate %)	25	39	47	1	5	14	10	36	0	0	11

Based on these result, we can concluded that the optimized Viterbi decoder in the ARM Cortex R4 single-core version is accomplished and it is also well suitable for porting on EVP.

5.3 Pareon's analysis in the multi-core version

In the profile report of the DRM program on Cortex R4 shown in section 4.3.2, the Viterbi decoding function takes 45% of the processing time. Therefore, this part is suitable to port on EVP since it takes almost half of the execution time. From the section 5.1, it can be seen that the trellis construction processing has a high computational load but the construction process is simple and repetitive. There are 128 addition operations, 64 comparison operations and 64 store operations to generate new PMs in each construction step. However, each PM's generation pattern is static and simple, which makes it easy to vectorize the processing in each trellis construction step. From the vectorization view, the Viterbi decoding function is suitable to be rewriten in EVP-C and be implemented on EVP.

Though it is already known that the Viterbi decoder function is suitable to be ported on EVP, which is the same as the conclusion in [8], there are still some remaining topics of the multi-core version needed to discuss. It remains to be see whether the EVP related parts also have to be ported on EVP. The multi-core analysis should also consider timing performance, communication data size, EVP-C realization efficiency.

If the DRM application is not changed and only the Viterbi decoder is ported on EVP, there is an obvious disadvantage that it cannot achieve pipelined processing. The ARM core has to wait until the work of the EVP is finished. Only until Viterbi decoder results are transferred back from the EVP, the ARM core can continue do the following MLC decoding processing, which can be seen in Figure 5.10. From the frame processing view shown in Figure 5.11, it means that only when the current frame processing is finished, the DRM can step into the processing of the next frame.

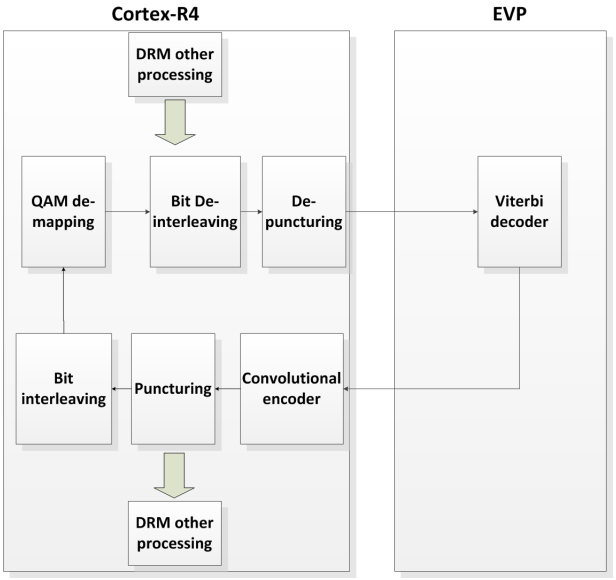


Figure 5.10: Non-pipelined multi-core version

If a pipelined processing is expected, a review of the DRM receiver processing data flow in Figure 2.4 should be done. The Viterbi decoder function is in the MLC stage. And the MLC processing is the main routine of the channel decoding and it is applied to 3 different information channels, FAC, SDC, MSC separately. After the channel decoding, the decoded information of the FAC and SDC have to be utilized to change the configuration parameters of the receiver, but the MSC decoded result is not used to alter the setting. Therefore, if the MSC channel decoding processing is totally ported on EVP, it can achieve pipelined processing as shown in Figure 5.12. The channel decoding includes processing stages of De-interleave, MLC decoder, De-multiplexer and Split MSC. From the frame processing view shown in Figure 5.13, when the MSC channel decoding task is sent to EVP, the ARM core can begin to handle the demodulation of the

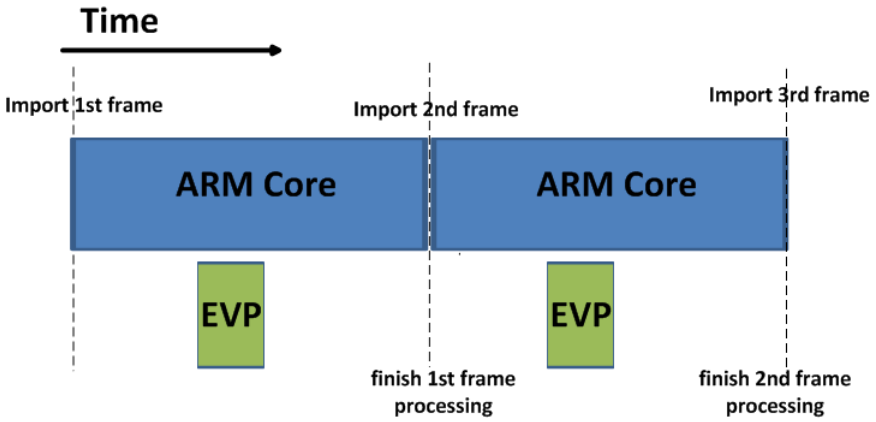


Figure 5.11: Non-pipelined frame processing

next frame rather than waiting the end of the EVP task. Though the channel decoding stage of SDC and FAC still performs on ARM, it is still expected to provide a great improvement on the time performance since the MSC decoding workload is much higher than FAC and SDC, for instance, in the benchmark, mytest1(MSC frame size is 6984, SDC is 630 and FAC is 72) and the MSC channel decoding takes more than 90% of the total channel decoding time.

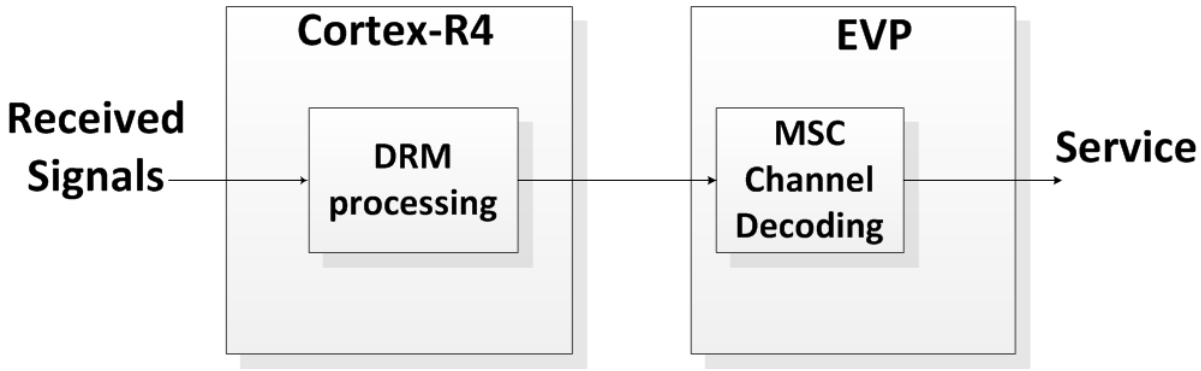


Figure 5.12: Pipelined multi-core version

Porting the whole MSC channel decoding processing on EVP may deliver a high performance and pipelined processing. However, if considering the practical realization, it is not a good choice. There are a large amount of complex functions in De-interleave, MLC decoder, De-multiplexer and Split MSC and the only vectorizable function is the Viterbi decoding of the MLC. And in the processing stage of De-interleave and some parts in MLC, the processing operates on floating point variables, which must be rewrite for EVP. In summary, it is hard to rewrite the whole code in MLC channel decoding in EVP-C to generate an efficient decoding implementation on EVP. The channel de-

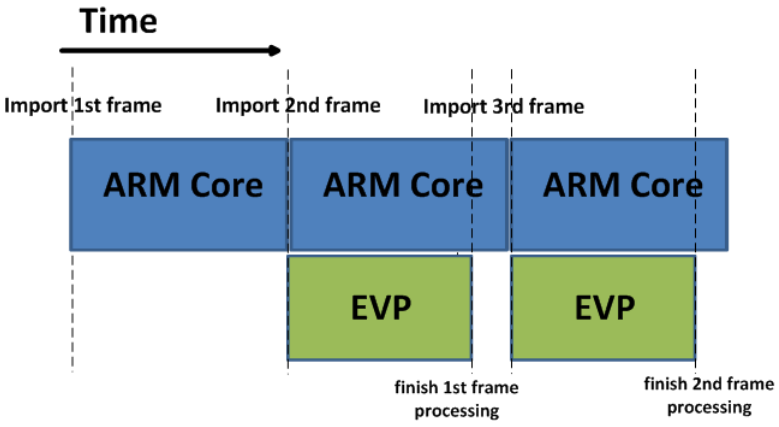


Figure 5.13: Pipelined frame processing

coding's slow execution on EVP may even eliminate the acceleration benefited by the pipelined processing. When the ARM core finishes the works of the next frame, it may have to wait for the EVP to finish the current frame's decoding task.

An alternative option is to port the De-puncturing function into EVP, because this processing function is closely related to the Viterbi decoder function. In this function, the BMs are achieved by calculating the soft-decision inputs, combining the puncturing table. This function only takes a small amount of execution time, just 9.2% of the Viterbi function based on the Cortex R4 optimized program's result. However, the EVP-porting of the De-puncturing function will influence the size of data transferred from the ARM to the EVP. Thus there are two versions, namely the De-puncturing, Viterbi decoder EVP version and Viterbi decoder EVP version. We have to compare two versions in the size of the communication data to be transferred from ARM to EVP and from EVP to ARM.

The Vector Fabrics Tool, Pareon is used here to compare the communication of two versions. This tool aims to help analyze the multi-core code. It is designed to find the dependency within program functions and give a guidance in optimizing the C/C++ code on the multi-core architecture including Intel processors and ARM cores. This tool is still under development, now it only supports the homogeneous multi-core system with shared memory. The functionality of support for the heterogeneous multi-core architecture and the simulation for the data transfer are not available yet. However, the current version of Pareon can still be used to analyze the multi-core version.

The program needs to be recompiled by the Vector Fabrics compiler toolchain **vfcc** and **vf++**. Then the Pareon can analyze the code after the program's execution. The GUI presents a function distribution as shown in Figure 5.14 and the loop number is also shown in the GUI. In the source code, the function **CViterbiDecoder::Decode** only contains De-puncturing and Viterbi decoder and the Viterbi decoder processing is in

CViterbiDecoder::Decode's subfunction **main_viterbi**. Therefore, the tool, Pareon is used to measure the data dependency between **CViterbiDecoder::Decode** and the high level MLC decoder's loop, **main_viterbi** and the high level MLC decoder's loop. In the practical use of the Pareon, the high level loop **Loop_20514** and **CViterbiDecoder::Decode** are dragged to the dependency task window and the memory dependency is chosen. The memory dependency is divided into two kinds in Pareon, namely inbound dependency and outbound dependency. The inbound dependency means dependencies that cross the start boundary of the invocation of the target function which can be treated as the transferring-in data. Similarly, the outbound dependency stands for the transferring-out data. Here the inbound dependency is chosen to measure, since it is already known that **CViterbiDecoder::Decode** and **main_viterbi**'s transferring-out data are the same, which is the decoded result bits. The Pareon uses transfer rate to present the transferring data amount and it is in the unit of $Mi\ reads/s$. Mi means 1×10^6. $read$ means one time access to the memory to read data. The transfer rate is calculated by the equation 5.2 in the Pareon tool. This equation is given by the Vector Fabrics company. N_m is the total memory load times related to the memory dependency. N_i indicates the target function invocation times. T_{ti} is the total running time of the target function in execution. One important factor is that the transfer rate only shows the number of access to the memory per second in average in the target function execution period. However, every time the program accesses to the memory to fetch a variable and the variable can be in a different size, 16-bit int, 32-bit float, 64-bit float, etc. So the transfer rate does not show exactly the size of data to be transferred. The result can be seen in Figure 5.15 and Figure 5.16. From the result, it can be seen that the total transfer rate is $6.1Mi\ load/s$. There are two main memory dependencies. One is in Figure 5.15 $4.7Mi\ load/s$. The detailed content of the dependency is observed, it includes the puncturing table and the outside defined vector structure and function. In the program, the puncturing table is presented in the variable of **veciTablePuncPat** while the vector related dependency is more complex. The variables are stored and operated in the defined vector structure and function and these vector structures and functions are defined outside **CViterbiDecoder::Decode**. The tool Pareon views them as the memory dependency. Another memory dependency is $1.9Mi\ load/s$, which is related to the soft-decision input **vecNewDistance**.

$$\begin{aligned} transfer\ rate &= \frac{N_m/N_i}{T_{ti}/N_i} \\ &= \frac{N_m}{T_{ti}} \end{aligned} \tag{5.2}$$

After finishing the data dependency analysis of **CViterbiDecoder::Decode**, **main_viterbi** is analyzed in the same way and the result can be seen in Figure 5.17. Since the scaling processing results in the value of BM less than 12, 4 BM variables are packed into a 16-bit int in practical communication. The packing process is also applied in the program under analysis to help the Pareon to deliver a proper result. The total transfer rate is $2Miload/s$ and the main dependency shown in Figure 5.17 is $1.9Mi\ load/s$ which is the

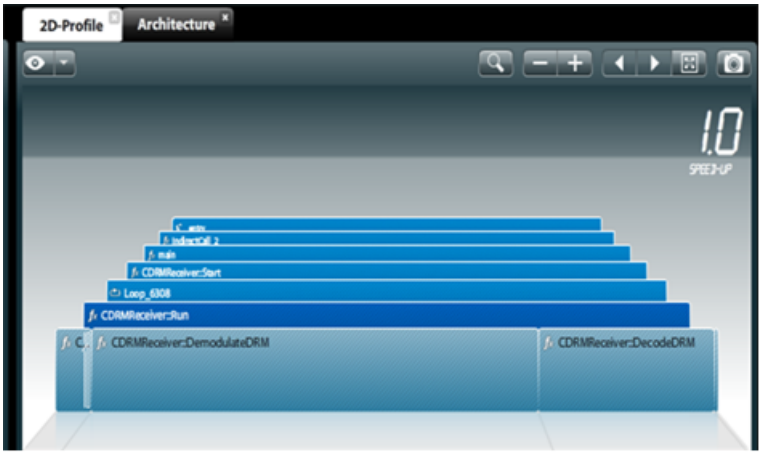


Figure 5.14: The Pareon's function distribution of the DRM program

BM variable inputs.

