Performance evaluation and optimization of an OMAP platform for embedded SDR systems

CARLO RINALDI

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:21

Performance evaluation and optimization of an OMAP platform for embedded SDR systems

Carlo Rinaldi

Master of Science Thesis
15 February 2011

School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden

Supervisor and examiner at KTH:
Prof. Gerald Q. Maguire Jr

Industrial supervisor at Saab Systems:
Marcus Dahl

Abstract

During recent years, waveform signal processing within a radio system is performed more and more in the digital domain rather than the analog domain. This is exemplified in Software Defined Radios (SDRs) systems. A SDR is a radio system whose components are realized in software rather than in hardware. Among the main advantages of such systems, the most important are flexibility and portability. A SDR system is flexible since its components can be modified and reconfigured without physically modifying the system. Furthermore, a SDR system can be ported to a number of different environments, hence it is not tied to a specific hardware platform. Due to these characteristics, SDRs are being used more and more in both military and public safety sectors.

A straightforward consequence of the adaptability to variable environments is the porting of SDRs to embedded processors and handheld devices. These devices usually have significant limitations both in terms of computational performance and power constraints. Although the trend in the development of General Purpose Processors (GPPs) and Digital Signal Processors (DSPs) dictated by the Moore's Law has increased the performance of embedded devices, currently they face limitations due to both the power consumption and to the execution time when executing even partial SDR systems.

The objective of this thesis project is the evaluation and the optimization of the performance of software running on the OMAP3530 platform on a BeagleBoard. This thesis focuses specifically on the system performances as a function of the configuration of the communication link between the GPP and the DSP in order to reduce as much as possible the system delay due to the communication among the processor cores in the system. Furthermore, this thesis compares the performance achieved by the system by exploiting the DSP and the NEON vector coprocessor. The results of this study show reduced communication delays, thus facilitating the porting of a SDR-like system to an OMAP platform. The experiments were performed on a BeagleBoard Revision C3, a hardware platform based on the Texas Instruments OMAP3530. The OMAP3530 is a processor made up of two cores: the GPP, a 600-MHz ARM Cortex™-A8 Core and an advanced Very Long Instruction Word (VLIW) microprocessor Core, specifically the TMS320C64x+™DSP Core. The communication between the two cores is via the DSP/BIOS Link, software designed by Texas Instruments to facilitate the exchanging of data between the two cores. The optimal DSPLink setup was obtained with the MSGQ module. This offered good performance, while reducing the system power consumption and reducing the load on the GPP. Moreover, the DSP-based solution offered better performance than the NEON-based configuration.

Sammanfattning

Under de senaste åren har signalbehandlingstekniken i radiosystemen övergått till att använda digital teknik snarare än traditionell analog. Ett exempel på detta är Software Defined Radios (SDRs) där många av komponenterna är implementerade i mjukvara istället för hårdvara. De största fördelarna med SDR-tekniken är portabiliteten och flexibiliteten. SDR möjliggör omkonfigurering under drift utan att fysiskt behöva påverka systemet. Dessa fördelaktiga egenskaper har gjort att SDR-tekniken används mer och mer inom civil säkerhet och militära områden.

Även om General Purpose Processors (GPPs) och Digital Signal Processors (DSPs) blir effektivare och bättre med tiden enligt Moores lag, så är porteringsarbetet av SDR system till inbyggda plattformar och handburen utrustning tekniskt utmanande. Utmaningen består av att utrustningen ofta har begränsningar i form av beräkningsprestanda och strömförbrukning.

Utvärderingen fokuserar på optimering av systemprestanda med avseende på mjukvaran som är implementerad på en OMAP3530 plattform. Systemprestandan är beroende på konfigurationen av kommunikationslänken mellan processorkärnorna i systemet. Resultaten av denna studie minimerar fördröjningen i kommunikationslänken mellan processorkärnorna och visar att portering till SDR-liknande OMAP-plattformar är möjlig. Arbetet inkluderar även en prestandajämförelse mellan att utnyttja NEON vector processorn istället för DSP:n som även finns på platformen.

Försöken utförs på en BeagleBoard som är en hårdvaruplattform från Texas Instruments och bygger på OMAP3530. OMAP3530 består av två kärnor: en GPP (600-MHz ARM Cortex™-A8) och en avancerad Very Long Instruction Word (VLIW) Mikroprocessor Core (TMS320C64x+™DSP). Kommunikationen mellan kärnorna bygger på DSP/BIOS Link som är en mjukvara framtagen av Texas Instruments för att underlätta utbyte av information mellan de två kärnorna.

Acknowledgements

First of all I would like to express gratitude to my supervisor at Saab Systems, *Marcus Dahl*, for the support and motivation that he gave me throughout this thesis. I feel enormously indebt to him and to my director at Saab Systems, *Stefan Hagdahl*, for giving me this opportunity. I want to thank also my supervisor at KTH, *prof. Gerald Q. Maguire Jr.* for the guidance and help that he gave me in terms of advice, reviewing, and improving this work. Furthermore, I want to thank all my colleagues of the Security and Defence Solutions Department for creating a perfect environment to work and to spend a pleasant time.

A special thanks to all my *friends* I met, who have accompanied me in this wonderful journey of professional and personal growth started in Torino and ended in Stockholm. Thanks for supporting, putting up with me, and sharing with me unforgettable times.

Finally, I want to dedicate this thesis to my *family*, specifically to my parents, my sister Lucia, my little brother Mario Pio and to my grandma Maria. Thank you for supporting all my decisions and for your love. I hope I did everything possible so that you can be proud of me.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

ADC	Analog to Digital Converter
ALU	Artihmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AVS	Adaptive Voltage Scaling
BIOS	Basic Input Output System
CCNT	Cycle Counter
CSR	Control Status Register
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DPS	Dynamic Power Switching
DSP	Digital Signal Processor
EABI	Embedded Application Binary Interface
ECC	Error Correction Code
FFT	Fast Fourier Transform
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GIE	Global Interrupt Enable
GPP	General Purpose Processor
HDL	Hardware Description Language
IC	Integrated Circuit
IDE	Interactive Development Environment
IER	Interrupt Enable Register
IF	Intermediate Frequency
IPC	Inter Process Communication

IPS	Inter Process Signal
ISA	Instruction Set Architecture
ISP	Image Signal Processor
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
MAC	Multiply Accumulate
MIPS	Million Instructions Per Second
MMU	Memory Managment Unit
MPCS	Multi Processors Critical Section
MPU	Microprocessor Unit
NRE	Non-recurrent Engineering
OMAP	Open Multimedia Application Platform
OSSIE	Open Source SCA Implementation Embedded
PCB	Printed Circuit Board
PMIC	Power Management Multi-Channel IC
PMNC	Performance Monitor Control
POP	Package-on-Package
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RTOS	Real-time Operating System
RTSC	Real-time Software Components
SCA	Software Communication Architectures
SDR	Software Defined Radio
SIMD	Single Instruction Multiple Data
SLM	Standby Leakage Management
SoC	System on Chip
SSH	Secure Shell
TI	Texas Instruments
TLB	Translation Look-aside Buffer
VFP	Vector Floating Point
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
WAVE	Waveform Audio File Format

Chapter 1

Introduction

This thesis project will evaluate core-to-core communications in order to optimize the performance of software running on an Open Multimedia Application Platform (OMAP) platform. The evaluation done in this thesis and its results, will be used in projects related to porting a Software Defined Radio (SDR) system to embedded systems, such as the OMAP platform.

This master thesis represents the final project in my academic studies for the Master of Science in Computer Science Engineering conducted initially at Politecnico di Torino in Torino (Italy) and afterwards at Kungliga Tekniska Högskolan (KTH) in Stockholm (Sweden) as an exchange student in the Erasmus/LLP Double Degree programme.

The project has been carried out at Saab Systems in Järfälla at the Security and Defence Solutions department according to the company's requirements. Saab is involved in the development of products, services, and solutions ranging from military defense to civil security[1].

The present chapter gives a quick overview of the background concerning SDRs (section 1.1) and describes in detail the problems and the goal of the thesis project (section 1.2). Furthermore, the method to be used (section 1.3) and the organization of the complete thesis (section 1.4) of the project are described.

1.1 Background

The development of SDR systems (further details in section 2.1) has taken place over the last several decades. It has been driven by the evolution of radio communication systems from primarily analog processing to digital computation. In our society communicating is essential and radio communication systems play a fundamental role in enabling people to communicate (especially while on the move). A *radio* is a system that receives and transmits signals in the Radio Frequency (RF) part of the electromagnetic spectrum (ranging from 30 KHz to 300 GHz) in order to transmit and receive information. Today radio communication systems are

[1] http://www.saabgroup.com/en/About-Saab/Company-profile/Saab-in-brief/

embedded in many devices commonly used in the everyday life, such as cellular phones, computers, and even vehicles.

Until two decades ago, the only way to build a radio system was to use analog electronic techniques. With the improvements in the Integrated Circuit (IC) technology, as described by the Moore's law, the level of integration, the operating frequency, and the price/performance of Very Large Scale Integration (VLSI) circuits has enabled digital signal processing rather than analog signal processing in radio systems. The main idea behind a SDR system, is to realize a radio communication system where some or all of the physical layer functions are realized by software [1]. In a SDR we can replace the static analog platform with its pre-determined waveforms[2] (as in a canonical radio system) with general purpose hardware that provides the waveform processing as implemented by software.

The benefits of SDRs are manifest and cover different aspects. The main advantages are **flexibility** and **portability**. Since the waveform is software dependent, several types of waveform can be supported by a single platform. This means that a different radio can (often) execute on a single platform just by loading new software or new firmware in memory[3]. In addition, a single waveform can be ported to several different platforms quickly (often) without requiring major modifications. These features clearly lead to economic advantages. On one hand the prototyping time (and so the time-to-market) is considerably reduced since software design can take greater advantage of the design hierarchy than analog systems design. On the other hand the Non-recurrent Engineering (NRE) costs are reduced since the hardware platform can be designed once and then reused for a large number of products. Furthermore, maintenance and upgrading are speeded up since the repairs are mainly installing new bits to fix software bugs, rather than physically substituting components. In some cases, this maintenance or upgrading can be performed remotely without the radio being taken out of service. Moreover upgrading a product is flexible and new features can be installed quickly, remotely, and without the need of any physical intervention. The ability to remotely update the software greatly increases the speed with which upgrades can be deployed while increasing the scalability of the maintenance organization. In addition logistical and operational expenditures are lowered by utilizing a common radio platform for multiple markets. This is especially important for military and civil defense markets where the volumes of products are much lower than for consumer electronics, thus consumer devices can be "upgraded" into military and civil defense products as needed - radically changing the cost of these products.

People may wonder whether SDRs will be effectively adopted by society and

[2]In the SDRs context, the term "waveform" includes all the components needed to create a radio system.

[3]Assuming that the platform has adequate performance to meet the requirements of this radio.

how they are integrated with today's technology. An interesting study related to the adoption of the SDR technology is reported in [2]. In this paper the rate of adoption of the SDR technology is analyzed in different market segments. As we can see from figure 1.1, the SDR technology is well adopted in military communications where even the laggards and sceptics have adopted it. Even if it has been recently accepted in commercial wireless infrastructures, it has not yet "crossed the chasm"[4] concerning the mobile handsets and terminal market segment. The reasons for these trends are explained in [2]. In military environments, the radios targeted for military communication are based on reprogrammable reconfigurable processors. On the opposite side of the spectrum of volumes, although in the mobile handset market SDR technology is not yet mainstream, some steps in this direction have been made in this segment. An example is the Apple's 3G iPhone based on an Infineon baseband processor. It is made up of a Digital Signal Processor (DSP) for the baseband processing and a General Purpose Processor (GPP) for other kinds of computations. This approach of combining a GPP with a DSP has been used in wide area cellular handsets and wireless local area network access points for many years. as it leads to decreased costs, decreased time to market, and increased flexibility.

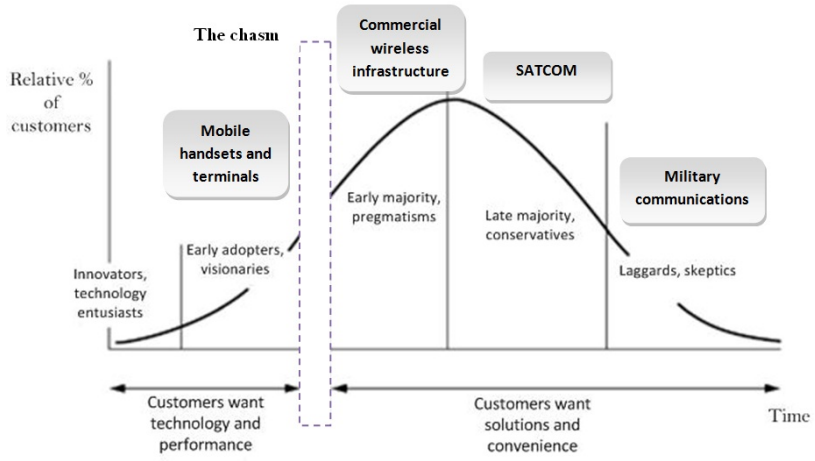

Figure 1.1. Adoption curve in different market segment of the SDR technology (adapted from [2])

[4]The "Crossing the Chasm" concept is described in the Geoffrey Moore's homonymous book [3]. In this book, the author focuses on the specifics of marketing high tech products during the early start up period. A product crosses the chasm when it becomes adopted not only by visionaries (early adopters), but also by the pragmatists who are the early-majority. This is the most difficult step for the product, but it defines at the same time the maturity of the product.

1.2 Motivations and problem statement

In the section 1.1 the current status of the SDR technology in the mobile handsets and terminals market segment was described. Embedded and handheld devices can be considered as part of this market segment. In this section we will take a deeper look at the problems of implementing SDR technology for this class of devices.

Waveform processing can be performed on four different types of hardware platforms and configurations (see section 2.2 for more details): General Purpose Processor (GPP), General Purpose Processor (GPP) + Digital Signal Processor (DSP), Field Programmable Gate Array (FPGA), or Application Specific Integrated Circuit (ASIC). While a large number of SDR products has been developed for running on a GPP (for example, in a desktop computer), the constraints of running on a handheld device and the interest in using SDR on such devices have presented new challenges for SDRs. The user requirements include small size and limited weight, and long battery life (the later achieved by a low power consumption). The challenge is to create SDR systems capable of meeting these constrains when running on embedded devices. One of the most popular tools in the SDR environment is GNU Radio (section 2.1.1), a free software development toolkit that provides signal processing runtime support and signal processing blocks to implement software radios. Although the GNU Radio is platform independent, because it is written using Python, the most critical blocks with respect to the performance are written in C++. GNU Radio was designed for running in powerful GPPs on desktop computers as it makes heavy use of hardware-accelerated floating point computations [4]. The extensive exploitation of floating point operations has limited its use on embedded systems which do not have floating point processors. Nevertheless, some projects are porting SDRs to embedded systems. Two examples are Open SDR[5] which intends to port GNU Radio to the BeagleBoard and OSSIE (section 2.1.2). The later can target a number of different platforms.

This thesis project takes place in this context of efforts to point SDR to embedded GPP+DSP platforms. Although other studies have been done about the performances of embedded systems running a SDR (such as the OpenSDR project, Philip Balister's master thesis [5] and paper [4]), the uniqueness of my thesis project is its focus on the communication link between the GPP and the DSP. Therefore, a great amount of effort was expended to understand what is the best link configuration with respect to the kinds of computations to be performed by the DSP and how much the system gains in terms of performance by using this configuration. The performance achieved by exploiting this configuration, is also compared to the performance that can be achieved by using the NEON vector coprocessor.

For this project, we will focus on the BeagleBoard, a low-cost hardware platform.

[5] http://www.opensdr.com/

BeagleBoard was designed for testing and for experimenting, rather than for developing final products. This board is based on the TI[6] OMAP processor family. The processor on the BeagleBoard is the Texas Instruments (TI) OMAP3530 (see section 2.4). This processor contains an ARM Cortex-A8 GPP and a TI C64x+ DSP. A number of peripherals are also available on the BeagleBoard. The operating system running on the GPP is the Ångström distribution[7], a Linux distribution for a variety of embedded devices. On the DSP side, there is no operating system - simply a Basic Input Output System (BIOS). This DSP/BIOS is a real-time multi-tasking kernel designed by TI specifically to run on DSP platforms. The two cores communicate and exchange data by means of DSP/BIOS Link (aka DSPLink). DSPLink is the basic software developed by TI for the Inter Process Communication (IPC) between the GPP and the DSP.

The goal of my project is to determine and to evaluate the best configuration of the DSPLink (in terms of its IPC mechanisms) in order to minimize the delay of the distributed software running on an OMAP system. The performance analysis will consider three different kinds of performance of the system: the latency concerning the exchanging of data on the link (DSPLink) between the GPP and DSP, the load on the GPP, and the load on the DSP. The results of this study should be used as a basis for the design of software architectures when porting SDRs to the BeagleBoard or in general to the OMAP3530 platform.

1.3 Method

The goal of this research work is the evaluation of an artifact. More specifically, the artifact in question is an OMAP3530 platform. This platform will be evaluated and studied in terms of distributed software performance split across the two cores.

The first phase of the project consisted of gathering information about the OMAP platform and the operating systems to be used. During this phase, a deeper understanding of the hardware capabilities of the system was acquired. The initial main goal of this thesis project was the evaluation of the performance of the GPP+DSP solution as a function of the IPC protocol used for the communication. In this context the DSPLink is in charge of the IPC-based communication between the GPP and DSP .

During the second phase, software to test DSPLink performance was developed. This test software, simulating a typical block of a SDR system, was designed in order to test the performance that could be achieved by using all of the different DSPLink modules.

The third phase analyzed the collected results in an effort to improve the overall system performance. After further study, the NEON vector coprocessor was studied

[6] http://www.ti.com/
[7] http://www.angstrom-distribution.org

and exploited. Test software for targeting the NEON coprocessor was designed and implemented. Using this software, the performance of the GPP+DSP solution was compared with the GPP+NEON solution.

During the last phase, my attention was shifted towards floating point operations. The hardware for the execution of floating point operations was studied and the test software suitably modified to exploit this hardware. Finally an analysis of the collected data was performed to complete my study of how SDR might be realized on the BeagleBoard..

1.4 Thesis organization

Chapter 2 of this report, explains the background of the project. A brief explanation of SDR systems is given, as well as a look at SDR implementations for embedded systems. Additionally, the hardware platform and software running on it are analyzed in detail. The chapter ends with an overview of some previous work done on embedded SDRs and on the analysis of SDR performance.

Chapter 3 introduces all of the tools and methods necessary to analyze the target system's performance. The experimental and development environment are described. Furthermore, the test software for the different system configurations is described and explained in detail. Finally the tools used for performance measurements are described.

Chapter 4 reports on the analysis of the data collected according to the method described in chapter 3. Two proposed system solutions are analyzed and compared. This thesis ends with chapter 5 which summarizes results obtained from chapter 4 and contains some proposals for future works and extensions of this masters thesis project.

Chapter 2

Background

Building upon the motivations for this thesis project and the brief overview of SDR in the chapter 1, this chapter presents the current state of the art regarding the tools used in this project. First, a general explanation of SDR will be given. Section 2.1 introduces the technical basis and the motivation for SDRs without going deeply into technical details. For further details, please refer to [6] and [7]. In section 2.2 the current state of the art regarding embedding SDRs in hardware systems is clarified.

Next focus of this chapter shifts towards technical details of both the hardware and software tools used during the project. Sections 2.3 and 2.4 give an overview of the hardware platform used.

Then, the software tools that will be used during this project are described. In section 2.5 the operating systems running on the OMAP cores is explained in detail, while the section 2.6 gives details concerning the DSPLink software. Finally the chapter finishes by giving an overview of the previous works related to this thesis project (section 2.7).

2.1 Software Defined Radio

A Software Defined Radio (SDR) is "*a radio that is substantially defined in software and whose physical layer behaviour can be significantly altered through changes to its software*" [7]. Hence a SDR is a radio system in which the waveform signal processing is performed digitally. In SDRs a large portion of the functionality is implemented through software. This approach increases the flexibility of the device, as it can change its operating parameters and new features can be added to it without any physical modification to the system. Decades ago, the only way to design a radio system was by means of analog circuits. Thanks to the improvements in VLSI technology, the possibility of realizing radio components (e.g. mixers, filters, amplifiers, modulators, demodulators, detectors, etc.) as software running on personal computers, embedded computing devices, or programmable gate arrays has become a reality.

In an ideal SDR, either digitization occurs at the antenna or following a very flexible Radio Frequency (RF) front-end. This flexible RF front-end is needed in order to handle a wide range of carrier frequencies and modulation formats [7, page 3]. The ideal scheme for a SDR is shown in figure 2.1. The antenna receives the analog radio signal. This flexible RF front-end convertes the analog radio signal into the digital domain by an Analog to Digital Converter (ADC). The stream is then received and processed in a combination of software and hardware. These software and hardware process the waveform. An output waveform is sent as a digital signal to be converted by a Digital to Analog Converter (DAC) into an analog signal. The analog signal is generally amplified and transmitted into the ether by a radio antenna.