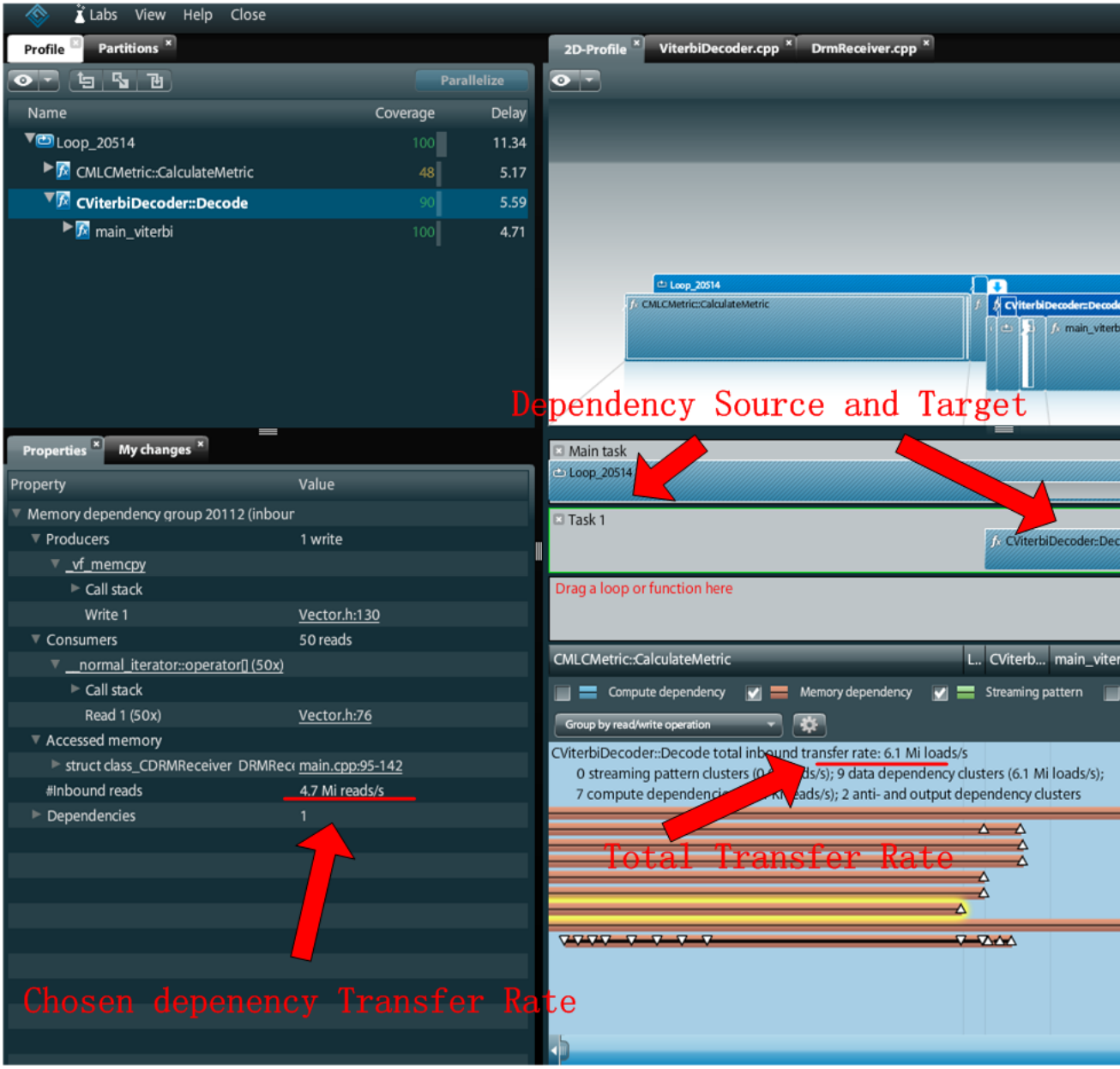


Figure 5.15: **CViterbiDecoder::Decode** inbound memory dependency 1

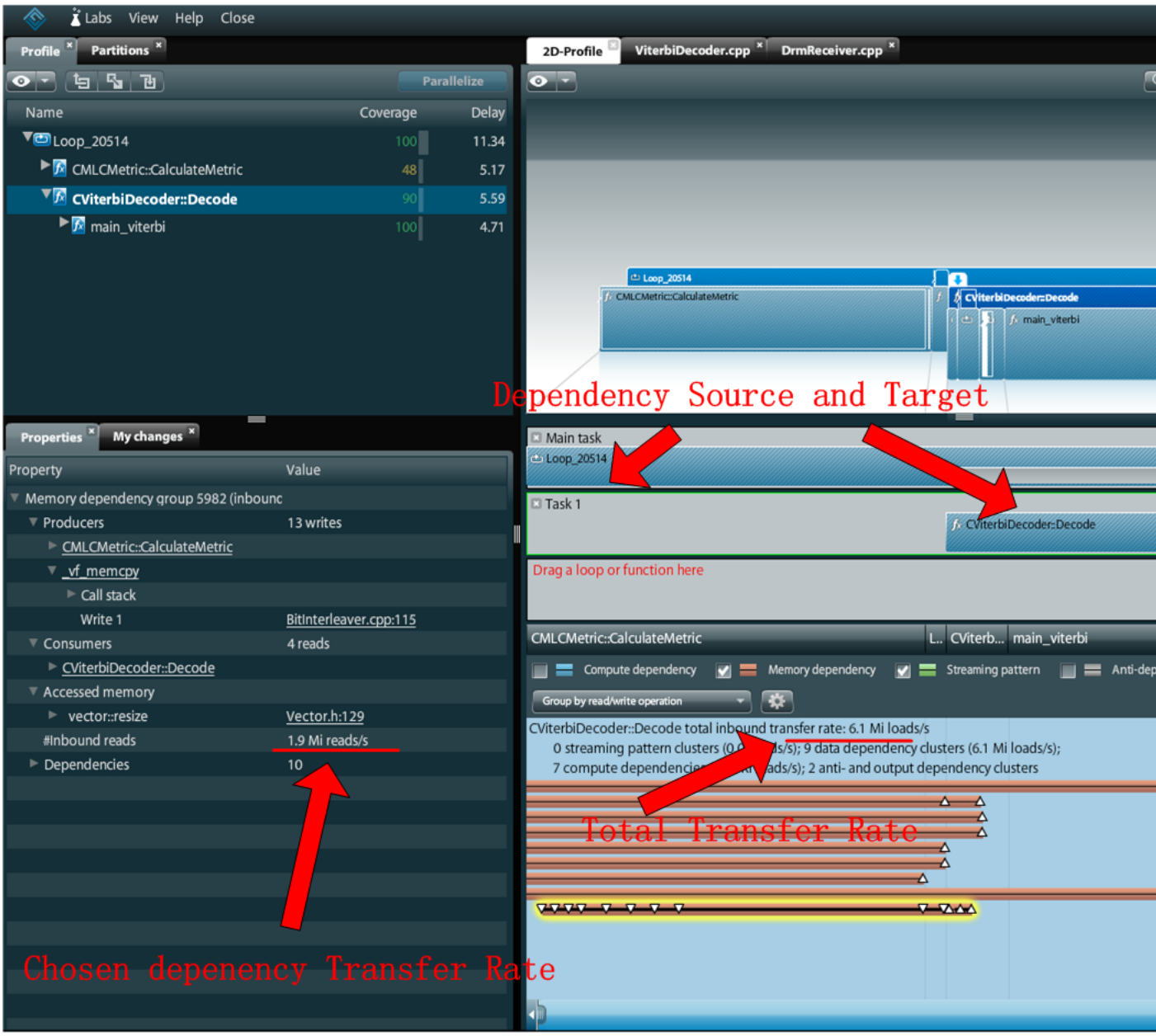


Figure 5.16: **CViterbiDecoder::Decode** inbound memory dependency 2

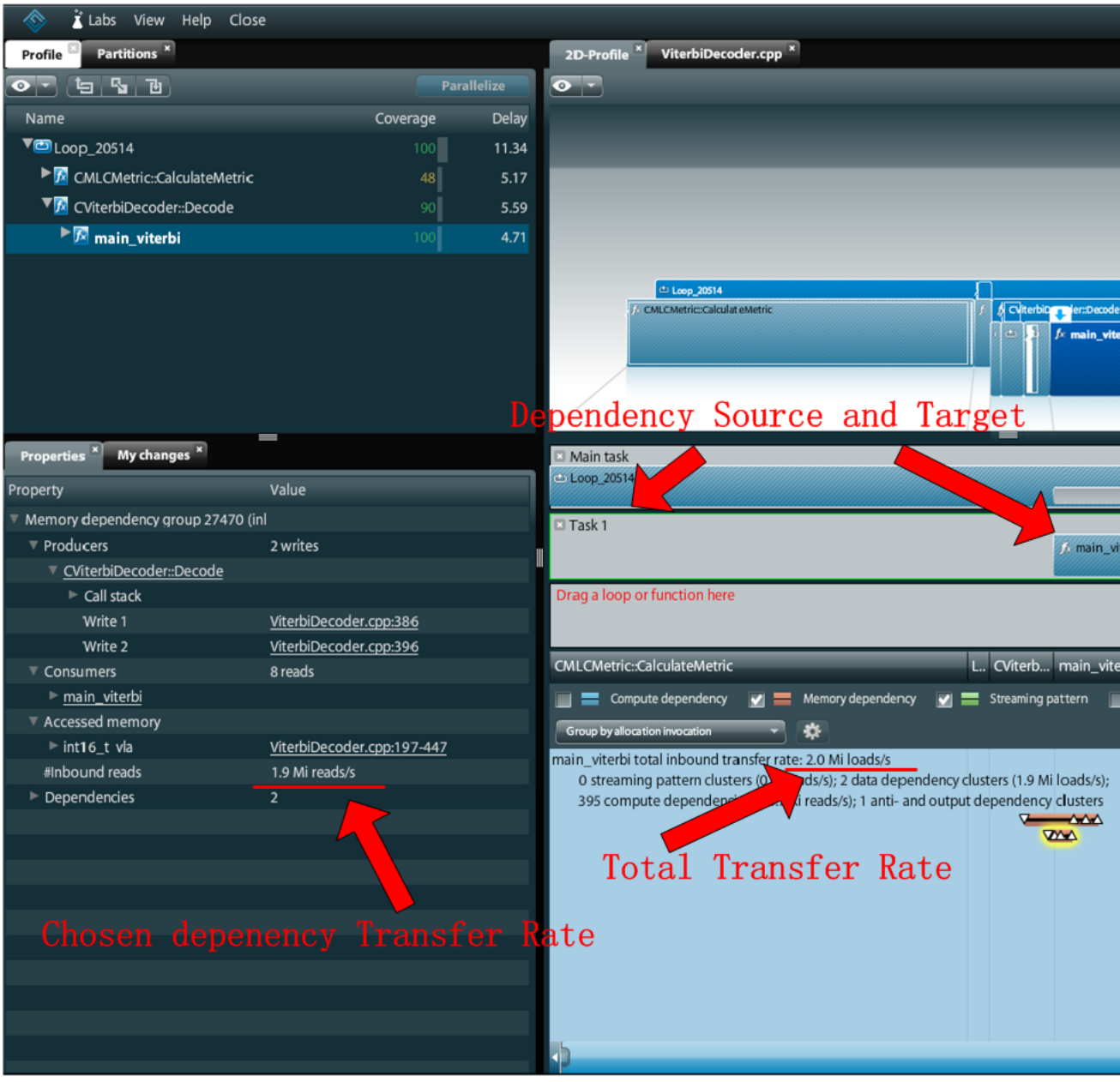


Figure 5.17: **main_viterbi** inbound memory dependency

From the total transfer rate view, the function **CViterbiDecoder::Decode** has a bigger data dependency. If the exactly communication data size needs to be analyzed, the equation 5.3 is used. *#memory access per second* is the so-called transfer rate in the Pareon. **CViterbiDecoder::Decode** has a higher transfer rate than **main_viterbi** as it is shown before. $T_{totalinvocation}$ of **CViterbiDecoder::Decode** is obviously larger than that of **main_viterbi**. Because **main_viterbi** is a subfunction of **CViterbiDecoder::Decode**. For *size of each memory access*, the variable needed to read in **CViterbiDecoder::Decode** is 32-bit float. However, in **main_viterbi** the variable size is smaller, 16-bit integer. Thus considering all the three parameters, *#memory access per second*, $T_{totalinvocation}$, and *size of each memory access* in the equation, it can be concluded that the size of data needed to be transferred in **CViterbiDecoder::Decode** is bigger than **main_viterbi**.

$$transfer\ data\ size \quad = \#memory\ access\ per\ second \times T_{totalinvocation} \times size\ of\ each\ memory\ access \tag{5.3}$$

To confirm the result obtaining from Pareon, the size of the data needed to be transferred is calculated. For n step Viterbi decoder, the data size transferred from the ARM to EVP in two version is presented in Table 5.3. For **main_viterbi**, n step needs $8 \times n$ BM inputs. 4 BM variables are packed into a 16-bit int which is 2 byte. Thus $n \times 4$ *byte* needs to be transfered. In **CViterbiDecoder::Decode** version, the data needed to be transferred, include soft-decision input **vecNewDistance** and puncturing table. The vector related dependency is not considered here, because in the practical implementation, the vector structure and function can also be located in the EVP. Puncturing table size calculation is simple, $n \times 4$ *byte*. However, **vecNewDistance** is much more complex because of the various puncturing patterns. In different puncturing patterns, there can be 2 soft-decision inputs, 4 soft-decision inputs, 6 soft-decision inputs or 8 soft-decision inputs involved in calculating the BMs of one step trellis construction. Each soft-decision input is in the variable of 32-bit float and a, b, c, d stands for number of steps punctured in certain puncturing pattern. Thus the equation shown in 5.3 can be obtained. $a \times 8 + b \times 16 + c \times 24 + d \times 32 + n \times 4$ is obviously larger than $n \times 4$, therefore transferring in data size of **main_viterbi** is smaller than that of **CViterbiDecoder::Decode**, which is consist with result obtained from Pareon.

Thus after analyzing the results from the Pareon tool, it is apparent that porting the De-puncturing function into EVP is not wise.

Table 5.3: Transferring in data size of two version

main_viterbi	**CViterbiDecoder::Deocde**
$n \times 4$ *byte*	$a \times 8 + b \times 16 + c \times 24 + d \times 32 + n \times 4$ *byte*$(a + b + c + d = n)$

For other functions in MLC, they only take a smaller part of execution time and these

functions are either hard to be vectorized or are implemented using floating-point variable. It is not wise to port them too.

In conclusion, it is not chosen to port the whole channel decoding process of MSC on EVP because of its low efficiency in EVP-C realization and its possible bad time performance on EVP which eliminates the speed's improvement benefiting from pipelined processing. The De-puncturing function is also not ported on EVP, since the Pareon shows that this multi-core version leads to a larger amount of data needed to be transferred from ARM to EVP. Finally it is determined that the Viterbi decoder function is ported on EVP.

A coarse calculation of this multi-core version's performance is done here. The processing time of each frame can be obtained from Figure 4.7. The longest processing time of generating a single service frame is 634 ms (3rd frame and 9th frame), except the first frame. In the multi-core version, if the 3rd and 9rd frame processing time can be reduced to less than 400 ms, thus all the frame processing time will be less than 400 ms and the receiver can deliver a real-time service. 45% of the 634 ms processing time, is contributed by the Viterbi decoder function, which is 289.35 ms. In the multi-core version, the Viterbi decoder function execution time is decreased. If the processing time is expected to be less than 400 ms, the Viterbi decoding process on EVP should be less than 46.35 ms which is presented in Figure 5.18. Therefore, if the communication overhead is not taken into consideration, the EVP must deliver a Viterbi decoding processing at least 6.24X ($6.24 = \frac{289.35}{46.35}$) faster than the ARM version in order to achieve a real-time service. The communication overhead is discussed in section 6.1.

In the next section, the detailed Viterbi decoder function's realization on EVP is discussed.

5.4 Achieving the Viterbi decoder on EVP

On account of the result from section 5.3, the Viterbi decoder of the DRM receiver is ported on EVP.

In the original Viterbi decoder, the frame based trace back is used. It is reasonable since the target platform of the Dream DRM receiver is on PC which has a fast processing speed and a large amount of memory available. Therefore, the PC version ignores the demands for speeding up or reduction of the memory, but mainly focuses on increasing the accuracy. The original Viterbi decoder also ignores overflow, since the PC can use 64-bit double or 32-bit float which provides a bigger range. Considering porting on EVP, these two mentioned settings have to be discussed whether they need to be changed when the decoder is ported to the EVP. The EVP processor runs at 419 MHz and the parallel processing ability (SIMD) already gives an extremely increase in processing speed, it is not necessary to change the trace back method to speed up. For memory size, a calculation is needed. For the max step decoder, 4206 trellis steps are achieved and

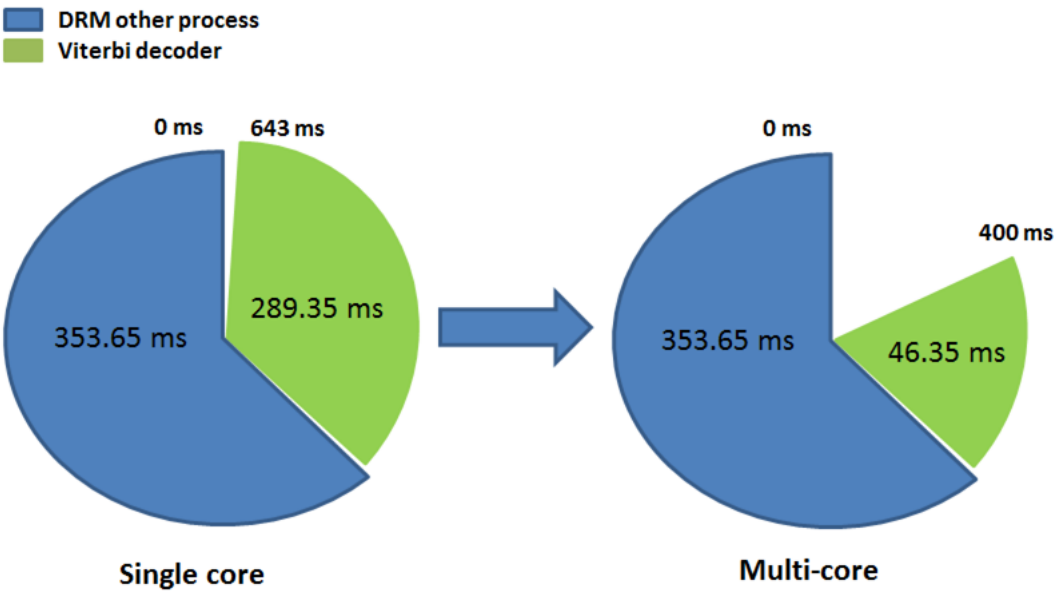


Figure 5.18: A coarse calculation of this multi-core version's performance to reach the real-time requirement

4206*64 decision bits are generated. The generating decision bits takes about 33 KB memory while the EVP data memory is 512 KB. The only 6% usage of the memory is acceptable. All these considered, the frame based trace back method is kept.

Focusing on overflow, an observation of BM and trellis steps in 3 benchmarks has been done on the original DRM program and the result can be seen in Table 5.4. From Table 5.4, an estimation of the pessimistic maximum value of PM can be achieved, benchmark1: $342.8 \times 3501 = 1200142.8$, benchmark2: $451.4 \times 4206 = 1898588.4$, benchmark3: $610.5 \times 2456 = 1499388$. In EVP, the common use variable is 16-bit int. The width can be widened to 40 bits and floating point is also supported. However, the 40 bits floating point variable is not easy to be stored in vector and it is slower than the integer on EVP. Therefore, the 16 bits unsigned integer variable is chosen to store the PM and the range is from 0 to 65535. It is obvious to see that the overflow problem may occur.

Table 5.4: The observation results of BM and trellis steps in 3 benchmarks

benchmark	**BM range**	**trellis step range**
mytest1	0-342.8	78-3501
mytest2	0-451.4	78-4206
mytest3	0-610.5	78-2456

The scaling of BM is chosen to prevent overflow rather than normalization. If applying normalization to the Viterbi decoder on EVP, a frequently judgment of overflow has to

be used and it is not easy to realize the judgment by EVP-C without decreasing the execution efficiency. Based on the register range 65535 and max trellis steps 4206, it can be calculated that the max BM value is $65535 \div 4206 = 15.58$. Considering the most pessimistic situation, the puncturing pattern is dummy, therefore BM equals the sum of 4 soft-decision inputs. The peak value of the soft-decision input should be less than $15.58 \div 4 = 3.89$. Combing the error ratio testing in Table 5.2, the soft-decision input should be scaled to 3.

The Viterbi decoder on EVP is realized based on the Viterbi decoder source code provided by Ericsson. The given code achieves a Viterbi decoding function with a decoding rate of 1/2 , constraint length of 7 and without puncturing process. In order to deliver a Viterbi decoder for the DRM according to the discussion before, these factors have to be discussed.

In the provided decoder, the input of the function is soft-decision. Since de-puncturing function is not used here, soft-decision inputs are easily calculated to generate BM. However, the needed Viterbi decoder's input should be BM. Therefore, the BM generating part is removed. An additional process has been done to store the BM in vector by EVP broadcast and shuffle operations.

The provided code is designed and implemented using the sliding window trace back. It has to be modified to frame-based trace back.

Another important change is that the given decoder can only process a frame with a constant bit size which is an integer multiple of four. The expected Viterbi decoder should process the frame with any size.

Based on those factors, the Viterbi decoder is adapted. The decoder suitable for the DRM receiver is achieved and the detailed vectorized work is discussed below.

In the trellis construction, the computation is built based on a basic butterfly which is shown in Figure 5.19. The length 7 decoder has 64 states grouping into 32 butterflies. In order to achieve the vectorization conversion of the construction process, 8 basic butterflies are combined to form a super butterfly shown in Figure 5.20. Thus one trellis construction step consists of 4 super butterfly. In a super butterfly, there are 16 PMs in the type of 16 bit integer and they are just enough to be stored in a 256 bit vector. Considering BM in a basic butterfly, there are 4 branches but every two branch values are the same. Therefore a super butterfly contains 32 BMs in 16 bit integer, however it only needs to store 16 BMs in the 256 bit vector.

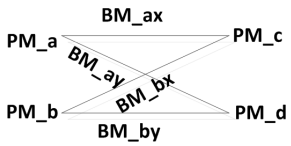


Figure 5.19: The basic butterfly in the trellis construction

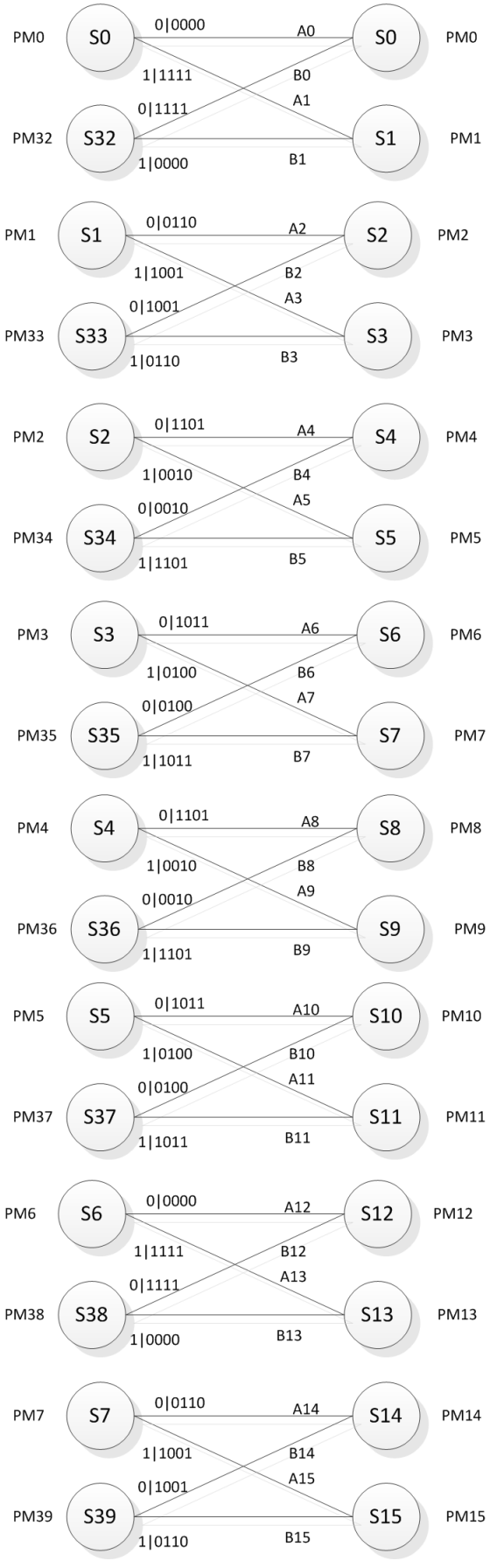


Figure 5.20: The super butterfly in the trellis construction

Thus BMs and PMs are stored in vector and thevectors can be programmed to perform the super butterfly computation action. The detailed realization of one time super butterfly computation action is presented in Figure 5.21. All the possible BMs are firstly calculated. The BM vector of the latter computation is generated based on the former BM vector by the EVP vector shuffle operation. Then two possible PM vectors are compared to decide PMs and the decision bits are also produced at the same time. One super butterfly generates 16 decision bits occupying 16 bits. Therefore 4 steps of construction produces 256 bit decision bits and they are stored in a 256 bit vector. However, this also results in a problem when the step number is not a multiple of four. Then a separated processing has to be done to the last few steps of construction based on its remainder by dividing 4.

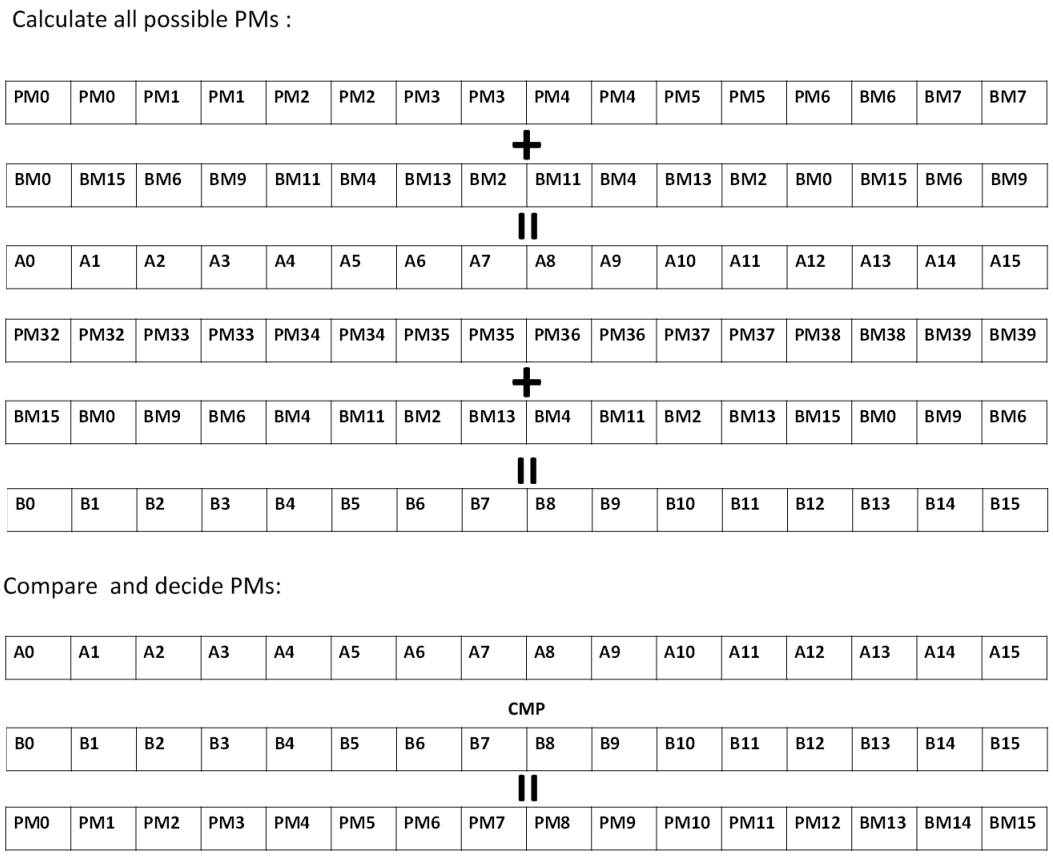


Figure 5.21: Vector computation of the super butterfly

In the trace back part, the start point is chosen as the state 000000 which is the decision bit for $PM0$ and the backward path is built. In the detailed process, the decision bits are stored in vector, however the build path routine is not vectorization. A 32 bit integer type point is applied here to load the decision bits in vector. Then according to the state in the path, the output bits can be decoded and exported.

The achieved EVP Viterbi decoder can deliver a high timing performance. The detailed timing results can be seen in Table 5.5. In the table, it shows the timing performance of the n step Viterbi decoder function including the integer version Viterbi decoder running on the Cortex R4 model, rhe EVP version Viterbi decoder's cycle number obtained from the Vdebug simulator, the estimated execution time based on the cycle number and the working frequency 416 Mhz and the execution time of the EVP Viterbi decoder on the EVP model of the System-C platform model.

It can be seen from the table that the estimated execution time is slightly longer than the actual execution time on the EVP System-C model. In the estimation, the cycle number is obtained from the Vdebug + Vsim simulator. The Vdebug simulates a full memory model including the memory access penalty. The cache is also simulated in the Vdebug. The cache coherence and cache miss problems are also considered. However, in the System-C model, the cache is not modeled and the memory access penalty is also ignored. Thus the overhead in fetching data from the memory is smaller in the system-C model. This is the reason for the shorter execution time in the System-C model than the estimated result.

From the results, it can be summarized that the execution cycle is approximate $31 \times n$ *cycle* and the execution time is about $n \times 7.5 \times 10^{-5}$ ms. The EVP version is more than one hundred times faster than the ARM version, which is much higher then the 6.24x speed up requirement. Thus a time period of about 45 ms is left for the communication latency and other overheads of a frame processing.

Table 5.5: Viterbi decoder's timing performance

Decoder Step n	**Cortex R4 (ms)**	**Vdebug + Vsim Simulator (cycle)**	**estimation (ms)**	**EVP model(ms)**
78	1.657	2705	0.007	0.007
216	3.003	6971	0.017	0.017
1171	16.001	37177	0.089	0.086
2337	25.007	73839	0.177	0.170

In this chapter, the DRM program's multi-core version is analyzed with the help of the tool Pareon and finally it is determined to port the Viterbi decoder function on EVP. The detail of the realization of the Viterbi decoder on EVP is also presented. The timing performance of the Viterbi decoder on EVP is outstanding and leaves a time of about 45 ms for the communication latency. In next chapter, the development of the communication between the EVP and ARM processors is discussed.

Chapter 6

Communication between the ARM and the EVP

In this chapter, the working principle of the DMA controller and its detailed configuration is introduced in Section 6.1. Afterwards, the design of the communication between the EVP core and the ARM processor is presented in Section 6.2.

6.1 Programming on the DMA

The DMA controller on the target platform is AXI DMAC CORE V2. It is an AXI bus controller with three 64-bit master ports and one 32-bit slave port which efficiently moves data from one location to another. Since the DMA launches communications instead of the processor, the control task is offloaded from the processor and the processor can operate other tasks in parallel with DMA's communication tasks.

The architecture of the DMA implementation on the target platform is presented in Figure 6.1. The EVP and ARM processors can access the DMA parameter register through the slave port to control the configuration of the transfer including source address, destination address, transferring data size, etc. According to the setting, the 3 master ports can access the source and destination components to execute the data transfer. From Figure 6.1, it can be seen that the slave port is connected to the APB bus rather than AXI bus. As it is discussed in section 3.3, the AXI bus delivers a burst-based, pipelined data transfer. However, the written transfer to the DMA register is simple data transfer rather than the burst transfer because scalar variables needs to be transferred to various registers in scattering addresses in the DMA configuration. Thus APB is a suitable bus which reduces interface complexity and lowers power consumption and provides a non-pipelined data transfer. A bridge is located between the APB and AXI to serve as a frequency and transfer protocol convertor. For the purpose of energy saving, the APB bus works at a much lower frequency than AXI. Since the DMA configuration registers

are in the data width of 32 bits, a data width conversion from 64 bits to 32 bits is necessary.

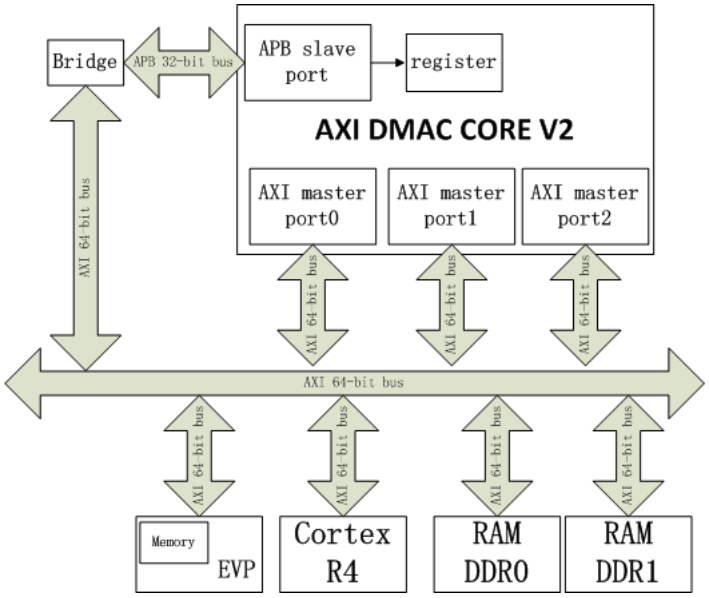


Figure 6.1: The architecture of DMA implementation on the target platform

The DMA controller can handle 4 kinds of transfer classified by source and destination type, namely memory-to-memory, peripheral-to-memory, memory-to-peripheral and peripheral-to-peripheral. In this DRM implementation, the DMA only involves in the memory-to-memory transfer to achieve the communication between the EVP and ARM processor. The memory of the EVP is embedded in the EVP processor and memory of the Cortex R4 processor is external DDR RAM0 and RAM1. The DMA launches the data transfer between the EVP memory and the DDR RAM.