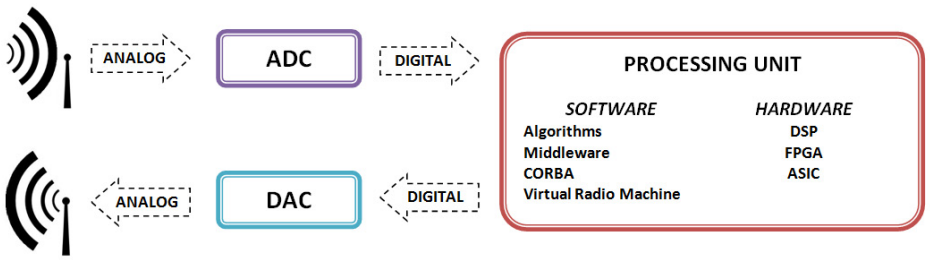

Figure 2.1. Model of an ideal SDR system

A more concrete scheme is shown in figure 2.2. The main difference from the ideal scheme is that an intermediate step before conversion is needed in the receiver. This conversion to an intermediate frequency is required since SDRs must deal with radio frequency signals (ranging from 30 KHz to 300 GHz), but current technology does not allow a signal conversion (from digital to analog domain and vice versa) with both a high enough rate and a sufficient accuracy for frequencies above 35 MHz[1]. This step transforms the received high-frequency signal into a so called Intermediate Frequency (IF). For received signals, this transformation is done by a tuner. Following this the intermediate frequency is filtered and digitized[2]. The filtering is done to prevent aliasing of high frequency signals into the band of frequencies that are being digitized. A similar transformation can be made to shift the IF frequency back for transmission. Both figures 2.1 and 2.2 cite CORBA as a software tool in the processing unit. Common Object Request Broker Architecture (CORBA) is a standard that enables components written in multiple programming languages and running on different computers, to communicate by means of interfaces written by using Interface Definition Language (IDL). According to the Software Communication Architectures (SCA), the transfer

[1]This limit is not a hard limit. The frequency at which direct data conversion can be done increases in the case of multi-GHz processor clocks.

[2]This transformation can be achieved by means of a *super heterodyne* receiver. In order to tune the high-frequency signal into IF, a variable-frequency oscillator, mixer, and filter can be used.

of data between two components in the waveform must be implemented as CORBA remote procedure calls. In this way, CORBA enables components designed by different vendors to work together.

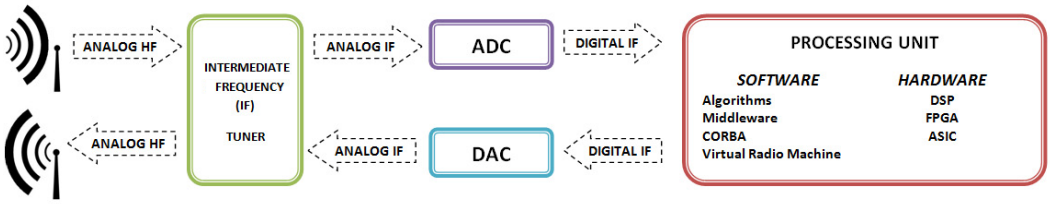

Figure 2.2. Model of a real SDR system

As figure 1.1 showed, SDRs are today widely used in the commercial and military fields due to their benefits [7, page xv, preface]:

- *Ease of design*: traditionally radio systems required years of design experience to be able to design a complex analog system and a deep understanding of the system components interaction was required. Using SDRs the time-to-market of a product can be reduced since a common hardware platform can be reused for a multitude of radio products. Furthermore a deep understanding of the analog part of the system is no longer mandatory.

- *Ease of manufacture*: since the behaviour of analog components varies, huge costs for quality control were common for high quality analog radios. In contrast, the behaviour of processors is more deterministic since given the same input, two processors will generally produce the same output[3].

- *Flexibility in multimode operations*: supporting different communication standards and protocols means loading new software into the SDR without requiring any physical modification of the device. This enables a product to be updated remotely, thus saving money and time.

- *Developing new functionalities*: thanks to the flexibility of SDR, new techniques can be developed giving new capabilities to the radios system, examples include data encryption, voice and speech recognition, data compression, advanced error recovery, interference rejection techniques, and software-enabled power minimization and control. All these functions are implemented by the processor, eliminating the need for further components - hence reducing the system cost and enabling a reduction in the product cost.

[3]While the processors may compute the same output (if the processors are working correctly), there may be a difference in the time when the processors produce this output. As a result there is also an expensive (an extensive) testing process for processors - this process can be used to sort the processors into both functional and non-functional chips, but can also sort them into different performance grades based upon the clock speed at which the processor executes correctly.

2.1.1 GNU Radio

GNU Radio[4] is a free software toolkit for developing SDRs. It provides a library for signal processing, enabling programmers to create SDRs using available low-cost hardware and external RF interfaces. GNU Radio is written in Python. Nevertheless, libraries that involve intensive signal processing tasks are written in C++ for performance reasons. The role of Python is to connect the C++ blocks by using SWIG [5]. The programmer creates a radio system graphically (or logically) by interconnecting blocks. Each block represents a component in the radio system, while the connecting edges represent signal dataflows. An ideal infinite streaming flow of data is processed by each block. In addition, GNU Radio offers the possibility to understand the algorithmic implementation of a radio system and the possibility to modify and create your own custom blocks.

GNU Radio is intended to run on a desktop computer. This means that the basic system should have a 1 or 2 GHz processor with at least 256 MB of RAM [8]. This requirements seems to be ridiculously low compared to the newest desktop machines. For example, a 3 GHz processor could evaluate up to 3 billion floating-point FIR taps/s if a single-cycle floating-point unit is available[6]. However, today's embedded devices do not meet these requirements (we will examine this further in section 2.4).

Since GNU Radio is only a software package, some hardware is required to build a complete SDR system. The Ettus Research (now part of National Instruments) **USRP** is a low-price hardware device designed by Matt Ettus that implements both the receiver and the transmitter in the SDR system. It connects the GNU Radio software with the real world by means a USB 2.0 interface. More recently the company has released an improved device called the USRP2. It is an improved version of the USRP and consists of [10]:

- Two 100 MS/s 14-bit ADCs
- Two 400 MS/s 16-bit DACs
- A Xilinx Spartan 3-2000 FPGA
- Gigabit Ethernet interface
- 1 Megabyte of on-board high-speed SRAM

The FPGA can be used for on-chip processing at the board's high sample rates. The gigabit Ethernet interface enables the board to deliver to applications running on a

[4] http://gnuradio.org/redmine/

[5] http://www.swig.org/. SWIG (this stands for Simplified Wrapper and Interface Generator) is a software development tool that allows programs written in C/C++ to be connected to software written in other higher level languages.

[6] An example of design of a syngle-cycle floating point unit is [9]. In this paper, a single-cycle floating point unit is designed as a pipeline of three stages. Each stage (operands alignment, addition or subtraction of mantissas, and normalization of the result) is performed by a single-cycle unit.

network attached computer samples of up to 50 MHz of RF bandwidth. Moreover the USRP2 is capable of processing signals up to 100 MHz wide. The schematics of USRP project are freely available. In addition, there are drivers to integrate the device into GNU Radio. A variety of daughterboards, sold by Ettus Reasearch, are available to extend the USRP2's functionality.

GNU Radio can be compiled and installed on the BeagleBoard. The GNU Radio package can be compiled by means of `bitbake` or a compiled version can be downloaded and installed from the Ångström distribution package repository website[7] as an IPK package. To install the package, it is sufficient to type in the command shell:
`opkg install <file.ipk>`[8]

2.1.2 OSSIE

The Open Source SCA Implementation Embedded (OSSIE) project[9] is an open source SDR based on the SCA specification[10]. The software is written in C++ using the omniORB CORBA ORB [11]. The current (0.8.1) version of the software is designed to be executed on a Linux operating system and on Intel and AMD processors. Nonetheless, experimental versions have been ported to processors that are widely used in embedded devices. The scope of the OSSIE project is to release a software version with enhanced support for embedded systems. Experimental embedded versions have been ported to the following devices:

- TI 320C6416 DSP;
- ARM 9;
- Marvell PXA270[11];
- PowerPC;
- PowerPC 405.

OSSIE offers a variety of tools for rapid prototyping of a waveform.

OSSIE Eclipse Feature (OEF) This Eclipse plug-in offers a simple drag-and-drop interface to create a waveform. It provides a GUI to create signal processing components and helps programmers to interface OSSIE with CORBA;

[7] http://www.angstrom-distribution.org/repo/

[8] The file name used in this project was `gnuradio_3.1.3-r3.1_armv7a.ipk`

[9] http://ossie.wireless.vt.edu

[10] SCA provides a common infrastructure for the development and managing of SDR based systems. The main goal of SCA is to implement portability and interoperability among the different SDR products, to define commercial standards, support the reuse of waveform design modules, and build on evolving commercial frameworks [6].

[11] This is one of the processors in what was formally known as the DEC StrongARM, then Intel XScale processor family.

ALF This tool helps to debug waveforms. The programmer can launch the waveform, view a block representation of the waveform, and can inject or monitor the state of the signals during the application flow.

Waveform Dashboard (WaveDash) This tool allows users to configure and modify the waveform at run time from a GUI.

2.2 Embedded SDRs

Our presentation thus far has focused on SDR systems suitable for running on desktop PCs (GNU Radio (2.1.1), OSSIE (2.1.2)). This section deals with choosing the hardware to be adopted when porting SDRs to embedded devices. For detailed comparisons among available embedded solutions refer to [12] and [7, chapter 7].

We will start by enumerating differences between different kinds of processing units (see figure 2.1), then look at the main differences between a SDR system implemented on a desktop computer and on an embedded platform [12]. A SDR can have several advantages when running on a desktop personal computer (PC):

ease of use the SDR can be built graphically, interconnecting radio components using a GUI;

computation power usually PCs have powerful CPUs able to perform a large number of operations per time unit. Additionally SDRs can usually exploit fast, single-cycle floating-point units; and

extensive support with respect to drivers and upgrades.

Nevertheless, a desktop PC is not portable, consumes a lot of power, and the operating system is scheduling many processes to run on a single processor (or a small number of processor cores). In contrast, an embedded solution offers:

- low power consumption (from 100 to 400 times lower);
- dedicated hardware: the hardware (and operating system) are dedicated and optimized for specific tasks; and
- potentially lower cost as only the hardware and software resources needed for the target task are needed.

The shortcomings of embedded systems are mostly related to their utilizing a much more constrained set of resources. These constraints make programming more complex.

We will examine several different digital hardware choices for SDRs by comparing them according to the following attributes [7]:

[12] A DSP-based system is used as representative of embedded solutions in this comparison.

Flexibility the ability to handle different protocols and waveforms. The capability of supporting future developments in protocols or technologies is desirable.

Modularity the subsystems must be easily replaced or substituted when new technology becomes available.

Scalability allows the radio to be enhanced with further capabilities and functionalities.

Performance in terms of power consumption, computational power, and relative cost (Prof. Mark T. Smith characterizes this as MIPS/Watt/$).

The main hardware alternatives that can be used to implement a SDR are DSP, GPP, ASIC, and FPGA. Each of these will be examined in more detail below.