The data flow of DMA memory-to-memory data transfer can be seen in Figure 6.2. The master read port0 reads data from the memory through AXI bus. The data is stored in the FIFO buffer of DMA. When the data stored in the FIFO is enough to start a transfer task launched by the master write port, the data in the buffer is written to the destination memory through the AXI bus. The byte size of a single transfer task is decided by the value in PLEN (Packet Length) register which indicates the maximum number of bytes that will be transferred per task. When the transfer task ends, the DMA processes an arbiter operation to choose the next processing channel. If the same channel is chosen to continue processing, a similar transfer process is repeated. However, the arbitration process between transfer tasks will add an additional overhead to the transfer. Thus the data in the size of Packet Length is transferred from one memory to another. This process is called a transfer task and multi transfer tasks compose a DMA transferring job. The max data size of a DMA transferring job is 0.5 MB.

In the practical use of the DMA controller, the ARM Cortex R4 processor is programmed

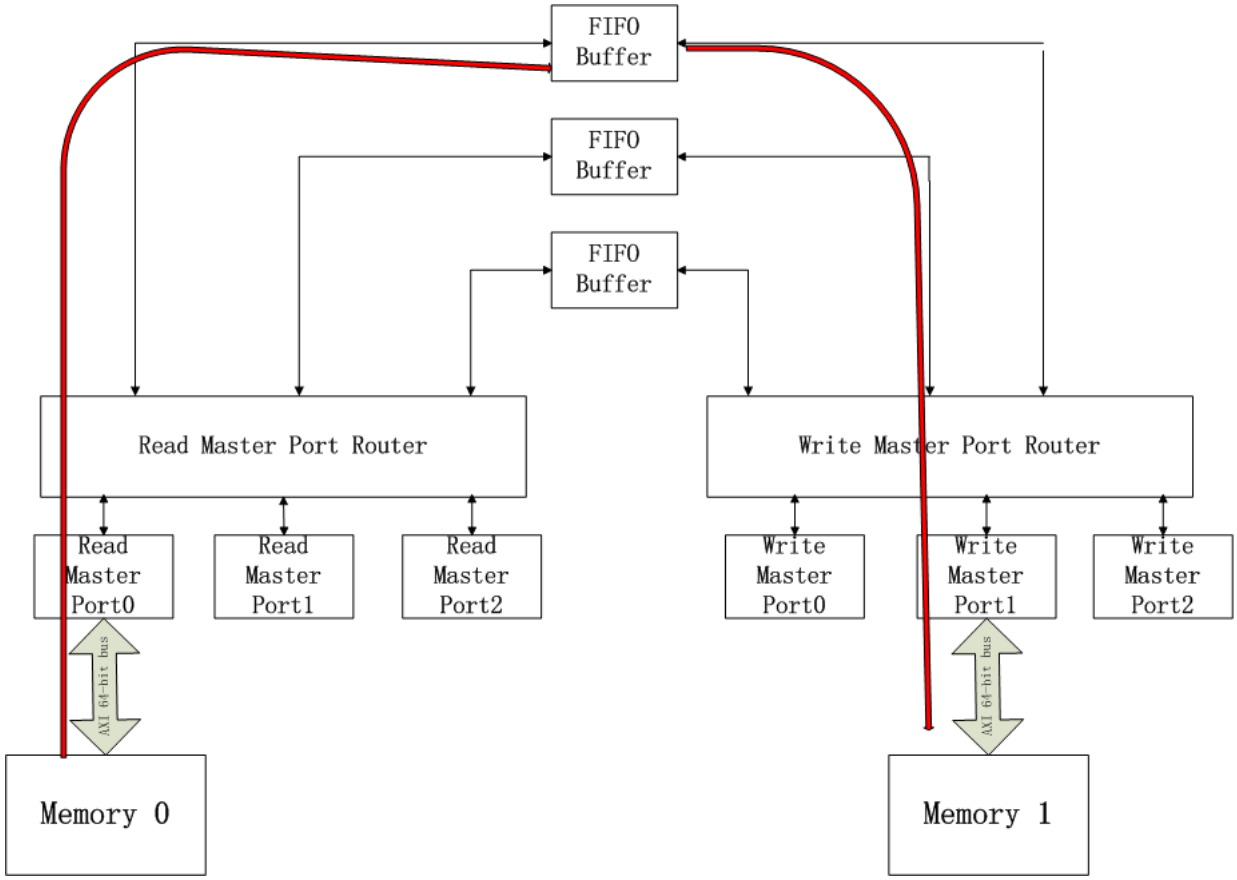


Figure 6.2: memory to memory DMA data transfer

to write to the registers of the DMA controller to configure the transfer. At the start of the program, the DMA transfer is initialized. The detailed register configuration is shown in Table 6.1. The Packet length is set to the max value 128 bytes in order to decrease the arbitration overhead. Since the bandwidth is 64 bits along the whole bus of the system, the Source Bus Size and Destination Bus Size are assigned as 64 bits. Furthermore, the source address and destination address is configured to auto increment with the transfer.

Once the transfer is initialized, the Cortex R4 can set specific transfer parameters to the DMA controller to launch a transfer. The specific parameter is set as in Table 6.2. The transfer data size, source address as well as destination address are set according to the transfer's detail. Once all the parameters have been sent to DMA registers, the channel enable filed is assigned to 1 to launch the transfer. With the data transfer, the value of the register Transfer Count Value decreases until zero which indicates that the data transfer is ended. Thus one time data transfer has been finished.

In the target platform system-C model, it uses Transaction Level Modeling 2.0 (TLM

Table 6.1: The detailed register configuration of DMA transfer initialization

Field name	**Setting**
Request Mode	Memory to Memory
Master Mode	master 0 read, master 1 write
Packet Length	128 byte
Source Bus Size	64 bit
Destination Bus Size	64 bit
Source Address Increment	Increment by Source Bus Size
Destination Address Increment	Increment by Destination Bus Size

Table 6.2: The detailed register configuration of a specific transfer

Field name	**Setting**
Transfer Count Value	communication data size in byte
Transfer Count Enable	count down
Source Address	source address in the global memory map
Destination Address	destination address in the global memory map
Channel Enable for channel 1	enable

2.0) to model the communication. However in order to reduce the model's complexity, the model does not simulate the burst transfer time of the communication controlled by DMA. In the DMA model-C code, the function **MasterReadFormatTLM2** is called to process the master port's reading from the source component. In the function, the blocking transport call's delay is set zero. It is the same case in the function **MasterWriteFormatTLM2** which process the master port's writing to the destination component. Thus the latency in the burst transfer is zero in the model. An additional transfer latency should be taken into consideration in the profile report in the multi-core DRM software on the target model.

6.2 Communication between the ARM and EVP with the DMA

In the last section, the data transfer controlled by the DMA controller has been achieved. Based on the DMA data transfer, the communication protocol between the EVP and ARM cores implementing in the DRM receiver multi-core version, is presented in this section.

Since the EVP core is idle during most of the execution time, it is set in the low power mode when it is not waked up by the external interrupt sent by the ARM core. Once

the EVP task is finished, the EVP will interrupt itself to turn to the low power sleep mode.

The detailed EVP code of the state switching and response to the interrupt is based on the EVP example implementation **Sound Machine**. Three interrupt requests, namely **WakeUp, Sleep and Exit** and one semaphore event, **CONTINUE** are defined to realize the state switching and they are presented in Figure 6.3. In the run state, it processes the decoding task and switches into the sleep mode if it receives a **Sleep IRQ**. In the sleep mode, it uses the EVP function **evp_wait()** to wait for a semaphore event. Once **WakeUp IRQ** occurs, it calls for the semaphores event, **CONTINUE** to trigger the EVP into the run state. The state, run and sleep both can switch into the exit state to completely end the EVP's execution.

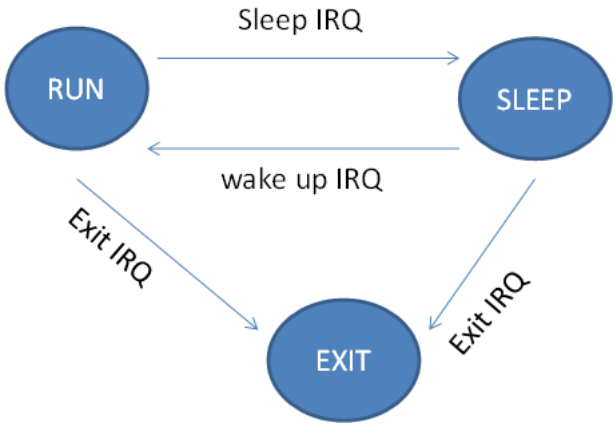


Figure 6.3: The design of EVP state switching

However, in the practical implementation of the sleep mode on the EVP model, we have to prevent an error occurring that the EVP core stalls forever when it steps into the sleep mode. When there is a load started before a wait instruction, the stall can happen that the read data is returned after the core clock has been switched off by the wait instruction. It can be solved by adding a **evp_dsb()** function before **evp_wait()** to insert a data synchronization boundary instruction that forces the memory read to be finished before the wait instruction is executed.

The components, ARM, EVP and DMA involved in the communication need to synchronize with each other to inform their own working progress. The detail of the communication design is presented in Figure 6.4.

From the sequence diagram, it can be seen that the ARM core firstly initiates a DMA transfer to send the Viterbi decoding input data to the EVP. After configuration, the DMA starts the burst transfer. The ARM core is informed of the transfer's ending by checking the transfer status register of the DMA. When the transfer ends, the ARM core sends request to the DMA to clear the transfer status register to prepare for the next

time transfer.

Once the Viterbi decoding input data is already available in the EVP memory, the ARM core can send the interrupt to the EVP to wake it up. Then the EVP starts the decoding task. When the task is finished, it changes the flag to show that and turns into the sleep mode. At the same time, the ARM core checks for the EVP's flag to obtain the progress of the EVP task . Once being informed of the finish of the decoding task , the ARM core initiates a DMA transfer to read the Viterbi decoding results into the ARM's memory in the same way as the write to the EVP memory.

There is also a pack work implemented in the data needed to be transferred to decrease the communication data size. As it is mentioned in Section 5.3, the data transferred from the ARM core to the EVP can be packed. Each variable's value is less than 12 which can be presented by a register of 4 bits, therefore, four variables can be packed in a 16-bit short int. The decoding result which need to be transferred from the EVP to the ARM is binary bit and each only occupies 1 bit. Thus 16 variables can be packed in a 2-byte int. There are unpacking routines in both the ARM and EVP sides in order to use the data.

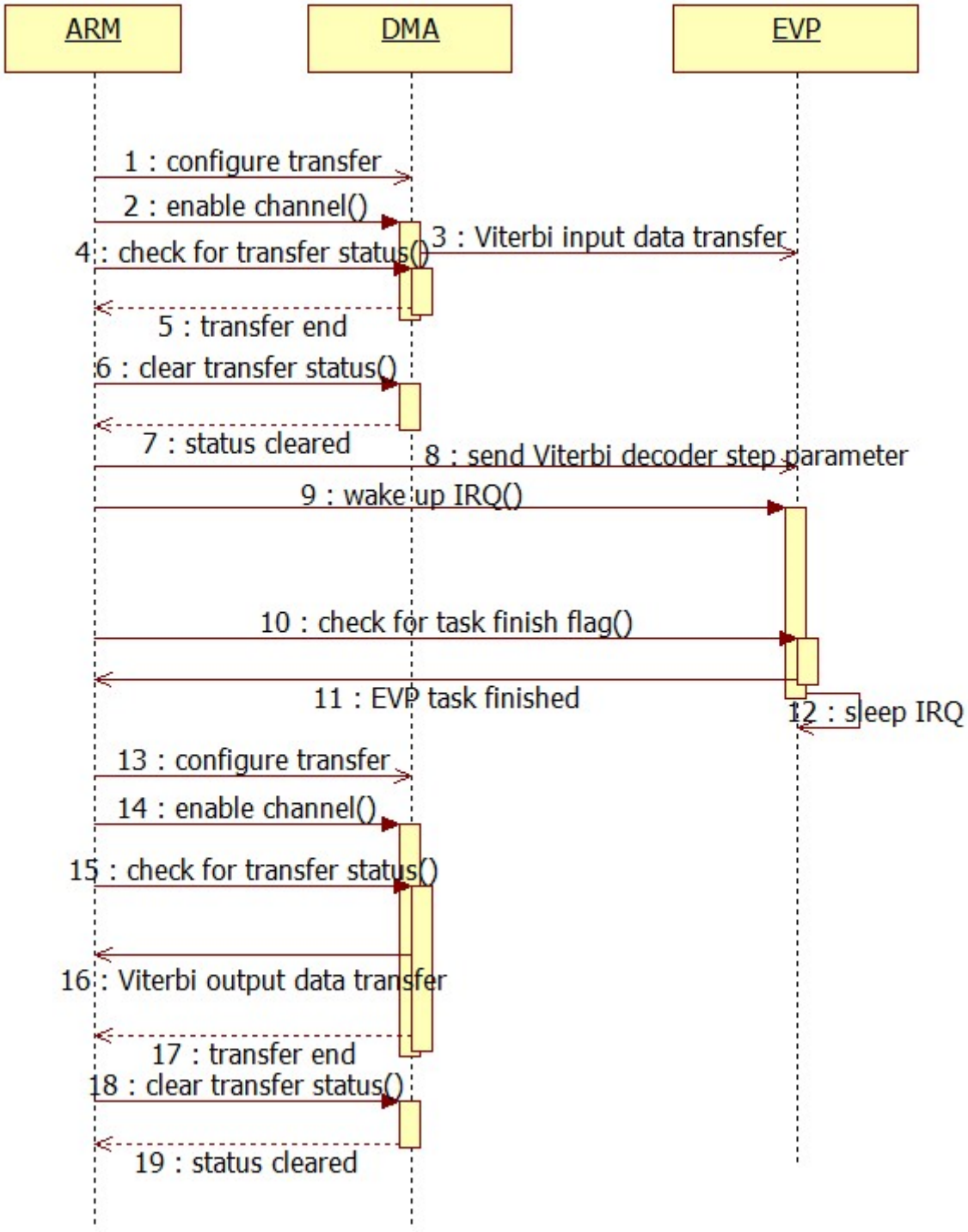


Figure 6.4: The design of the communication between the ARM and EVP

After finishing the detailed development of communication, the Multicore Communications API (MCAPI) is used to modularize the multi-core communication. The detailed MCAPI design is presented in Appendix B.

In this chapter, the development and design of the communication between the EVP an ARM processors is presented. A estimation of the multi-core performance can be done based on the communication design and the timing performance on EVP. The process of the estimation can be seen in next chapter.

Chapter 7

Multi-core version Performance's Estimation and Verification

This chapter consists of two parts. One is to estimate the multi-core timing performance in section 7.1. Another in section 7.2 is to present the profile report on the platform model and the actual performance is compared with the estimation.

7.1 Estimation of the Multi-core version Performance

Combining the design of the multi-core software and communication settings in previous chapters, a multi-core version performance can be estimated here. The processing time of each frame in benchmark1 is estimated using equation 7.1. The estimation does not involve in the first frame, since it will not influence the real-time service. T_{DRM} is the execution time of the DRM other code on ARM. T_{DMA} indicates the time spent in the DMA configuration for a specific transfer . The time resulted from synchronization is in $T_{synchronization}$. The burst transfer latency is $T_{communication}$. At last, T_{EVP} is the Viterbi decoding processing time on EVP.

$$T_{frame} = T_{DRM} + T_{DMA} + T_{synchronization} + T_{communication} + T_{EVP} \tag{7.1}$$

T_{DRM} is easy to obtain by using the Cortex R4 results in Figure 4.7 and the Viterbi decoder's 45% contribution. The processing time of each frame is multiplied by 55% to get the DRM other processing's execution time and the result is in Table 7.1

For the calculation of T_{DMA}, $T_{synchronization}$, $T_{communication}$ *and* T_{EVP}, the frame decoding workload should be known in each frame. In the bechmark1, the 3-level MLC decoder is applied for the MSC in the transmitter. And the number of source information bits in each stream are 1171, 2337 and 3501 respectively. Since the iterative parameter is 2 in

Table 7.1: T_{DRM} from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{DRM} ms	310.2	348.7	306.9	305.8	345.4	306.35	308.55	348.7	305.8	306.9

the MLC decoder of the receiver, thus there are two 1171-step, two 2237-step and two 3501-step Viterbi decoding routines in the process of decoding a MSC frame. The 2-level MLC is applied to the SDC frame and the number of source bits in each stream are 216, 426 respectively. Thus there are two 216-step, two 426-step Viterbi decoding routines in the process of decoding a SDC frame. In FAC, the 1-level MLC is used and the number of source bits in the stream is 78. Though the iterative loop is 2, it is not suitable for the 1-level MLC decoder. There is only one 78-step Viterbi decoding routine in the decoding one FAC frame. In a super frame, the first frame contains one MSC, one SDC frame and one FAC frame and the other two frame only contain one MSC and one FAC frame. For a larger decoding workload of the frame, the work consists of two 1171-step, two 2337-step and two 3501-step, two 216-step, two 426-step and one 78-step Viterbi decoding routines. For a smaller workload of the frame, there are two 1171-step, two 2337-step and two 3501-step and one 78-step Viterbi decoding works. The detailed workload in each frame can be seen in Table 7.2. The number of Viterbi decoding routines in one frame is shown in 7.1. The trends of T_{DMA}, $T_{synchronization}$, $T_{communication}$ *and* T_{EVP} are the same as that of $\#Viterbi\ routines$ in each frame.

Table 7.2: Workload from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Frame Work Load	FAC + MSC	FAC + SDC + MSC	FAC + MSC	FAC + MSC	FAC + SDC + MSC	FAC + MSC	FAC + MSC	FAC + SDC + MSC	FAC + MSC	FAC + MSC

In T_{DMA}, the ARM core sends source address, destination address, transfer size as well as enable signal to the DMA controller. These values are firstly calculated in the ARM program and then copied to the DMA address by **memcpy**. Each **memcpy** instruction takes 0.0002 ms. It is measured by the analysis function of the Synopsys tool. Four **memcpy**s are called in the configuration. Thus one DMA configuration takes about 0.0008 ms. The DMA register has to be configured twice in each Viterbi process on EVP for data transferring in and data transferring out. For 2nd frame, it has to decode one MSC, one FAC and seven Viterbi decoder functions are called. Therefore, T_{DMA} equals $0.0008 \times 2 \times 7$ ms. The rest frames can be calculated in the same way and the result can be seen in Table 7.3.

$T_{synchronization}$ indicates the time spent in the synchronization between different compo-

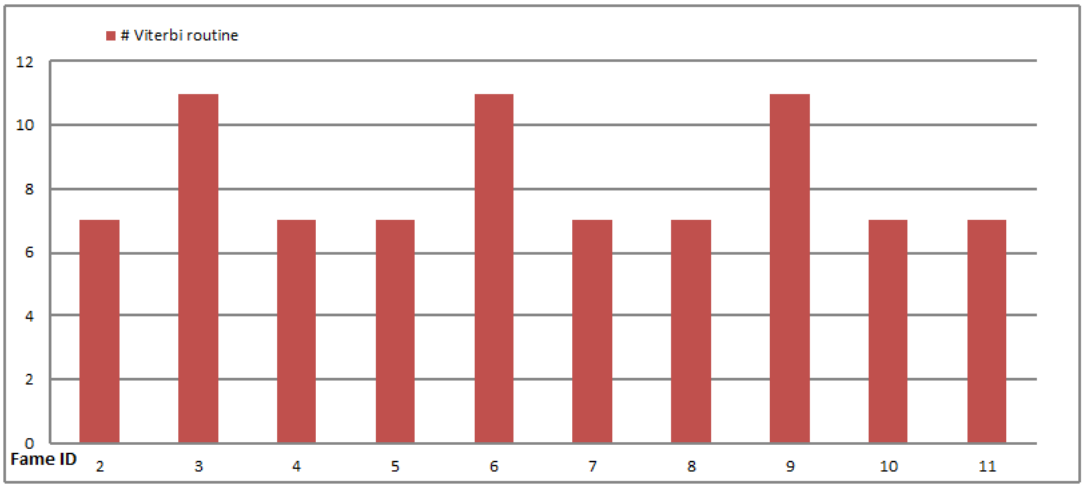


Figure 7.1: The number of Viterbi decoding routines in each frame

Table 7.3: T_{DMA} from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{DMA} ms	0.011	0.018	0.011	0.011	0.018	0.011	0.011	0.018	0.011	0.011

nents. The DMA has to inform the ARM core of the data transfer's ending or the status register's clear. The ARM processor requests the EVP to start work and the EVP has to inform the ARM core that the Viterbi processing is finished. It is hard to predict on this value, here we assume that each synchronization process takes 0.001 ms which is about 400 ARM cycles. There are 6 synchronization processes in each Viterbi decoding. Thus $T_{synchronization}$ can be calculated and the results are in Table 7.4.

Table 7.4: $T_{synchronization}$ from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{syn} ms	0.042	0.066	0.042	0.042	0.066	0.042	0.042	0.066	0.042	0.042

Based on the Viterbi decoding input and output data size and hardware details, the communication latency $T_{communication}$ is estimated as following.

Firstly, a coarse calculation excluding the burst transfer's details and the DMA control, is achieved to measure the communication time. For a n step Viterbi decoding, combining with the data package condition, the data size from the ARM core to the EVP core is $n \times 4$ $byte$ and from the EVP core to the ARM core is $\lceil \frac{n}{16} \rceil \times 2$ $byte$. For instance, the processing of the 2nd frame, contains two 1171-step, two 2337-step and two 3501-step and one 78-step Viterbi decoding and each Viterbi processing's transferring in and

transferring out data can be calculated, then the results are accumulated. Thus the total data transfer from the ARM core to the EVP core and from the EVP core to the ARM core is calculated in Table 7.5.

Table 7.5: Data transfer calculation in each frame in beanchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
ARM to EVP byte	56384	61520	56384	56384	61520	56384	56384	61520	56384	56384
EVP to ARM byte	1762	1926	1762	1762	1926	1762	1762	1926	1762	1762
Total byte	58146	63446	58146	58146	63446	58146	58146	63446	58146	58146

The bus's theoretic highest throughput is calculated by the formula 7.2. For the AXI bus, $width_{bus}$ is $64bit$ and $frequency_{bus}$ is $104Mhz$, thus $throughput_{bus}$ is $793Mb/s$. The transferring data first is read by the DMA master port and then is written to the target memory. These two time periods have to be added in calculation the latency.

Combing the transferring data size and the AXI bus throughput, a rough $T_{communication}$ in frame can be calculated in Table 7.6.

Table 7.6: The theoretic $T_{communication}$ estimation from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{com} ms	0.140	0.153	0.140	0.140	0.153	0.140	0.140	0.153	0.140	0.140

Secondly, a more accurate communication latency is discussed. The DMA controller's working frequency f_{DMA} is $104Mhz$. In the DMA controlling burst transfer as shown in Figure 7.2, one communication consists of multi tasks. The DMA controller needs an arbitration operation to choose the next process communication task. According to the DMAC V2 user reference, the arbitration cycle c_{ta} is about 20 cycle. The maximum number of bytes transferred for each task is programmed in the PLEN register (up to a max of 128 bytes). As mentioned in 6.1, the PLEN of the DMA is configured as 128 bytes to decrease the arbitration overhead. One task has serval bursts. When one burst transfer is finished, a burst synchronization c_{bs} is needed. There is no detailed value of this synchronization cycle in the reference, We assume c_{bs} is 1 cycle. Each burst contains some beats. According to the AXI bus protocol setting, each burst can contain 1 beat, 4 beats, 8 beats and 16 beats and transferring one beat needs 1 cycle. Here it is assumed that 16-beat burst is chosen. The number of byte per beat depends on the bus bandwidth. Therefore, the beat size is 8 byte on the 64-bit AXI bus. One burst send

data of 128 byte, so a communication task consists of 1 burst. Furthermore, a initial latency c_{il} has to be added before launch one communication job and it is 25 cycles according to the user reference.

$$throughput_{bus} = width_{bus} \times frequency_{bus} \tag{7.2}$$

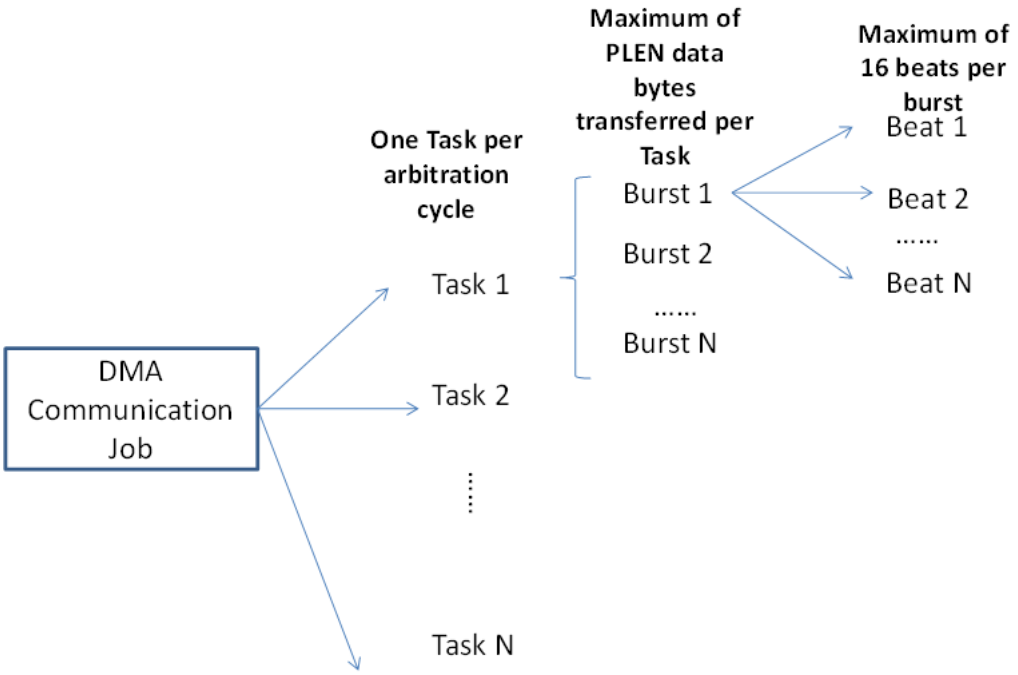


Figure 7.2: The structure of the DMA burst communication job

Based on the DMAC V2 controlling burst transfer detail and the assumption, it can estimate the communication latency in equation 7.3. The communication latency consists of 3 parts, DMA control latency, read master port's burst transfer latency as well as write master port's burst transfer latency. DMA control latency includes the initialization latency c_{il}, arbitration cycle between two tasks and synchronization cycle between two burst. According to the PLEN and burst setting, one task only has one burst. Since n is the transferring data size in byte and burst number is $\lceil \frac{n}{128} \rceil$, the total arbitration cycle is $(\lceil \frac{n}{128} \rceil - 1) \times c_{ta}$ and total burst synchronization cycle is $\lceil \frac{n}{128} \rceil \times c_{bs}$. The master read latency equals to the master write latency. It can be calculated by burst number multiplied by the number of cycles for each burst transfer. Since it is a 16-beat burst, c_{burst} is 16 cycle. α is used to present the utilization of the bus. Only when multiple DMA channels are being serviced simultaneously, the DMA can achieve the bus's theoretical max throughput. However, since only one DMA channel is used, it cannot achieve the theoretical max throughput. Based on the experimental data in the user reference, the one DMA channel can reach 50% usage. Thus α is set as 50%.

$$\begin{aligned} T_{latency} &= T_{DMA\_overhead} + T_{master\_reading} + T_{master\_writing} \\ &= [c_{il} + (\lceil \frac{n}{128} \rceil - 1) \times c_{ta} + \lceil \frac{n}{128} \rceil \times c_{bs}] \times T_{DMA} + \frac{\lceil \frac{n}{128} \rceil \times c_{burst} \times T_{bus}}{\alpha} + \frac{\lceil \frac{n}{128} \rceil \times c_{burst} \times T_{bus}}{\alpha} \end{aligned} \tag{7.3}$$

In summary, the equation 7.3 can be used to give a more accurate estimation of the communication latency. However, the estimation ignores the cache's influence in the communication. In the actual hardware's communication, data to be transmitted is firstly read from the memory to the cache. Then the cache data is transferred into the AXI bus. The cache controller reaches the optimal performance when it receives AXI transactions that target full cache lines. Since the access latency of the cache is much less than the latency of the memory, it decreases the fetch overhead. However an additional overhead occurs for reading the data from memory to cache and it also raises a cache coherence referring to the consistency problem of data stored in local caches of a shared resource. In the system-C model, the access penalty for all type of memorys is zero. Therefore, the cache is useless in this way. The model ignores the cache to decrease the model complexity to increase the simulation speed. The AXI bus reads data directly from the memory.

Using equation 7.3, the latency of each transfer can be calculated and $T_{communication}$ is achieved by adding every transfer time in each frame and the result can be seen in Table 7.7. Comparing with the coarse estimation, the $T_{communication}$ value is two times larger. The accurate $T_{communication}$ is used in the following calculation.

Table 7.7: The accurate $T_{communication}$ estimation from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{com} ms	0.380	0.418	0.380	0.380	0.418	0.380	0.380	0.418	0.380	0.380

T_{EVP} is achieved using the equation $n \times 7.5 \times 10^{-5}$ ms from section 5.4 and the result can be seen in Table 7.8.

Table 7.8: T_{EVP} estimation from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{com} ms	1.057	1.154	1.057	1.057	1.154	1.057	1.057	1.154	1.057	1.057

Finally, T_{DRM}, T_{DMA}, $T_{synchronization}$, $T_{communication}$, T_{EVP} are accumulated to get T_{frame} and the result can be seen in Table 7.9. From the estimation, all the frame's processing time is less than 400 ms. Therefore, it is expected that the multi-core version of DRM receiver can deliver a real-time service before the actual porting work.

Table 7.9: T_{frame} from frame2 to frame 11 in benchmark1

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
T_{DRM} ms	310.2	348.7	306.9	305.8	345.4	306.35	308.55	348.7	305.8	306.9
T_{DMA} ms	0.011	0.018	0.011	0.011	0.018	0.011	0.011	0.018	0.011	0.011
T_{syn} ms	0.042	0.066	0.042	0.042	0.066	0.042	0.042	0.066	0.042	0.042
T_{com} ms	0.380	0.418	0.380	0.380	0.418	0.380	0.380	0.418	0.380	0.380
T_{EVP} ms	1.057	1.154	1.057	1.057	1.154	1.057	1.057	1.154	1.057	1.057
T_{frame} ms	311.690	350.356	308.390	307.290	347.056	307.840	310.040	350.356	307.290	308.390

7.2 Profile results on the M7400 platform

For the actual porting of the multi-core DRM program on the M7400 platform System-C model, A functional check is done firstly. Since the multi-core version demodulate and decode the signal in exactly the same way and in the same precision as the single-core version, the error rate of three benchmarks is completely the same as Figure 4.6 in Section 4.3.2, which shows the error rate of the modified DRM program running on the Cortex R4 model.

Before discussing the timing performance, it has to mention the platform model's setting. Above all, the burst transfer latency on the AXI is set as zero and the cache is also not modeled. The ARM's code, data are stored in the RAM memory, but the access time to the RAM memory and the cache are all zero. Therefore, if the DRM receiver is ported on the real hardware, the transfer latency, the access penalty to the RAM and the time spent in loading data from RAM to cache will be added to the execution time.

A similar timing performance measurement is done as before, using the benchmark 1. The time of generating each MSC frame is presented in Figure 7.3. It can be seen that all the frame's processing time are below 400 ms, therefore, it provides a real-time service. As it is known that the data transfer time is zero, even though the transfer latency is taken into consideration using $T_{communication}$, all the processing time are still below 400 ms.