2.2.1 DSP

A DSP is a microprocessor optimized for digital signal processing operations. It is optimized to offer high-performance when executing repetitive, numerically intensive tasks, with high-performance I/O. A DSP consists at least of an Artihmetic Logic Unit (ALU), an accumulator, Multiply Accumulate (MAC) unit[13], and buses. A DSP is usually able to perform several memory accesses in a single clock cycle. To achieve that, the DSP architecture breaks the classical **von Neumann Architecture** by implementing a **Harvard Architecture**. The von Neumann Architecture has a single memory interface for both instructions and data accesses, thus a single access to memory can take place in each clock cycle. This dramatically limits the processor performance. In contrast, a Harvard Architecture separates the data and instruction memories enabling one instruction and one data memory access to occur on each cycle; however, this requires two dedicated buses. An improved version of the Harvard Architecture implements several data memories, each with dedicated buses so that every memory can provide data in parallel resulting in multiple memory accesses in a single clock cycle. In the last decades, superscalar implementations have enabled multiple instructions to be fetched, decoded, and executed in parallel, examples are Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) architectures (see sections 2.4.1 and 2.4.2).

Since the DSP's functionality is determined by the executed software, the flexibility, scalability, and modularity of a DSP solution are good. However, this solution typically has high power consumption, but can offer quite high performance when measured in multiply-accumulates per second. This later metric is quite important as many signal processing operations (such as filtering) can be implemented as multiply-accumulate computations. However, one of the most significant limitations of this system is that few programmers are able to get high performance on more than a limited subset of the code, hence limiting the overall

[13] An example of a MAC operation: $a \leftarrow a + b * c$

performance of the system. As a result carefully written libraries of subroutines (often provided by the hardware vendor or a third party) are used by most programmers - enabling them to achieve high performance without needing to understand all of the details of the processor.

2.2.2 GPP

The GPP solution offers very high programmability. Unlike DSP programming, which requires extensive experience and a deep knowledge of the DSP architecture and assembly language to design and implement an efficient algorithm, a GPP can be programmed using higher level languages, while exploiting the operating system and extensive libraries of routines. The achievable performance can reach that of DSP with the introduction of coprocessors and architecture modifications (section 2.4.1).

The main advantage of a DSP over a GPP is the deterministic execution of the code. In a DSP all the hardware and software running on the processor is executing only one task - as the DSP generally does not have an operating system coordinating multiple tasks. In a GPP the operating system scheduler breaks this deterministic behaviour by making extensive use of multitasking, thus this complicating performance analysis of the system. However, in multiple processor (and multiple core) GPPs, one processor might be dedicated to a specific task, thus regaining the deterministic execution of a task. Additionally, real-time operating systems enable deterministic scheduling - but at the cost of increased programming effort and a requirement of deeper knowledge of both the hardware and software.

2.2.3 ASIC

In an ASIC solution, the entire integrated circuit is designed to implement a specific computation at the gate and sometimes even the transistor levels. ASICs are the optimal solution in terms of run-time performance. They are capable of achieving fast execution times with the minimum power consumption. Unfortunately, this is at the cost of greatly reduced flexibility. The cost of the system (both Non-recurrent Engineering (NRE) and production costs) is high and the system design time can be very long. To reduce development time a developer can use a Hardware Description Language (HDL) and purchase the design for entire sub-systems (so called "intellectual property", for example an Ethernet interface, a 48 bits floating point multiplier, etc.).

2.2.4 FPGA

An FPGA is an integrated circuit that can be customized by programmers after having been manufactured. Using a FPGA avoids some of the development costs of the ASIC approach, while offering both flexibility and higher performance than both DSP or GPP based solutions. For the FPGA solution programmers must design their circuit by means of HDLs. Using a FPGA greatly increases the flexibility when

compared to the ASIC approach. In some cases, the FPGA can be reconfigured on the fly. In some cases, different parts of the FPGA can be reconfigured while other parts are used to execute a computation. Depending on the FPGA, there is a wide range of flexibility and modularity. Additionally, the types of gates which different vendors offer range from very simple logic gates to much more complex logic, with some FPGAs offering embedded processor cores, memories, network interfaces, etc. as blocks that the programmer can configure into their circuits. One difficulty is that increased on-chip complexity of blocks increases the cost and decreases the potential flexibility of circuits that can be realized with a given FPGA. Another difficulty is that mapping designs to a given FPGA may be very difficult, with small changes leading to very big differences in performance. However, the performance of FPGAs can be very high since the system functions are still implemented in hardware and they can execute in parallel.

2.2.5 Conclusions concerning alternative solutions

As already stated in the section 2.1, the main advantage of SDRs is their flexibility. For this reason the best embedded solutions for such systems are the GPP, DSP, and FPGA. The main limitation of the first two of these systems is their performance. To increase their performance, a hybrid configuration can be created in which the GPP and DSP cooperate to achieve higher performance. In such a system the GPP controls the DSP and coordinates tasks, while implementing the most computationally demanding operations in the DSP. General purpose I/O operations are performed by the GPP. Although the global system performance is increased, the complexity of programming is increased since the programmer must deal with the communication between the two cores. Furthermore, since data must be sent over a communication channel, the potential parallelism may not be fully exploited. The GPP+DSP configuration, represents the main trend in the integration of SDRs in embedded devices. Nevertheless, FPGAs are used for performance critical tasks where the performance provided by the GPP+DSP system is insufficient. An example of this is the use of an FPGA in the USRP, where the FPGA is used for the signal decimation and for converting a signal to and from baseband[14]. Table 2.1 summarizes the comparisons made in this section. In this table the scores are from 1 (worst) to 5 (best) and they are related to each other.

2.3 BeagleBoard

BeagleBoard is a single-board computer system based on TI's OMAP3530 (see section 2.4). It is able to achieve laptop-like functionality thanks to its performance and to the expansion interfaces and peripherals available on the board. In addition

[14]The USRP FGPA can also be used to perform other signal processing that requires both high performance and direct access to the samples, such as the recognition of the start of a WLAN frame and timestamping as shown in [13].

Table 2.1. Comparison of embedded SDR solutions (adapted from [12])

Solutions	DSP	GPP	FPGA	ASIC	GPP + DSP
Flexibility	5	5	3	1	5
Performance	2	1	4	5	3
Programmability	4	5	2	1	4
Development cycle	5	5	3	1	5
Cost	5	4	3	1	4
Power consumption	2	2	4	5	1

to its performance, it is at the same time a low-power and low-cost embedded computer system. At the time of writing, the cost of a BeagleBoard-xM is US$ 149. This board is targeted at the Open Source Community. Since some key features of the OMAP system are missing (in fact the intefaces of the OMAP for the high speed data transfer are not exposed), it is not intended to be used in a final product, but it is designated as an experimental and test platform [14]. The BeagleBoard used during this project was the version BeagleBoard Revision C3. The table 2.2 shows the key features of this board. The core of the BeagleBoard C3 is the **OMAP3530 ES3.0** [15] processor (2.4) packaged in a Package-on-Package (POP). In the POP packaging techniques, the memories chips are mounted on the top of the processor package. The version of the BeagleBoard that was used is shown in figure 2.3.

With regard to the **memory**, in the Micron POP there are two integrated memory devices: a 2 Gb NAND x 16 (256MB flash memory) and a 2 Gb MDDR SDRAM x32 (256MB @ 166MHz). These two devices are the only on-board memory available. Nevertheless, since BeagleBoard has standard interfaces for connecting external storage devices. Additionally, it is possible to extend the system memory by means of SD or MMC cards or by an USB flash or hard drive. However, accessing these external memories will be quite slow.

TI's TPS65950 chip is used for **power management**. The TPS65950 is a Power Management Multi-Channel IC (PMIC) solution. In a single IC a multichannel power-management device and an audio coder/decoder are integrated. This chip in charge of controlling the power for the both peripherals and for the OMAP processor.

A 14-pin JTAG interface is also provided to permit software **debugging** and programming of the on-chip FLASH memory (i.e., to install a system image or boot loader). Support for RS232 via UART3 is provided by a 10 pin header. Through this interface is it possible to access the BeagleBoard using a IDC to DB9 flat serial cable.

2.4 OMAP3530 Microprocessor

The Texas Instruments Open Multimedia Application Platform (OMAP) is a family of microprocessors specialised for multimedia applications and designed for portable

[15] ES3.0 refers to the silicon version of the OMAP processor

Table 2.2. Key features of BeagleBoard C3

BeagleBoard Revision C3 Features	
Processor	OMAP3530 ES3.0 600 MHz
Memories	2Gb NAND (256MB) 2Gb MDDR SDRAM (256MB)
PMIC TPS65950	Power Regulators Audio CODEC Reset USB OTG PHY
Debug support	UART 14-pin JTAG LEDs GPIO pins
HS USB Host Port	Single USB HS Port (up to 500 mA power)
Audio connectors	L+R out (3.5 mm) L+R stereo in (3.5 mm)
SD/MMC Connector	6 in 1 SD/MMC/SDIO 4/8 bit support, Dual voltage
Video	DVI-D S-Video
Power Connector	USB Power DC Power
Printed Circuit Board (PCB)	3.1" x 3.0" (78.74 x 76.2mm) 6 layers

and embedded devices. Due to these characteristics, the OMAP microprocessors have been extensively utilized in cellular phones.

There are three groups of microprocessors in the OMAP family. Each segment is distinguished from the others by its performance and intended application:

- *High performance*: these processors are intended to be used in smart phones or handheld devices. Such devices need sufficiently powerful processors to run embedded operating systems (typically a Linux or Symbian OS), to support mobile connectivity and multimedia applications. The following processors families belong to this segment: OMAP1, OMAP2, OMAP3, and OMAP4;

- *Basic multimedia*: they are intended for handset manufactures and their main feature is low-cost and high degree of integration. The OMAP331 and OMAP310 are examples of such microprocessors, while the DMx series of digital media coprocessors are used to support advanced cameras on some mobile devices;

- *Modem and applications*: this segment features low-cost and low-power, low-

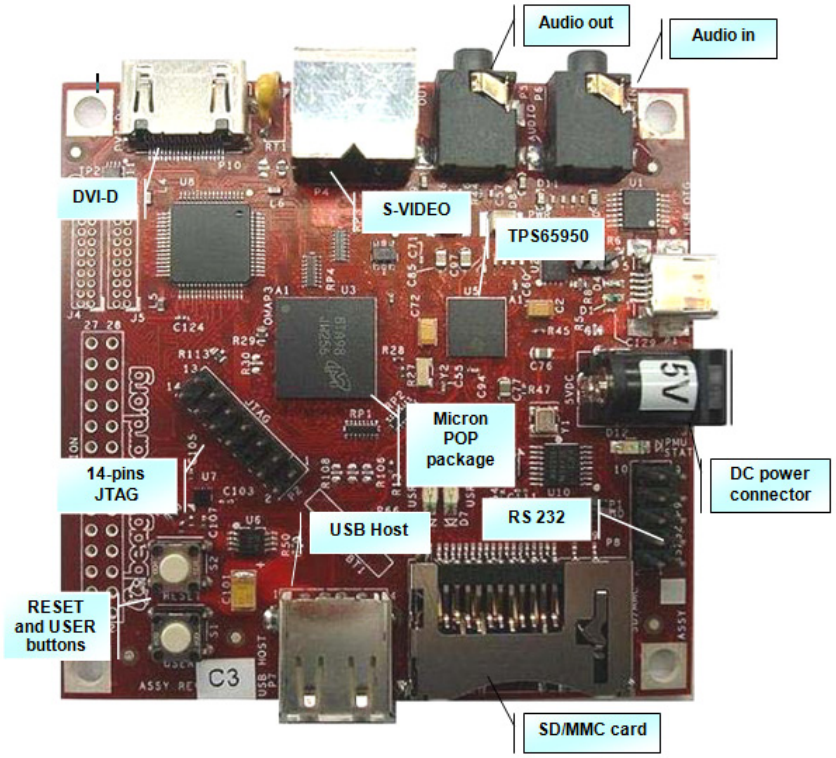

Figure 2.3. BeagleBoard overview

frequency microprocessors. These are primarily intended for simple mobile phones.