Profile reports based on benchmark 1 are achieved as before to research on the function distribution on generating 1st MSC frame and generating 2nd frame to 11th frame respectively and the results are presented in Figure 7.4 and Figure 7.5. The function contribution in the first frame is similar with the Cortex R4 single core version and the Time synchronization routine takes more than 70% of the execution time. However, in the 2nd frame to 11th frame processing, it is rather different from the Cortex R4 single core version. With the help of the EVP, the Viterbi decoding time is significantly decreased, thus its belonging processing stage, MLC's portion is reduced from 58% to 25%. And the Viterbi decoder processing including the EVP processing time and DMA configuration time, synchronization overhead and other latency, only takes 3.5% of the

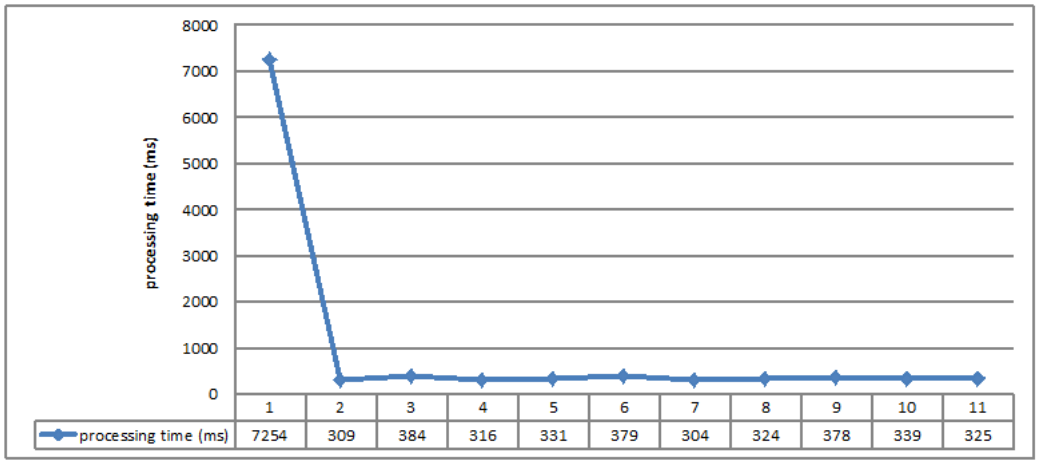


Figure 7.3: MSC frame's processing time on multi-core

whole time, but the burst transfer latency is ignored in the model. Now the most time expensive part is the Channel Estimation stage, and it takes 44% of the execution time.

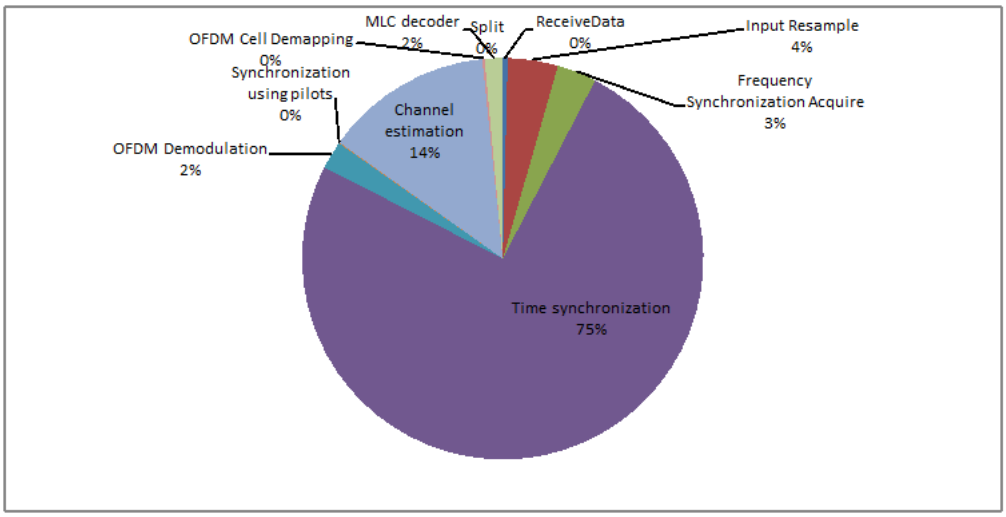


Figure 7.4: Profile report of processing the first frame on multi-core

Comparing the actual timing performance with the estimation shown in Table 7.9 of last section, it can be observed that the estimation gives an optimistic prediction in execution time though the model already ignores the communication time. It is because that it delivers a much bigger T_{syn} in the model than the estimation. It is observed in the running of the model that the ARM processor sends the interrupt to the EVP to wake it up from the sleep mode and this process takes about 1.5 ms which is extremely

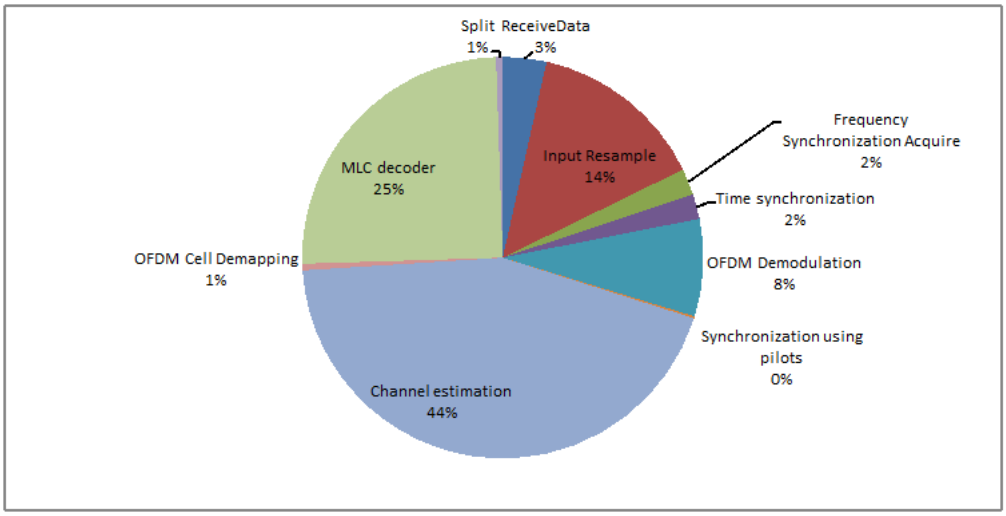


Figure 7.5: Profile report of processing the 2nd to 11th frame on multi-core

high. And it happens in every EVP Viterbi processing and the time is various. Thus in a frame, for instance in Frame4, about $1.5 \times 7 = 10.5$ ms synchronization spent in wake-up, will occur. Normally, the EVP's response to an interrupt will not takes such a long time. It is the System-C model's simulation process that leads to this.

We run the System-C model in the speed optimized mode. In the speed optimized mode, the two core models' execution timing synchronization results in this high response and this synchronization problem is called quantum effect. The ARM core and the EVP core are modeled separately and each core emulates its execution time respectively. Using the Synopsys Virtual Prototype Analyzer, the execution time in each core is observed. When the ARM core sends the interrupt, it is in the time of $1,432,948,100$ ps and the EVP core which is in the sleep mode, is in the time of $2,999,114,365$ ps. Then the model starts the two core's synchronization, the EVP core's time is kept and it does not execute instructions until the ARM core's time reaches $2,999,114,365$ ps. Then the EVP core wakes up and starts the Viterbi decoding processing. Thus an additional 1.567 ms response overhead appears. This model synchronization problem only appears in the model and the program does not need to undergo this additional overhead in the real hardware.

However, in the full optimized mode of the System-C model, this model's time synchronization problem does not appear. The two core's execution time is always synchronized. This full simulation mode also results in an about 500 times slower simulation speed comparing with the optimized mode. In one second simulation time (the real word time on watch), the speed optimized mode simulates 41 ms time of one EVP and one ARM core's running. However, the full simulation mode just simulates 0.074 ms running. This mode is in a extremely slow simulation speed. So the full simulation mode is not suitable for simulating big and complex programs like DRM, though it does not have

the synchronization problem.

Another reason for the inaccurate estimation, is the error in the estimation of T_{DRM}. It is calculated by using the function distribution in the whole process from 2nd frame to 11th frame and applying the distribution in each frame. However, in each frame, the distribution is approximately but not exactly the same as it is shown in section 4.1. Therefore, the achieving T_{DRM} is not accurate.

In conclusion, the multi-core version DRM receiver's processing speed can guarantee a real-time service based on the timing performance on the model which approximately accords with the performance estimation in the last section. Compare with the single-core version on the Cortex R4, the multi-core version has a speed up of $1.7X$ and each frame's processing time is 339 ms in average as it is shown in Table 7.10.

Table 7.10: The multi-core version DRM's speed up

Frame ID	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**AVG**
Single Core *ms*	564	634	558	556	628	557	561	634	556	558	581
Multi Core *ms*	309	384	316	331	379	304	324	378	339	325	338.9
Speed Up	1.8	1.7	1.8	1.7	1.7	1.8	1.7	1.7	1.6	1.7	1.7

Chapter 8

Conclusion

8.1 Conclusion

This project's main purpose is to accomplish a multi-core version of the DRM receiver program on the Ericsson M7400 platform. There are ARM Cortex R4 processor and EVP processor connected with the AXI bus on the platform. In the analysis of the multi-core version, the tool, Pareon is used to help to determine the communication data size in various versions.

The source code of the DRM receiver implementation is from the Dream DRM project. The code and its related libraries are analyzed and isolated to obtain an independent, compatible and minimal program only containing the main DRM receiving process. The corresponding benchmarks are also created to test the DRM's functional correctness and execution time. Before directly porting on the ARM core of the target platform, an intermediate porting step is achieved to port the DRM program on a simulation Cortex-A8 RTSM with a fast simulation speed. Based on the profile report, the DRM program is optimized to increase the execution speed. After that, the modified code is ported on the Cortex R4 model of the Ericsson platform. From the timing performance on the Cortex R4 model, it can be seen that the frame processing time in average is 581 ms which is far away from the real-time requirement. In order to further improve the processing speed, the Viterbi decoder taking 45% of the execution time, is a candidate to port on EVP. An investigation is done to check if the Viterbi related code also should be ported to the EVP. The tool, Pareon is used here to analysis the data needed to be transferred in various multi-core versions. Through the analysis, it is chosen that only the Viterbi decoder function is chosen to port on EVP considering the improvement of the time performance , communication data size, EVP-C realization efficiency. Then the Viterbi decoding process on EVP as well as the communication controlled by DMA between the ARM core and the EVP, are realized. Based on the multi-core distribution and communication design, the timing performance of the multi-core version DRM receiver

is estimated and it is compared with the actual profile report on the multi-core platform model. It can be seen from the profile report that the multi-core DRM receiver delivers a real-time service with a frame processing time of 339 ms in average which is 1.7X faster than the Cortex R4 single-core version.

In summary, the cooperating architecture of ARM and EVP on the M7400 MSS is a suitable architecture to handle wireless communication's demodulation and decoding process. Even a complex receiving implementation like DRM receiver can reach the real-time requirement on the M7400 platform, if a proper multi-core version of the program is chosen. The tool, Pareon is an appropriate tool in the analysis of the multi-core version. The tool is still under development, the future version of Pareon will deeply benefit the analysis or even give a multi-core version performance estimation automatically as it is artificially done in Section 7.1.

8.2 Future Work

In Section 6.2, it discusses the synchronization between the ARM core and the EVP core. It can be seen that when the EVP is working, the ARM core is not involved in any work, but just waits for the finish of the EVP task. For instance, a 2337-step Viterbi decoding's execution time is 0.17 ms, thus there are 0.017 ms which is 70720 *cycle* wasted in the ARM. The ARM is set to sleep in the WFI (Wait for Interrupt) state, when the EVP core is working. An interrupt is sent to the ARM core as soon as the EVP's work is finished and the ARM core continues its work. Thus the energy of the ARM is saved in some degree. Since this project is mainly focused on the timing performance, this design is not applied. However this implementation of the ARM sleep mode can be chosen as an improvement of the power performance in the practical porting on the hardware.

In the analysis of the multi-core version program in Section 5.3, the pipelined processing option of porting the whole MSC channel decoding task on EVP is declined, because most of the code in this part are not vectorizable and difficult to realize in efficient EVP-C. It can be solved by using another Cortex R4 processor on the platform. The whole MSC channel decoding task can be ported on the ARM core B and the Viterbi decoder is still on EVP as it is shown in Figure 8.1. The performance of this 3-core pipeline processing can be estimated combining with the profile report of 2-core version in Figure 7.5 of Section 7.2. We assume that the processing time of a frame in the 2-core version is T_f. The execution time of MSC channel decoding task on the ARM core B and EVP is about $0.25T_f$ and other tasks on ARM core A takes $0.75T_f$. It can be calculated that in the 3-core version, processing time of a single frame is $0.75T_f$ as show in Figure 8.2. Therefore, it is expected that the 3-core version can deliver a speed up of $1.3X$ when ignoring the communication overhead and other latency. This 3-core version can be implemented in the future work and it still needs to discuss if it is worth to gain a $1.3X$ acceleration with adding the energy consumption of an additional ARM core.

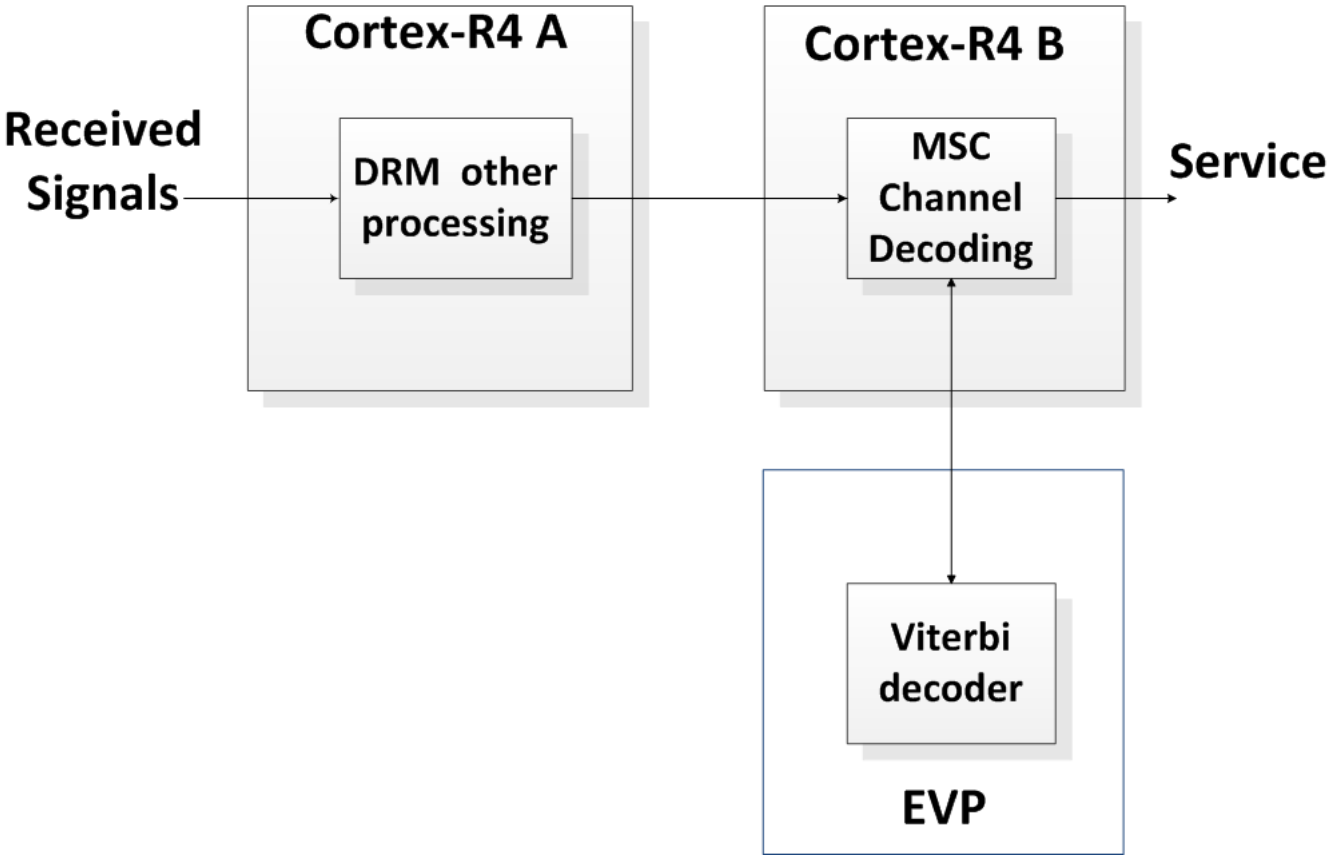


Figure 8.1: The pipelined 3-core version

In this project, we focuse on the main DRM processing including demodulation, demapping and channel decoding. However, the complete DRM receiver should also include the source decoding stage. It can be implemented in a Codec (coder-decoder) processor outside the platform to perform the audio decoding task. What's more, it can be seen that the processing of the first frame takes a large amount of time, for instance, it is about 7 s in the benchmark 1. The antenna continuously receives the radio signal, but the received data cannot be processed immediately. Thus based on the latency of the first frame, a corresponding size of the memory is needed to store the received signals which cannot be handled in time.

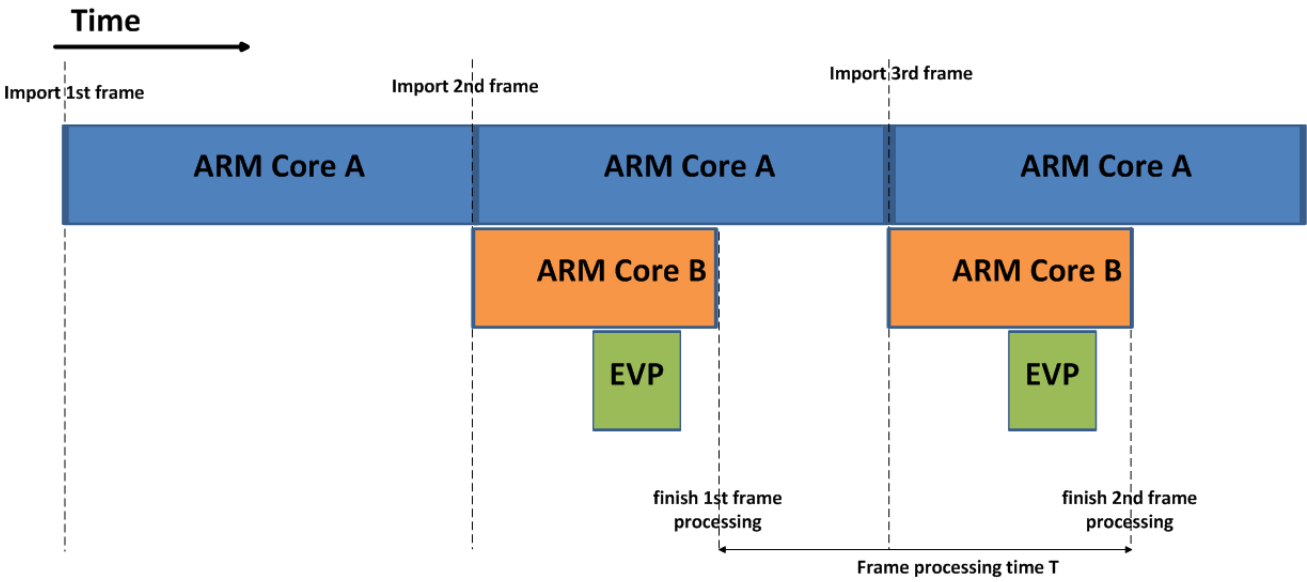


Figure 8.2: The pipelined frame processing in 3-core version

Appendix A.

ARM Linker configuration on Cortex R4

This part discusses the ARM linker configuration on Cortex R4 model. The ARM linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an ARM ELF image [34]. For the library, the link can automatically select the appropriate standard C or C++ library variants to link with, based on the build attributes of the objects it is linking [34]. Therefore, it is not necessary to indicate which library has to be linked.

The ELF image structure is divided into 3 parts, namely RO section, RW section and ZI section. The RO section contains code and Read-Only (RO) data. The RO data includes the addresses of variables that are accessed by the code, floating-point immediate values, immediate values that cannot be loaded directly into a register and so on. Read-Write (RW) data stores in RW section and the Read-Write data which is zero-initialized at image startup is called Zero-Initialized (ZI) data is in the ZI section. [35]

The image is firstly load into memory before execution. In the load view, the RW section and RO section are loaded into the ROM. When starting execution, the ZI section is created and loaded into RAM together with RW section. The load view and the execution view are presented in Figure A..1. The loading address of each section can be defined.

Except the RW section and ZI section, there are stack and heap in RAM. Stack is used to handle processor exceptions. There are 7 kinds of exceptions including Reset, Undefined Instruction, Software Interrupt (SWI), Pre-fetch Abort, Data Abort, IRQ as well as FIQ [35]. The stack is used to store the contents of any registers if an exception occurs and re-store them when returning. Heap defines dynamically-allocated read-write memory. It is used to create new variables, for instance, when the C/C++ language function *malloc()* is called, it will create a space in the heap memory. The address and size of heap and stack also can be configured.

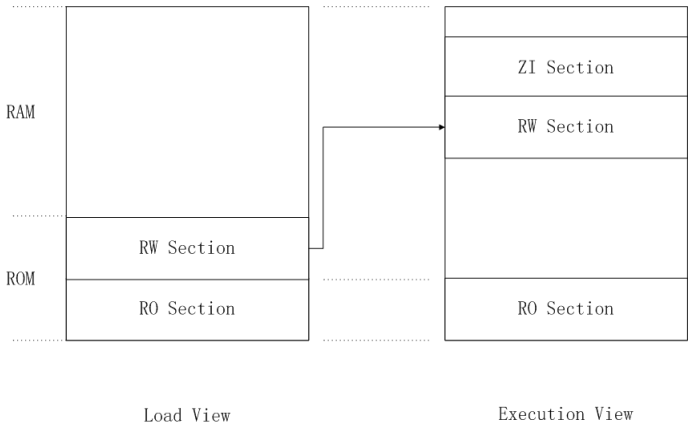


Figure A..1: The load view and the execution view of the memory structure

The boot file is written based on the provided CoWare example program's boot file. The compile option *-info totals* is used in *Makefile* to observe the DRM receiver image information. The information can be seen in Table A..1. The ARM core on the platform is connected to 2 DDR RAMs with the size of 256 MB and 128 MB respectively. These two RAMs are assigned in a unified memory map, when programming on the ARM core, it does not need to pay special attention to the two physical-separated memories. The memory is divided into ROM, RAM parts and works as the memory for the ARM core to load image. In *Makefile*, the compile options *–ro_base 0x00000000* and *–rw_base 0x002fffff* are used to define the image load address. The base address of RO region starts at *0x00000000* and this region covers about 2 MB leaving 3 MB memory space before the RW region. The RW region address starts at *0x002fffff*. These two regions cannot overlap and the empty memory space between two region is left for the possible change of the source code resulting in a larger RO region. The heap memory is located between the end of the RW region and the start of the stack space. The DRM source code requires a large heap memory. If not enough heap memory is assigned, a hardware error will occur while running. Unlike the RO, RW region and heap memory, the stack memory address grows downwards from the base address. In the boot file, the base address and stack size of 7 stack spaces corresponding 7 processor exceptions are assigned. 7 stacks' limit addresses are calculated to find the lowest stack end address which is allocated as the end of heap memory. To leave a big enough space for heap, the stack limit addresses is assigned at *0x09ffffff*. Thus considering RW base address and RW size, a space of 314 MB is assigned to heap. Based on these configurations , a memory structure in the execution view can be seen in Figure A..2.

Table A..1: The DRM receiver ELF image information

	RO Size	**RW Size**	**ROM Size**
Equation	Code+RO Data	RW Data+ZI Data	Code+RO Data+RW Data
Value (Kb)	630.65	1768.89	2385.68

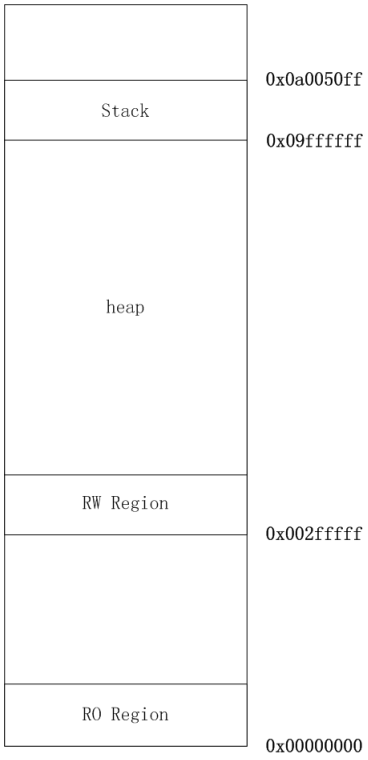


Figure A..2: The execution view of the configured memory structure

Appendix B.

Multicore Communications API's implementation

In order to modularize the multi-core communication achieved between the EVP and ARM processors, and help the further analysis of the Pareon software, the Multicore Communications API (MCAPI) is applied to design the communication interfaces in the DRM two-core version.

The MCAPI defines an API and a semantic for communication and synchronization between cores in embedded system [36]. However, it is a high level application and does not involve the lower level definition including protocol, hardware model and other detailed settings.

MCAPI consists of these major concepts. An MCAPI domain could be a single chip with multiple cores or multiple processor chips on a board and it stands for one or more MCAPI nodes in a multicore topology. The MCAPI node is an independent thread of control, such as a process, thread, processor, hardware accelerator, or instance of an operating system [36]. MCAPI endpoints are socket-like communication-termination points. A node can have multiple endpoints. Thus, an endpoint contains a topology-global unique identifier $< domain\_id, node\_id, port\_id >$.

MCAPI [36] delivers three fundamental communications types, namely Message, Packet Channel as well as Scalar Channel. The latter two types are chosen in our design. Packet Channel transmits connection-oriented, unidirectional, FIFO packet streams and connection-oriented, single-word, unidirectional, FIFO scalar streams are sent by Scalar Channel [36]. Channels are set up at initialization. Once built, the channel can achieve the data transfer with little overhead. When the data reaches the destination endpoint, it is added to an endpoint receive queue. The data typically remains in the endpoint's receive queue until it is received by the application that owns the endpoint.

The design is based on the two-core architecture platform. The platform is defined as

domain_0. The core for the DRM main processing is assigned as *node_0* and another for the Viterbi decoder is *node_1*. Each node contains 4 endpoints to set 4 channels for communication. 2 Packet Channels are implemented to perform the Viterbi input data transfer and Viterbi output data transfer respectively. Scalar Channels are used to acknowledge the finish of the Viterbi decoding and realize the synchronization of the communication. The detailed design is presented in Figure B..1.

The Multicore communication API Working Group provides some head files containing communication functions to help detailed design. The head files used here are *mca.h*, *mcapi.h*, *mca_impl_spec.h* as well as *mcapi_impl_spec.h*.

The first step is to initialize nodes and bind the channel in Figure B..2. Once set up, two nodes can communicate with the binding channel. *mcapi_node_init_attributes* is used to initialize the data type defining the node attribute. Then *mcapi_initialize* is implemented to apply the node attribute to initialize the node. Thus the initialization of the node has been done. The function *mcapi_endpoint_create* is applied to create an endpoint. The endpoint is created dynamically which means a global variable *MCAPI_PORT_ANY* is involved to indicate the next available endpoint on the local node. The endpoint is statically created using the static ID in the remote node and obtained in the local node by *mcapi_endpoint_get*. *mcapi_endpoint_get* is a blocking function. It blocks until the specified remote endpoint has been created or a timeout is reached. After that, *mcapi_pktchan_connect_i* is located to bind the a Packet Channel and the behavior of open a local send port is achieved by *mcapi_pktchan_send_open_i*. While the open function *mcapi_pktchan_recv_open_i* is used in the remote node to synchronize and initialize the local packet receive port. Thus a Packet Channel is bound. The Scalar Channel is created in the same way.