The BeagleBoard that we have used is based on the OMAP3530 microprocessor. The OMAP3530 is a dual-core microprocessor belonging to the OMAP3 family, hence it is in the *high performance* segment. As reported by Texas Instruments in [15], the OMAP3 architecture is designed to provide video, image, and graphics processing. The computation power of this architecture is sufficient to support media applications such as streaming video, 3D mobile gaming, video conferencing, and high-resolution still images. The OMAP3530 is able to support operating systems such as Linux or Windows CE. The subsystems that compose the device are [16] :

- **ARM Cortex™-A8** Microprocessor Unit (MPU) (up to 720 MHz);
- TI **C64x+ DSP** (up to 520 MHz);
- Imagination Technologies POWERVR SGX™subsystem for 3D graphics acceleration;

[16]In our project, the OMAP3530 ES3.0 includes ARM Cortex-A8 processor (revision r1p3, 600 MHz) and TI C64x+ DSP (480 MHz)

- Image Signal Processor (ISP) for the processing of different images formats;
- level 3 (L3) and level 4 (L4) interconnects for high speed data transfer with memory controllers (either external or on-chip ones).

Furthermore advanced services are implemented in the OMAP3530. A remarkable capability of the system is its **power management**. The active power consumption is reduced due to automatic control of the operating voltage of individual modules and by supporting the SmartReflex™technology [17]. In an OMAP3430 this reduces active power consumption by 66 percent and standby power leakage by up to three orders of magnitude[16]. For readers interested in the advanced features and in the details of the OMAP3530 microprocessors, please refer to [15]. Programmers can refer to [17].

2.4.1 Cortex-A8 Processor

The Cortex-A8 processor is a microprocessor designed by ARM Holdings based on the ARMv7-A, a 32-bit Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA). The Cortex-A8 is a low-power, high-performance single core microprocessor designed for portable devices having the following main features[18]:

- *frequency* from 600 MHz up to 1.5 GHz;
- Dhrystone performance [18] is 2.0 DMIPS [19] / MHz;
- a *superscalar* processor with two different pipelines. The first pipeline is in charge of the execution of integer ARM instructions. The second pipeline is a NEON pipeline for the execution of advanced SIMD and Vector Floating Point (VFP) instruction set;
- *dynamic branch prediction* with branch target address cache, global history buffer, and 8-entry return stack;

[17] Developed by Texas Instruments, this technology consists of a set of hardware and software techniques for dynamic control of power consumption, voltage, and frequency in mobile devices. It guarantees a trade off between a limited power consumption budget and enhanced multimedia application performance field. The techniques involved covers different design levels. At the silicon level, the contribution of the static leakage power is reduced. At hardware level, Adaptive Voltage Scaling (AVS), Dynamic Power Switching (DPS), and Standby Leakage Management (SLM) technologies are used. At software level, an open software framework assures compatibility between low hardware level and the OS's power managers[16].

[18] This refers to the performance as measured by means of the Dhrystone benchmark. This benchmark was created in 1984 by Dr. Reinhold P. Weicker and it tests integer computation performance of a processor without any floating-point operations. It became popular since it is free of charge, while the most popular benchmarks belonging to the SPEC suite are quite expensive. However, it has several notable limitations as it does not consider many important factors such as the RISC nature of the processor, multitasking, memory hierarchy, and advanced processor designs (as found in superscalar and VLIW computers)

[19] Dhrystone Million Instructions Per Second (MIPS)

- Memory Managment Unit (MMU) and two 32 entries Translation Look-aside Buffers (TLBs) for data and instruction (respectively);
- static and dynamic power management;
- L1 instruction and data cache of 16KB or 32KB (configurable size). The L1 cache is integrated on-chip so that it can be accessed in a single clock cycle;
- L2 cache up to 1 MB configurable size with parity and Error Correction Code (ECC) techniques implemented. The L2 cache is banked so that only the bank in question is activated for increased power saving.

Three technologies implemented in the Cortex-A8 are noteworthy for our project. The first one is the **Thumb-2** instruction set, an extension of the earlier Thumb instruction set. When the processor is in the Thumb instruction set state, it is able to execute variable-length instructions. In this state the instruction length is not fixed at 32 bits, but can be either 16 bits or 32 bits temporary breaking the RISC model. The main advantage is to reduce the *instruction code size*. This aspect can be very important when dealing with embedded devices with a limited amount of main memory. The short instructions (16 bits) utilize implicit operands or limitations of the more general instruction set. In fact only a limited set of operations can be expressed through these 16 bits instructions. Thumb-2 is an enhancement of the Thumb technique as it introduces the possibility to interleave 16 bit instructions with 32 bit instructions while still in the Thumb instruction set mode.

The second technology is the **Vector Floating Point (VFP)** architecture. This consists of a coprocessor extension of the ARM architecture capable of executing floating point operation with half, single, and double precision. It is fully compliant with the IEEE 754 floating point format.

Third is the **NEON** technology [19], a 128 bit SIMD architecture extension. Thanks to this, the Cortex-A8 is able to execute advanced SIMD instructions. SIMD is a class of parallel execution that exploits parallel operations on data. NEON is considered a *short-vector* architecture, this means that registers are considered as vectors of elements of the same type of data and the same operation is performed in parallel in different lanes (see figure 2.4). The data types available in this SIMD instruction set are signed and unsigned 8 bits, 16 bits, 32 bits, 64 bits and single precision floating point. This technology provide a significant acceleration in the performances of multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and speech processing, and image processing. The motivation for this is that in such applications it is very common that an operation is to be performed on an array of data, this is naturally highly parallel, hence it is well suited to a SIMD instruction set.

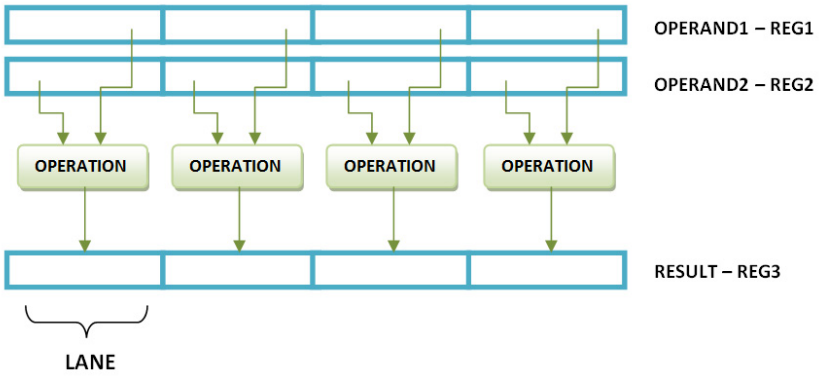

Figure 2.4. SIMD architecture

Unfortunately, the VFP technology is optional and according to [15], it has not been included in the OMAP3530 processor utilized by the BeagleBoard. However, the NEON technology is important for this project as it helps porting of SDRs to the OMAP processor as floating point operations are supported. The VFP functionality should be taken into account in future extensions of this work (for example, when implementing a speech CODEC on a different version of this platform).

2.4.2 TMS320C64x+ DSP

The TMS320C64x+ DSP is a VLIW architecture that executes up to eight 32-bit instructions per cycle ([20]). This is possible because in the CPU architecture 8 functional units are present. These functional units are divided into:

- 6 ALUs (single 32 bit, double 16 bit, or quad 8 bit arithmetic operations per clock cycle);
- 2 multipliers (two 16x16 bit multiplies or four 8x8 bits multiplies per clock cycle).

This DSP processor includes sixty-four 32-bit general purpose registers. The TMS320C64x+ benefits from its VLIW architecture[20]. The main advantage is due to the grouping instructions. This reduces the number of instructions that are produced for a given amount of code (hence less memory is needed), thus the number of fetches from the instruction memory is reduced (resulting in less power being consumed), and the execution time is reduced by exploiting the instruction

[20]VLIW architecture is a static way for exploiting the instruction level parallelism of a program. The compiler package groups of instructions that can be executed in parallel into longer instruction at compile time. This means that when the CPU executes one VLIW, several single instructions are executed in parallel each clock cycle. Due to the static nature of this technique, it is not able to exploit optimally all of the potential instruction level parallelism.

level parallelism.

The C64x+ is a fixed-point DSP. This implies that floating point operations are not executed in hardware, but rather are emulated by software. Nevertheless software performance can be improved by using TI's *IQmath Library for C64x+* (details in [21]). This library is a collection of highly optimized mathematical functions (written as C/C++ routines) aimed for porting floating-point algorithms to fixed-point code that can be executed by the C64x+ hardware. Another useful tool for improving performance of the software running on the DSP, is the *TI C64x+ DSPLIB* [22]. DSPLIB is a collection of high optimized C-callable routines that are written in assembly code. Most of these routines are used for signal processing, especially in computationally expensive real-time applications. The functions in the DSPLIB are organized into seven different categories:

- Adaptive filtering
- Correlation
- Fast Fourier Transform (FFT)
- Filtering and convolution
- Math
- Matrix and
- Miscellaneous.

2.5 OMAP3530: Operating Systems

A variety of operating systems can execute on OMAP processors. The ARM GPP is in charge of most of the platform functions including the control and the coordination of the DSP. While complete operating systems can be executed on the GPP, a simple Basic Input Output System (BIOS) is sufficient for the DSP, as the DSP is used for real-time computation and I/O[21], leaving the other tasks to the GPP. The operating systems that can be executed by the ARM GPP are Linux®, Symbian OS™, Microsoft's Windows Mobile™, and Android™. The BIOS that is supported by the DSP is the TI's DSP/BIOS Real-Time Operating System (section: 2.5.2). During this master thesis project, the Linux Ångström (2.5.1) distribution was used as the GPP operating system and DSP/BIOS Real-Time OS (2.5.2) was used on the DSP.

2.5.1 Ångström Distribution

Ångström [22] is a Linux distribution intended for the embedded devices. It claims to be versatile and scalable. It can be installed on systems having from 4 MB

[21] As already stated, the BeagleBoard does not expose the intefaces of the OMAP for the high speed data transfer

[22] http://www.angstrom-distribution.org/

to terabytes of memory. The Angström distribution is based upon the union of the OpenEmbedded, OpenZaurus, and OpenSimpad projects. The OpenEmbedded Project is a framework to create Linux distribution for embedded systems.

An important tool in the OpenEmbedded projects is **bitbake**. This is a tool used to build packages for the embedded distribution by means of *cross-compiling*. A cross-compiler produces executable code for a platform and a system that is different from the platform hosting the compiler. This technique is fundamental for building of software packages for those systems where compilation is not feasible due to limited system resources, processor speed, and ported compilers and OS.

2.5.2 DSP/BIOS™Real-Time OS

The Texas Instruments DSP/BIOS is a real-time multi-tasking kernel that was designed to run on DSP platforms, specifically the TMS320C6000, TMS320C5000, and TMS320C28x families. It does not require any license fee and it is available both standalone and integrated in the Code Composer Studio Interactive Development Environment (IDE) tool. DSP/BIOS offers many functions in order to support complex applications within the constraints and deadlines typical of real-time applications. It aims to achieve a minimal memory footprint by using configurable modules that can be excluded by the kernel if not used. From now on, the name of these modules will be indicated with the string of capital letters between parentheses as listed in table 2.3. Currently DSP/BIOS Real-time Operating System (RTOS) is available in two different versions: 5x and 6x. The latter version has the same core as the former one, but more effort has been made to increase the portability of the code and to add more Inter Process Communication (IPC) protocols. DSP/BIOS 5.x was adopted[23], hence our explanations will mostly focus on it.

From the point of view of **multithreading**, DSP/BIOS RTOS provides several kinds of threading mechanisms that can be classified into four levels depending on their priority [23] (see figure 2.5). The threading mechanisms are listed in descending order depending on the priority (remembering that a higher priority results in lower latency during the execution of a task):

- *hardware interrupts (HWI)*: these interrupts are triggered in response to external asynchronous events involving the DSP. A DSP, hardware interrupt is usually raised by a device (either on-board or external). After having received the hardware interrupt, the processor stops its normal thread of execution and executes the so called Interrupt Service Routine (ISR) that handles a critical task affected by a hard deadline. Interrupts should be limited to critical tasks that need to run at frequencies approaching 200 kHz (i.e., with a deadline within 2-100 microseconds). Since interrupt handler threads have a high priority, these threads can only be preempted by other HWI threads with an higher priority. Clock functions (CLK) are part of this category, since they are triggered by a hardware timer. Nevertheless it is possible to temporarily

[23]The reason of this choice are explained in section 3.2

Table 2.3. Kernel modules in DSP/BIOS RTOS

Kernel Module	**Description**
Hardware Interrupts (HWI)	Hardware interrupt manager
Software Interrupts (SWI)	Lightweight preemptible threads that use the program stack and can not yield the processor
Tasks (TSK)	Independent threads of execution that can yield the processor
Periodic Functions (PRD)	Lightweight threads time-triggered
Message Queues (MSGQ)	Variable length zero-copy messages
Mailboxes (MBX)	Synchronized data exchange
Locks (LCK)	Binary semaphores
Semaphores (SEM)	Counting semaphores
Clock (CLK)	Interface with the hardware timer
Streams (SIO)	Streaming I/O
General I/O	Extensible general I/O
Memory Manager (MEM)	Low overhead dynamic memory allocator
Buffer Manager (BUF)	Fast and deterministic fixed-sized buffer allocator
Power Manager (PWRM)	Power manager module
Cache Control (BCACHE)	Cahce controller and configurator

disable hardware interrupts to avoid lower priority tasks being preempted. This is done by setting the Global Interrupt Enable (GIE) or Interrupt Enable Register (IER) bits in the Control Status Register (CSR) depending on whether you wish to enable or disable interrupts.

- *software interrupts (SWI)*: these interrupts have lower priority than HWI. They invoke lightweight preemptible threads that share a common stack. Therefore the latency is reduced since a backup copy of the stack is not needed, thus saving time during the context switch. While HWI are generated by external hardware, software interrupt are due to particular SWI calls in the program. Programmers can use SWIs to handle tasks that need to be completed within a deadline that is longer than the HWI (a SWI task is for tasks that require completion within 100 microseconds or more). Both HWI and SWI interrupt service routines runs to completion. They can be preempted only by a hardware interrupt or by another software interrupt with a higher priority. According to the RTOS policies, the higher the priority integer associated with a thread, the higher the priority of that thread. There are fourteen priority levels available for software interrupts. If two threads have the same priority level, they will be served on a first-come first-served basis. *Periodic functions* (PRD) belong in the software interrupts threads category.

- *tasks (TSK)*: tasks are independent threads of execution that can voluntarily

yield the processor, additionally they can be preempted. The main difference between a TSK and a SWI is that the former can be preempted by any higher priority thread while waiting for available resources. Programmers should use inter process communication and synchronization mechanisms only between tasks since these mechanisms usually require the task to be suspended in order to synchronize with other tasks. There are 16 possible levels of priority. If the priority of a task is -1, then the task will be frozen until its priority level becomes at least 1.

- *background thread*: the background thread executes the idle loop (IDL) with the lowest possible priority, level 0. This loop runs continuously and is preempted by higher priority tasks. Note that the idle loop can be used to execute functions with soft deadlines.

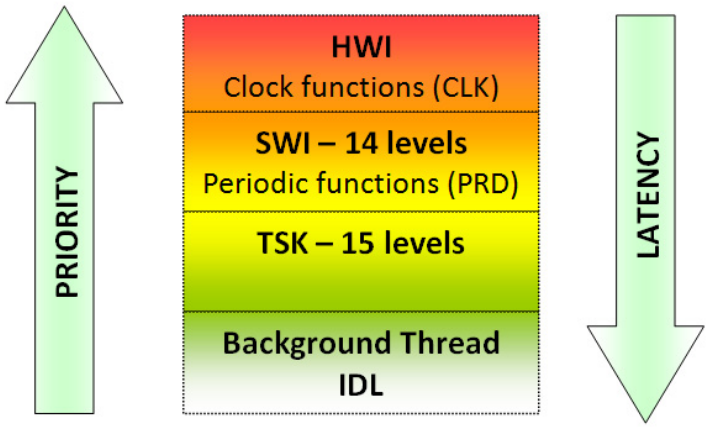

Figure 2.5. DSP/BIOS thread priorities (adapted from [23])

DSP/BIOS offers fundamental **IPC mechanisms** for synchronization and communication among threads. The basic structure is a *semaphore* that is used both for protecting resources from multiple accesses (i.e., to enforce mutual exclusion) and for the coordination and synchronization of concurrent threads (using a counting semaphore). A counting semaphore is a structure that keeps a counter of available resources. Usually the counter is initialized after creation with the number of available resources or to 1, depending upon whether it is a synchronization counter or a binary counter. A task that wants to gain access to a shared resource or to a critical region of code, must call the `SEM_pend` function. If the semaphore counter is greater than 0, than the thread can continue execution. Otherwise the thread is suspended until another concurrent thread leaves the critical region by calling the `SEM_post` function. DSP/BIOS permits the programmer to define the maximum duration of the waiting time by specifying the timeout parameter as an argument to the `SEM_pend` function. Besides semaphores, DSP/BIOS 5.x offers other two

IPC mechanisms: *mailboxes (MBX)* and *message queues (MSGQ)*. A mailbox is a synchronous way to exchange messages among tasks on the same processor (the DSP in our case). The exchange is synchronous implying that both the sender and the receiver must be ready to send/receive the message. In this sense, a mailbox is both a synchronization and a communication tool. In DSP/BIOS RTOS, exchanged messages must be fixed-size. The programmer can set the fixed size of these messages taking into account the limited memory that is available. If mailboxes are expected to exchange messages among threads located on multiple cores, then a *message queue transport* module is needed. Such a module is not available in the kernel, but it provided by the DSPLink (section 2.6). Message queues are used for asynchronous communications: the message is stored in a memory location and is retrieved by the receiver as soon as it is ready. There is no need for both actors to be ready for the communication. The message queue can handle variable-length messages between tasks.

DSP/BIOS provides a set of services for **interrupt management** aimed to maximize flexibility, while reducing data memory requirements. ISRs can be written by developers in C language thanks to the interrupt dispatcher. The dispatcher automatically calls assembly language macros that perform the following system level operations:

- ensure that SWI and TSK scheduler execute at the correct time;
- disable/enable hardware interrupts while an ISR is executing and
- save/restore any register or control word before/after the ISR execution.

By using such an interrupt dispatcher, the code complexity is reduced, thus avoiding inserting low level operations code to each ISR. Additionally, the data memory required is limited thanks to the existence of a separate *system stack*. A dedicated, shared system stack is used during the execution of both software and hardware interrupts, while each task uses its own stack. If no tasks (TSK) are running in the system, then all remaining threads (SWI and HWI) share the same system stack. This leads to performance benefits since:

- the system stack can be smaller since every task has its own stack. The smaller the stack, the faster the memory that can be used to realize this stack;
- in a system serving only ISRs from software and hardware interrupts, there is no need to save and restore the stack during context switching, significantly reducing the time needed to perform context switching.

To minimize the memory footprint DSP/BIOS supports TI's multicore DSP solution: when many cores should load the same system image, DSP/BIOS provides shared image support avoiding loading of image on each core. In this way memory usage is improved since less local memory is needed, enabling this local memory

to be used as local data memory. The MSGQ core-to-core communication module should be enabled for this.

For **power management**, DSP/BIOS focuses on minimizing power consumption while still meeting performance constraints. To achieve this, a range of power management features is provided. Some of them are:

- idling the CPU when the IDL cycle is running (i.e., suspending CPU operation when there is no computation to be done);
- providing Application Programming Interfaces (APIs) for voltage and frequency scaling;
- providing standby and hibernation APIs (to reduce power consumption for longer periods of time);
- automatic idling of the inactive peripherals; and
- coordination of complex systems power management by means of tracing and notify mechanisms.