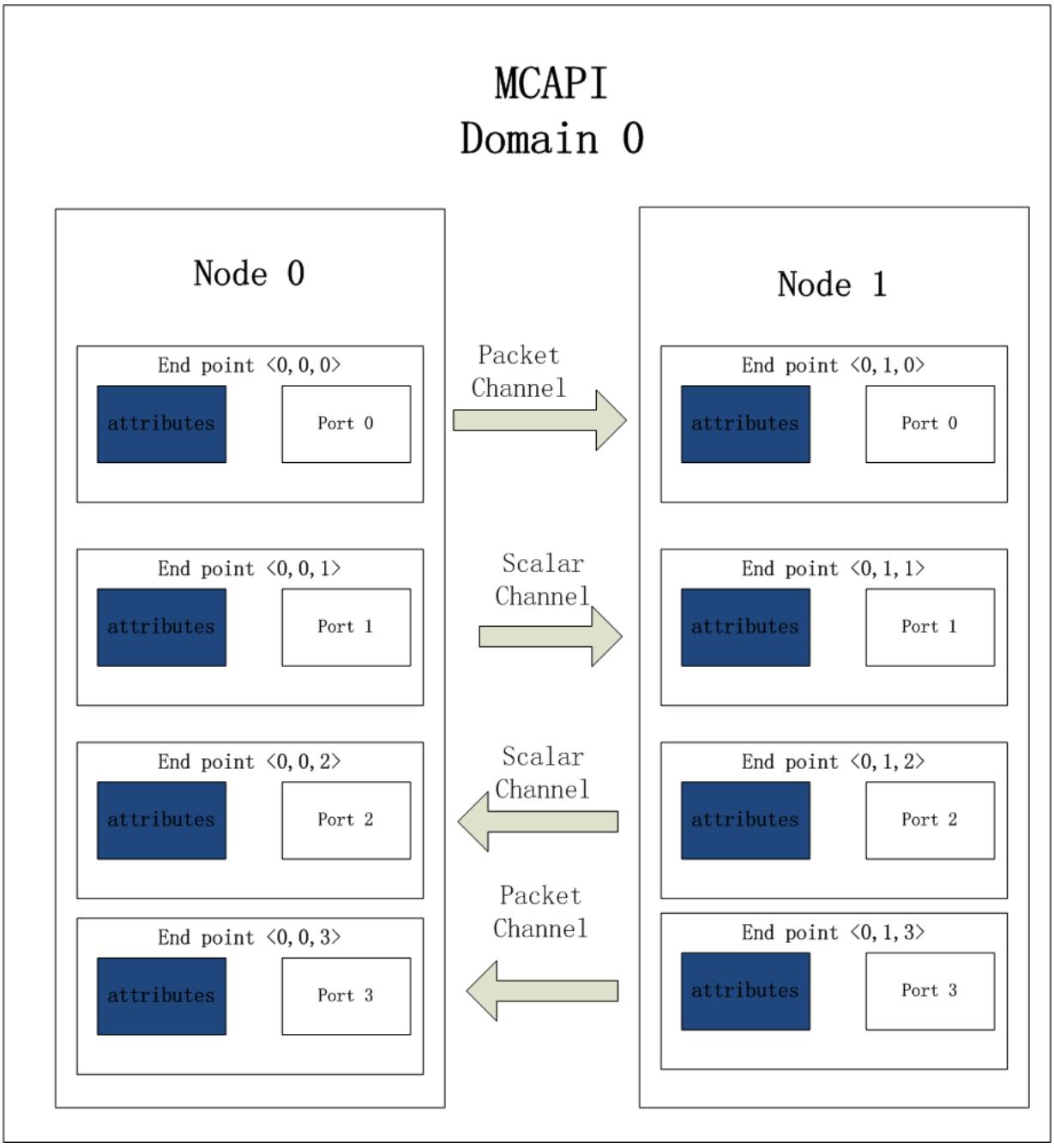


Figure B..1: MCAPI design

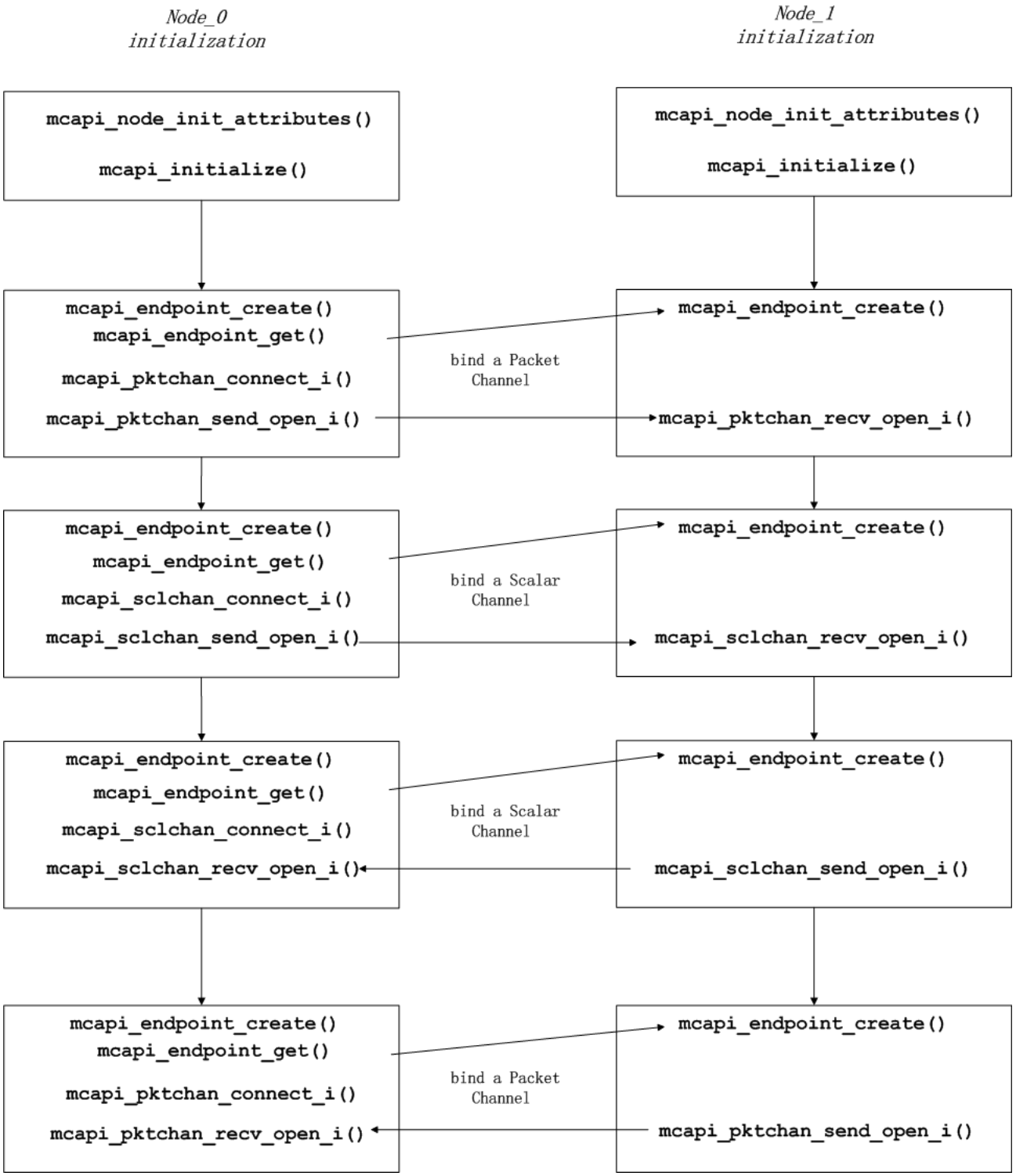


Figure B..2: MCAPI implementation in node initialization and binding channel

The communication is designed on the established channel in Figure B..3. The Viterbi decoding input data waited to send is wrapped into packets and stored in the endpoint buffer. The function *get_next_packet* is applied to judge whether the next package is available. If so, the package is delivered to the remote end point by *mcapi_pktchan_send* through the Packet Channel. Otherwise, it sends a scalar signal through the Scalar Channel, to the remote end to inform the end of transfer. While in the remote end, *mcapi_pktchan_recv* receives the package from the channel. After finishing one time receiving, a judgment should be done by *mcapi_sclchan_available* to detect the informing signal. Once it detects that the transferring has ended, the remote end invokes *mcapi_pktchan_release* to return the packet buffers to the system, thus the received data is available to use and the Viterbi decoding routine can be called. Afterwards, the decoded data is packaged and sent back the acknowledge signal to the local end to tell that the work has been finished. Then a similar transfer as before is launched to send the output data back to the local end by package.

Once the MCAPI design is finished, these API communication functions can be instantiated combing the real hardware. For instance, if it is implemented in the Ericsson M7400 platform, the packet sending function *mcapi_pktchan_send* is written to launch a DMA transfer. After the MCAPI design and instantiation, these API communications can be called in the program to realize the communication design. And the modularization communication is also easy to apply in the various multi-core versions.

For example, if the pipelined multi-core version mentioned in section 5.3 is expected to implemented, the whole MSC channel decoding processing stage is ported on EVP and the communication between the ARM and EVP needs to redesign using the API functions as it is presented in Figure B..4. It is different from the former design because it does not need synchronization. When the ARM's task is finished, it packs the data and stores the packets in the buffer of the endpoint. Then the non-blocking function *mcapi_pktchan_send_i* is called. This function returns immediately and the node does not have to wait the transmit to finish. Once the transfer request is sent, the ARM can continue to do the processing of the next frame. When the packet reaches the destination endpoint, it is added to an endpoint receive queue. In the EVP end, once the EVP finishes the current task, it uses *mcapi_pktchan_recv* to receive the packet from the queue and store in the buffer. When enough data is gained for one time routine, it calls the function *mcapi_pktchan_release* to obtain the data from the buffer and process the routine.

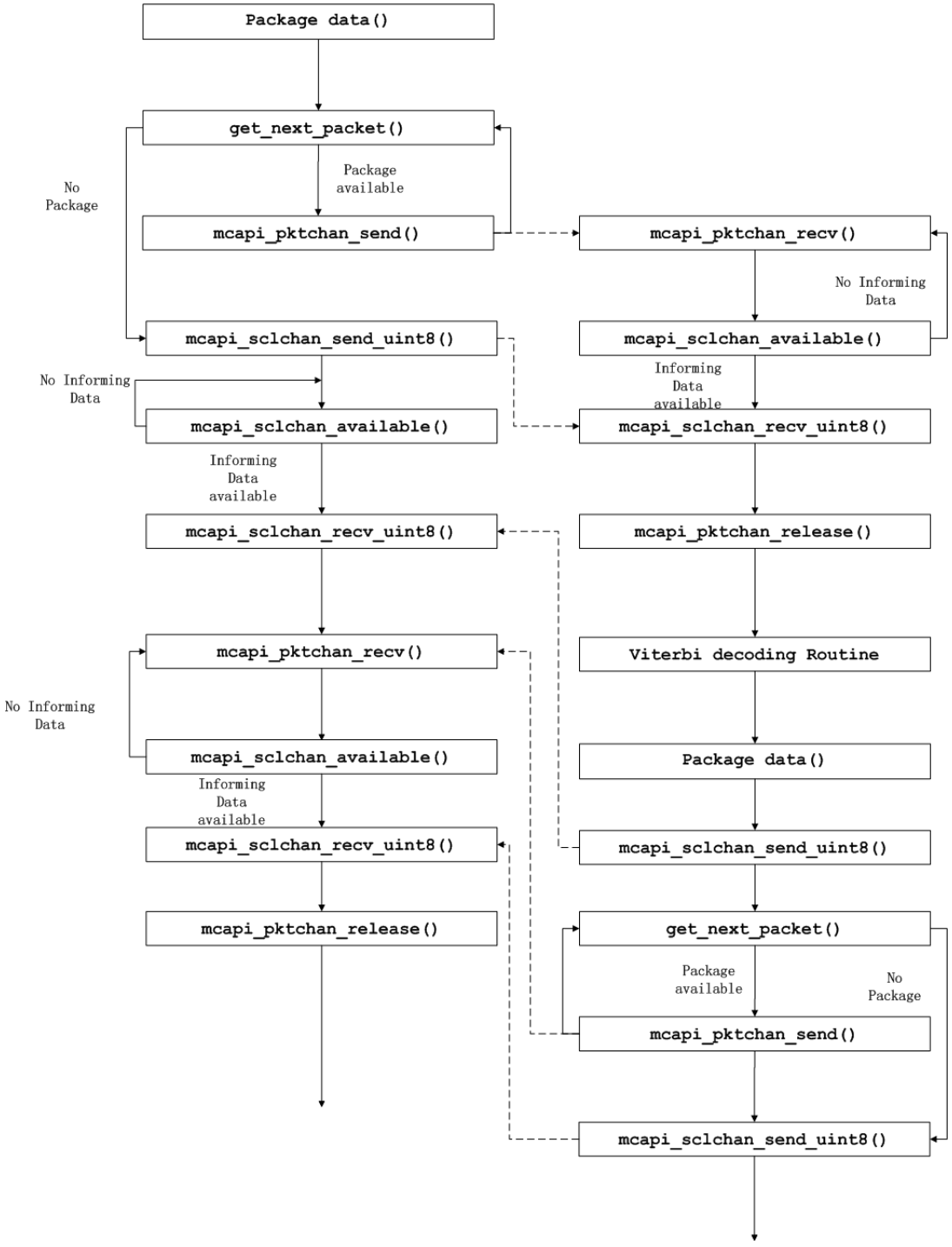


Figure B..3: MCAPI implementation in communication

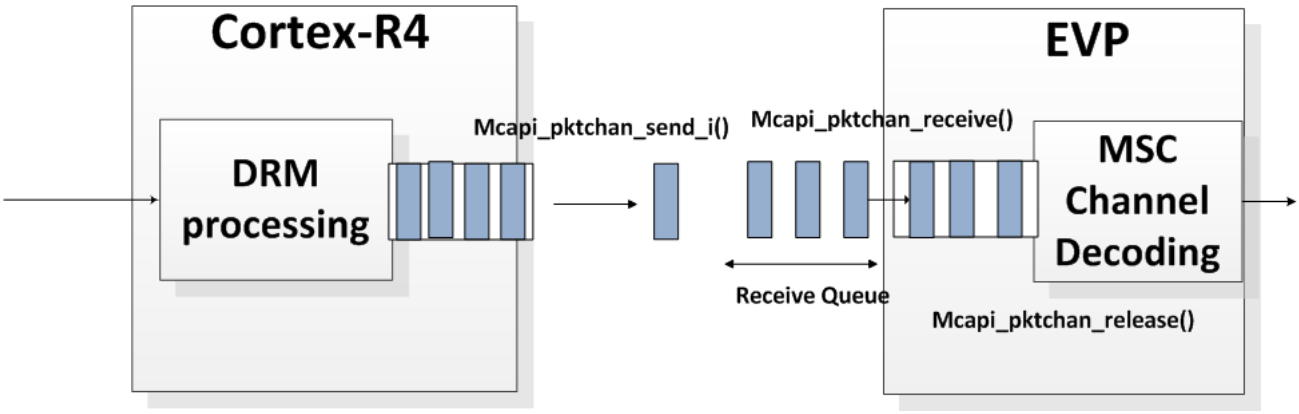


Figure B..4: MCAPI implementation in communication of the pipeline processing program

Bibliography

[1] Hyunseok Lee, Yuan Lin, Yoav Harel, Mark Woh, Scott Mahlke, Trevor Mudge, and Krisztian Flautner. Software defined radio–a high performance embedded challenge. *High Performance Embedded Architectures and Compilers*, pages 6–26, 2005.

[2] Markus Dillinger, Kambiz Madani, and Nancy Alonistioti. *Software defined radio: Architectures, systems and functions.* Wiley. com, 2005.

[3] Mathew NO Sadiku and Cajetan M Akujuobi. Software-defined radio: a brief overview. *Potentials, IEEE*, 23(4):14–15, 2004.

[4] Friedrich K Jondral. Software-defined radio: basics and evolution to cognitive radio. *EURASIP Journal on Wireless Communications and Networking*, 2005(3):275–283, 2005.

[5] Joe Mitola. The software radio architecture. *Communications Magazine, IEEE*, 33(5):26–38, 1995.

[6] Drm introduction and implementation guide. http://www.drm.org/.

[7] TS ETSI. 101980.digital radio mondiale (drm): System specification, 2001. http://www.etsi.org/.

[8] AndrTavares Coutinho. Drm analysis using a simulator of multiprocessor embedded system, 2008. http://ria.ua.pt/bitstream/10773/1950/1/2009000406.pdf.

[9] Lodewijk T Smit, Johann L Hurink, and Gerard JM Smit. Run-time mapping of applications to a heterogeneous soc. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 78–81. IEEE, 2005.

[10] Vector fabrics offical website http://www.vectorfabrics.com/.

[11] AF Kurpiers and Volker Fischer. Open-source implementation of a digital radio mondiale (drm) receiver. In *HF Radio Systems and Techniques, 2003. Ninth International Conference on (Conf. Publ. No. 493)*, pages 86–90. IET, 2003.

[12] Fftw offical website http://www.fftw.org/.

[13] Digital radio mondiale (drm): Mw simulcast tests in mexico, http://www.drm.org/.

[14] Arm offical website http://www.arm.com/.

[15] Katie Roberts-Hoffman and Pawankumar Hegde. Arm cortex-a8 vs. intel atom: Architectural and benchmark comparisons. *Dallas: University of Texas at Dallas*, 2009.

[16] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. A detailed analysis of contemporary arm and x86 architectures. Technical report, Technical report, UW-Madison, 2013.

[17] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schroder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The pure family of object-oriented operating systems for deeply embedded systems. In *Object-Oriented Real-Time Distributed Computing, 1999.(ISORC'99) Proceedings. 2nd IEEE International Symposium on*, pages 45–53. IEEE, 1999.

[18] Arm architecture reference manual armv7-a and armv7-r edition errata markup. http://infocenter.arm.com/.

[19] Cortex-r4 white paper http://infocenter.arm.com/.

[20] Cortex-r4 and cortex-r4f revision: r1p3 technical reference manual http://infocenter.arm.com/.

[21] Kees van Berkel, Anteneh Abbo, Srinivasan Balakrishnan, Richard Kleihorst, Patrick PE Meuwissen, and Rick Nas. Vector processing as an enabler for ambient intelligence. In *AmIware Hardware Technology Drivers of Ambient Intelligence*, pages 223–243. Springer, 2006.

[22] Akash Kumar and Kees van Berkel. Vectorization of reed solomon decoding and mapping on the evp. In *Proceedings of the conference on Design, automation and test in Europe*, pages 450–455. ACM, 2008.

[23] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Soda: A low-power architecture for software radio. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 89–101. IEEE Computer Society, 2006.

[24] Sudeep Pasricha and Nikil Dutt. *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010.

[25] Arm ds-5 version 5.13 eb rtsm reference guide. http://infocenter.arm.com/.

[26] Gcc online documentation. http://gcc.gnu.org/.

[27] Dhrystone benchmark http://www.roylongbottom.org.uk/.

[28] Whetstone benchmark http://netlib.org/benchmark/whetstone.c.

[29] Cortex-a8 revision: r1p1 technical reference manual http://infocenter.arm.com/.

[30] Assessing cortex-r4 and cortex-a8 signal and media processing performance http://www.bdti.com/.

[31] Realview compilation tools version 2.2 compiler and libraries guide http://infocenter.arm.com/.

[32] Kelvin Yi-Tse Lai. An efficient metric normalization architecture for high-speed low-power viterbi decoder. In *TENCON 2007-2007 IEEE Region 10 Conference*, pages 1–4. IEEE, 2007.

[33] John Davis, Andrew Lin, and Ayodele Thomas Njuguna Njoroge. Viterbi algorithms as a stream application. *"signal"*, 1:2, 2002.

[34] Realview compilation tools linker user guide http://infocenter.arm.com/.

[35] Arm developer suite version 1.2 developer guide http://infocenter.arm.com/.

[36] Mcapi api specification v2.015. http://www.multicore-association.org/.