The major new release of DSP/BIOS RTOS is version 6.x. It provides the same core features as the DSP/BIOS 5.x, but augments its functionality, performance, and most importantly its portability. The main improved aspects are:

- the usage of a new configuration technology called Real-time Software Components (RTSC) to improve the portability of the code;
- APIs were changed so a compatibility layer is present to ensure that no changes are needed to DSP/BIOS 5.x software. This compatibility layer provides 100% source code compatibility for the great majority of DSP/BIOS applications.
- the number of priority levels both for SWI and TSK is increased from 16 to 32;
- IPC mechanisms were improved by adding *Events* objects and *GateMutex*. An Event object lets a task wait for one or more events, such a semaphore post or a mailbox post. The GateMutex provides a priority-inheritance mutex. It augments the functionality of a binary semaphore by avoiding the phenomenon of *priority inversion*[24]. To avoid this, a series of scheduling algorithms are available. The GateMutex implements one of them, called *priority-inheritance* algorithm. According to it, if a lower-priority task gains the access to a shared

[24]Priority inversion is a scheduling scenario in which an higher priority task is indirectly preempted by a lower-priority one. Given L the lower-priority task, M a medium-priority task, and H the higher-priority task. If L asks for the shared resource before the others, then if H wants to later use the same resource, it has to wait until L releases it. The problem is that if M does not need to use the resource and L gains access to the resource, then H is blocked while M is executed preempting L so that it cannot execute and release the resource. In this way the higher priority task is served as the lowest-priority task. This is a serious problem if H has a hard deadline.

resource and a higher-priority task is waiting for the same resource, than the lower-priority algorithm inherits the priority of the highest task waiting for that resource[25].

- analysis of task performance is enhanced since per-task CPU load analysis is available;
- enhanced debug features by means of user-configurable debug instrumentation options. Hook functions[26] are supported for HWI, SWI and TSK objects.
- improvements in memory management. A new heap manager is introduced: HeapMultiBuf. The main characteristic of this manager is that it can support a deterministic allocation of both variable and fixed length memory area. The memory system performance does not degrade and its performance is not affected by fragmentation.

2.6 DSP/BIOS™Link (DSPLink)

Texas Instruments DSP/BIOS Link (also known as DSPLink) is foundation software for the IPC between the GPP and the DSP [24, page 9]. Today many DSP-based applications use the GPP to control one or a set of DSPs. The most common operations between the two cores are:

- exchanging of control and data information;
- booting of the DSP by GPP; and
- control and coordination of algorithms and tasks running on the DSP by the GPP.

DSPLink is software designed to facilitate such interactions. It offers programmers a set of APIs that abstract the characteristics of the physical communication layer. The programmers' attention can focus on the development of the application algorithms, rather than on the communication mechanisms. To achieve this, a Remote Procedure Call (RPC) layer is introduced so that the characteristics of the physical layer connecting the two cores are hidden from the application. The appearance is that the GPP uses local function calls to obtain data from the DSP. The RPC layer is responsible for delivering messages between the two actors. DSPLink is capable of abstracting the underlying physical layers both in a System on Chip (SoC) device or in a device where the two cores are discrete and are not placed on the

[25] In our example, using a priority inheritance scheduling algorithm, the task M would be not able to preempt L, since L inherits the priority from the H task which has a higher priority than M.

[26] Hook functions are functions called before or after a particular events happens. They are suitable for software debugging and by means of them, a programmer can customize their debugging operations.

same board (e.g. a DSP connected through a PCI interface).

DSPLink is designed to work with DSP/BIOS OS (2.5.2) on the DSP side, while no specific operating system is required on the GPP side. Since the DSPLink package contains all the source code, it is possible to port it to a variety of operating systems (i.e., all of those supported by the GPP). The figure 2.6 shows the software architecture distributed over the cores. On both sides, *link drivers* hide the physical link layer. On the GPP side, the *processor manager* exposes the DSPLink APIs to the client, while the *OS adaptation layer* makes the DSPLink components independent from the specifics of the OS.

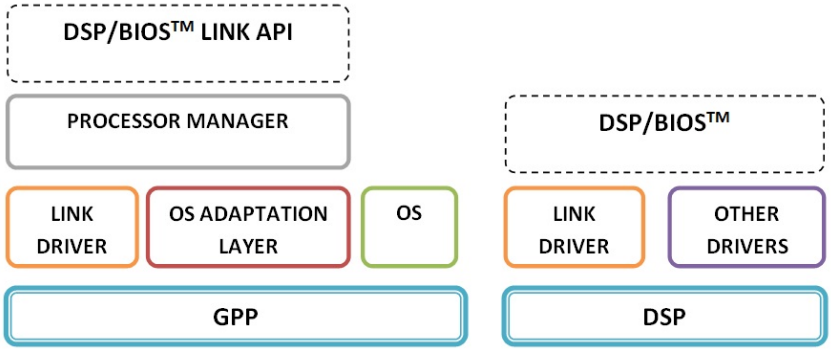

Figure 2.6. DSPLink software architecture (adapted from [24, page 13])

DSPLink offers the following **services** to the end-points clients[27][24, page 9]:

- basic processor control (via the PROC module);
- sharing and synchronization of a memory pool among the cores (POOL);
- notification of user events (NOTIFY);
- mutually exclusive access to shared resources (MPCS);
- linked list data streaming (MPLIST);
- data transfer over channels (CHNL);
- exchanging of messages (MSGQ);
- circular buffer data streaming (RINGIO) and
- zero-copy messaging.

[27] In brackets the name of the DSPLink module in charge of the service in question.

The list above represents the complete set of services offered by DSPLink. However, depending on the platform and operating systems used, the system may support only a subset of these services.

The last entry in the list refers to the **zero-copy transfer mode (ZCPY)**. Using this technique the data are exchanged among the cores by means of shared memory, avoiding any need to physically copy data to and from the respective memory spaces. In this way data transfer implies only the allocation of buffers in the shared memory region and addresses translation. The ZCPY system is composed of three sub-components [25]:

- *Shared memory allocator*: the data to be exchanged are stored in buffers allocated in the shared memory;
- *Address translator*: the buffers allocated in the shared memory need to be accessed by any core as user-space buffers. Since the DSP sub-system does not have a MMU, this component must perform address translations from the DSP physical address space to GPP virtual space and vice versa. Furthermore, the translation from GPP user and kernel spaces are performed by this component; and
- *Shared Memory Inter Processor Signaling*: this component is in charge of managing control structures and to inform the processors about changes in the shared buffers.

DSPLink provides several **advantages** for programmers working with an embedded multicore system:

- *Portability*: an application that uses DSPLink is easily portable to another architecture with no or few changes since architectural low level details are hidden;
- *Flexibility*: applications are more flexible since programmers can use the most appropriate high or low level communication protocols depending their necessities;
- *Scalability*: only needed modules need to be compiled and included in the executable file, saving resources; and
- *Physical layer abstraction*: the details of the physical link are hidden from the programmer, so that it is easier to focus on application development.

2.6.1 DSPLink components

Before starting talking about the modules and IPC mechanisms provided by DSPLink, the reader must be aware of the difference between the IPC mechanisms in the DSP/BIOS OS and in DSPLink. While the IPC techniques in the DSP/BIOS OS are intended for the communication among local threads, DSPLink makes possible

communication among threads located on different cores. Nevertheless, in some cases DSPLink bases its modules on the underlying DSP/BIOS OS ones (as in the MSGQ or POOL cases). The descriptions of the modules is based upon [26]. For more details (such as API descriptions) please refer to [24] and [26].

2.6.1.1 PROC

The PROC component (that stands for "processor"), represent the abstraction of the DSP core from the GPP's point of view. By means of this component, several operations can be performed:

- the DSP can be initialized by the GPP. This initialization is performed by means of the `PROC_setup()` API function that initializes the low-level components' structures. `PROC_destroy()` frees all these structures at the end of the cooperation;
- the GPP gets a local reference to the DSP (using `PROC_attach()`). The first GPP attached to the DSP becomes the "owner" of the DSP so that race conditions can not happen. The GPP calls `PROC_detach()` at the end of the application;
- the executable file is loaded in the DSP (using `PROC_load()`). By convention, DSP executable files have the `.out` extension;
- the execution of the executable file is started/stopped on the DSP (using `PROC_start() and PROC_stop()` respectively);
- direct reads and writes to the DSP memory can be done through `PROC_read() and PROC_write()`.

In the current DSPLink version, only one DSP is supported by the GPP. However, the APIs were made generic to enable future improvements in a multi-core system architecture.

2.6.1.2 POOL

The POOL module is the shared memory manager. It configures and allocates buffers in the shared memory region. These buffers can be used by IPC mechanisms to enable cores to communicate via shared memory. The functions of this component are:

- configure the shared memory region by means of `POOL_open()`. After all operations are concluded, the memory is freed by `POOL_close()`;
- allocate/free buffers via `POOL_allocate() / POOL_free()`;
- translate address of the allocated buffer into different address spaces (POOL_translateAddr());

- synchronize the memory content from the different processors' point of views.

The DSPLink offers two different kinds of memory pools. The first one is SMAPOOL where zero-copy buffers are allocated. The second one is BUFPOOL for the allocation of fized-sized buffers.

2.6.1.3 NOTIFY

The NOTIFY component allows clients to register and be notified of events happening on the other processor(s). A client can either wait for or generate an event. The `NOTIFY_register()` function is used to register for a particular event which could be generated locally or at a remote location. A callback function is also defined so that this function will be executed when the respective event is received. An event is unique since it is identified by an Inter Process Signal (IPS) event number in a defined IPS table. To unregister (in order to stop listening on the given IPS) `NOTIFY_unregister()` should be invoked. When a processor wants to signal the completion of an operation, it sends a notification through `NOTIFY_notify()`. The notification is broadcasted to all listeners. A custom 32-bits payload can be attached to the notification. The NOTIFY component also provides different priorities for the notifications. A lower priority number means higher priority. The notification technique is feasible only for infrequent notification. The reason for this is that a new notification can not be sent before the previous one has been read by the listener. This introduces some non-deterministic delays, which can be very dangerous in real-time systems.

2.6.1.4 MPCS

The Multi Processors Critical Section (MPCS) component provides mutually exclusive access to a shared memory region. The MPCS component enables different clients on different processors to access the same shared object while avoiding conflicts and data coherency issues. The functionalities provided by MPCS are:

- creation of a MPCS object with a system-level unique name (`MPCS_create()`);
- the handle to the MPCS object is retrieved by calling `MPCS_open()`;
- exclusive access to the shared region is gained by calling `MPCS_enter()`. After having completed the operation, the shared region must be left as soon as possible using `MPCS_leave()` to maximize system efficiency; and
- the MPCS object is closed and deleted when no longer needed (`MPCS_close()` and `MPCS_delete()`).

Unfortunately, the MPCS mechanism is not robust since it does not tolerate *deadlocks* and *priority inversions*. The former issue happens when multiple tasks are running on the GPP and DSP and they involve both an MPCS object and another

DSPLink IPC tool that internally uses a MPCS object. Priority inversions can occur due to thread priorities and can occur if the DSP side uses the SWI module (a high-priority thread). If a thread on the GPP is pre-empted after having gotten the MPCS lock, then the DSP side must wait until the thread running on the GPP completes. This means that the SWI task can not execute. This is really a serious issue for a real-time operating system, but fortunately the priority-inversion issue has been solved with DSP/BIOS 6.x (see section 2.5.2)

2.6.1.5 MSGQ

The MSGQ (message queue) module is in charge of exchanging of messages among processors. The message exchange paradigm is queue-based. This means that the reader waits for messages in a queue (`MSGQ_get()`) while the writer puts the message on a queue (`MSGQ_put()`). The queue can be located locally or remotely. For this reason, a MSGQ transport protocol is needed (`MSGQ_transportOpen()`) both to locate (`MSGQ_locate()`) and to handle a remote queue and to send messages over the link. A message queue can have only one reader. In contrast, many writers can send messages to the same queue at the same time. The MSGQ module is usually used when more than 32-bits of information must be sent (as in this case the NOTIFY component not suitable). The information can be of variable length since variable length payloads are supported. Moreover, if the application needs to send messages frequently, this module avoids the problem related to the NOTIFY component; thanks to the presence of a queue that stores messages in a First-In First-Out (FIFO) order.

This component utilizes *synchronous* communication, thus the `MSGQ_get()` and `MSGQ_put()` are blocking if the queue is empty or full, respectively.

In a multi-DSPs system, communication by means of message queues among multiple DSPs is possible even without DSPLink. In fact, it is sufficient because the MSGQ module of DSP/BIOS OS provides *ad-hoc* transport protocols.

2.6.1.6 MPLIST

The MPLIST is an extension of the MSGQ module and it can be used in applications that require multiple readers and multiple writers. It implements a doubly-linked list of messages within the shared memory region thus out-of-order information processing is feasible. The basic approach is still a message-like structure, but the structure is no longer a simple queue. The list is created using `MPLIST_create()`, by specifying a system-level unique name. It is opened using `MPLIST_open()` which returns a handle. A client may manipulate the elements in the list in one of the following ways:

- put the new element at the end of the linked list (with `MPLIST_putTail()`);
- read an element from the head of the list (`MPLIST_getHead()`);
- check if the list is empty (`MPLIST_isEmpty()`);

- insert an element in a given position specifying the address of the element that will follow the new one (`MPLIST_insertBefore()`);
- remove a given element from the list (`MPLIST_remove()`);
- get a pointer to the first element in the list (`MPLIST_first()`); and
- get a pointer to the element that follows the one specified as argument (`MPLIST_next()`).

By using the functions that are provided, out-of-order information processing can be performed. Additionally, information can have different priorities assigned depending on their position in the list.

2.6.1.7 CHNL

The CHNL component realizes a unidirectional logical data transfer channel between a single reader and a single writer using the same buffer size. Over a single physical link, multiple channels can be allocated and used at the same time. The end points of the data channel are defined during the creation of the channel (`CHNL_create()`) so that no information regarding the source and destination is stored in the data buffers flowing through the channel. This component utilizes the *issue-reclaim* model. According to this model, the client only needs to issue (`CHNL_issue()`) or reclaim (`CHNL_reclaim()`) a buffer (empty to get, full to send information) and is totally unaware of the channel state. Underlying mechanisms exchange buffers only when they are available to both end-points. Multiple buffers can be queued on the same channel to improve the performance.

2.6.1.8 RINGIO

The RINGIO component implements a circular buffer for use by one reader and one writer. A circular buffer is allocated in the shared memory (`RingIO_create()`) and is identified by a system-level unique name. RINGIO is the only IPC mechanism that enables the two actors to work with different sized buffers. Consequently the reader and writer are independent. The writer acquires empty memory space in the circular buffer (using `RingIO_acquire()`). Only when the data buffer is filled, does the writer release the buffer (`RingIO_release()`) and the chuck of data is committed to the shared circular buffer. On the other side, the reader acquires a region of the buffer with valid data and invalidates the buffer after having released it. A part of the shared memory is used for the control attributes related to the data. The RINGIO module provides great flexibility in terms of:

buffer size as an actor can acquire a certain amount of data and release a different amount; and

operations on data as the buffer can be totally flushed and unused data that was acquired but not released can be cancelled.

Due to its flexibility, RINGIO perfectly suits the processing of streaming media (or in our case a stream of digitized radio signals).

2.7 Previous work

Among the studies related to SDRs, two of them are very related to this master thesis project. The first one is Philip Balister's master thesis work [5]. The objective of this study is on one hand the feasibility of the porting of a SCA waveform developed through OSSIE to an embedded platform (specifically the OMAP 5912 platform[28]) and on the other hand to estimate the resources used by the waveform in terms of memory and processor usage. Performance results of single components of the waveform are presented. The results shown by Balister's study are interesting to port SDRs to embedded systems. By utilizing the methods and tools explained in the thesis work [5], is possible to detect the modules of the SDR system that requires an excessive amount of system resources. After that, this modules can be optimized and executed on different processors according to the optimization and execution techniques explained in this master thesis report.

The second study, by Tore Ulversøy and Jon Olavsson Neset, focuses on the analysis of a SCA-based SDR-application workload as a function of the granularity of the system [27]. SCA allows SDR applications to be made up of different distributed components. This introduces a processor workload overhead due to the communication between these components. This overhead is estimated as function of data packet size exchanged by the components by varying the workload of the single components. During their measurements, several components were deployed on the same CORBA capable GPP on a desktop computer. Results of this analysis show that the processor workload increases as the number of components and the data packet size increase. This means that a trade off is needed between the scalability of a distributed SDR system and the processor workload. For further details, see [27]. Even though their experiments were not executed on an embedded system but on a desktop computer, the results concerning the processor workload can be integrated with the analysis done in section 4.4.1 where the execution time of a chain of blocks is presented.

[28]The Texas Instruments' OMAP 5912 platform is made up of an ARM9 GPP and a Texas Instruments' TMS320C55x DSP

Chapter 3

Method

This chapter explains in detail the method used for the analysis of an OMAP3530 for the porting of an embedded SDR system. Different deployment solutions are proposed, implemented, and analyzed. This chapter focuses on the implementation part. Results and analysis are presented in the chapter 4.

The section 3.2 describes the software and hardware environments used during the development of the software and the analysis of experimental results. The testing software is described in detail in the section 3.3, while the two solutions proposed and used for the actual testing are explained in sections 3.4 and 3.5 respectively. A thorough explanation of the tools used in order to measure the performance of the system is given in section 3.6. Finally, section 3.7 explains how these two solutions handle floating point operations.

3.1 Introduction

As already stated in section 1.2, the aim of this masters thesis project is to analyze the performance of the OMAP3530 platform in order to port a SDR system to this platform. The OMAP3530 platform is made up of an ARM Cortex-A8 GPP and a TI C64x+ DSP (see section 2.4). Initially the focus was on the analysis of the system performance when a GPP + DSP configuration is exploited. In order to quantify the system's performance and to be able to perform an empirical analysis, the testing software explained in detail in section 3.3 was developed. The function of this software is to filter a noisy sound file in order to reduce the noise.

Afterwards my attention shifted to the NEON SIMD coprocessor in the Cortex-A8 GPP. Hence the testing software was modified in order to exploit this coprocessor instead of the DSP. This led to a comparison of the performance of these two different configurations.

Finally, the test software was modified to support floating point operations in order to be able to perform a comparison between the two solutions when executing

floating point operations.

In this report, sometimes the terms GPP and DSP are used to indice the processes which executes on the specific processor.

3.2 Development and experiments environments

In chapter 2 a general introduction to the hardware system was given. This section will focus on the specifics of hardware and software tools that were utilized. The empirical analysis was performed on a BeagleBoard (see section 2.3). The core of the board is an OMAP3530 processor made up of a an ARM Cortex-A8 GPP and a TI C64x+ DSP (see section 2.4). The specific versions of the system components are listed in table 3.1.

Table 3.1. Revisions of components of the experiments environment

System	Revision
BeagleBoard	C3
OMAP3530	ES 3.0
ARM Cortex-A8	r1p3

An important parameter that deserves to be emphasized is the clock frequency since most of our measurements are in terms of timing. The OMAP3530 ES3.0 platform can have a maximum clock rate of 600 MHz for the GPP and 430 MHz for the DSP. Nevertheless, the system working at these conditions is considered to be overdriven resulting into a shorter life expectancy of the system[1] and into a lower reliability of the system (since correct results may not be guaranteed anymore). The default clocking for our system, the best compromise between performance and longevity, is 500 MHz for the GPP and 360 MHz for the DSP. To learn the current clocking rate of the system, the code in listing 3.1 was executing on the GPP.

Listing 3.1. Clock rate code

```
1 while(1) {
2   ccnt_value = ccnt_read(); /* read the current cycle counter */
3   printf("Current CCNT value is: %u\n", ccnt_value);
4   usleep(1000000);          /* sleep 1 second */
5 }
```

Details about the function `cccnt_read()` will be given in section 3.6.1. The output of the code shows the value of the cycle count register (a 32-bit register) every second:

```
Current CCNT value is: 511665946
Current CCNT value is: 1011851249
```

[1]The life span of the system is reduced since an increase in processor's frequency causes heat to increase linearly. If the heat is not properly dissipated, the system becomes overheated. This can cause degradation effects (e.g. electromigration).

```
Current CCNT value is: 1512065528
Current CCNT value is: 2012229861
Current CCNT value is: 2512381502
Current CCNT value is: 3012555421
Current CCNT value is: 3512754317
Current CCNT value is: 4012945691
Current CCNT value is: 218182281
Current CCNT value is: 718308923
```

From the output values, we can compute the median value of the number of clock ticks in a second that is 500185303[2]. We can figure out that the clocking of the system is around 500 MHz I have decided to perform the experiments on a system working in the normal conditions according to the specification. Nevertheless, to increase the performance of the system, overclocking can be enabled by typing u-boot commands booting the operating system. Specifically, to set GPP clock to 600 MHz the command is `mw 48004940 0012580c`, while to set the DSP clock to 430 MHz, the command is `mw 48004040 0x0009ae0c`. These commands are intended for working on a system whose component versions are listed as in table 3.1.

In order to use the platform, an operating system must be executed both on the GPP and DSP side, respectively the Ångström Linux distribution (section 2.5.1) and the DSP/BIOS 5.x (section 2.5.2). The motivations for these choices are explained here. Although several operating systems have been ported to the BeagleBoard (e.g. Symbian OS, Android, Ubuntu, Micorsoft's Windows Mobile, etc.), the **Ångström Linux** distribution was chosen because it is an open source Linux system, stable and user friendly that offers everything necessary for the purposes of this thesis work. Additionally, a lot of documentation is available and a large amount of is given by the open source community. The **Ångström** OS was built from source code following instructions at [28] and then loaded on a dual partitioned SD-card[3]. The Linux kernel version of the system was 2.6.32. Since the Ångström distribution is based on the OpenEmbedded project, it inherits the package building system based on `bitbake`, a tool that supports cross compiling of packages. The build process provides a cross compiler toolchain. The toolchain consists of tools from the GNU Compiler Collection (GCC)[4] toolchain version 4.3.3 (the names of the tools are the same as the GCC toolchain with the prefix `arm-angstrom-linux-gnueabi-`), modified for targeting the ARM processors. During my project, the specific tools used were:

arm-angstrom-linux-gnueabi-gcc C compiler;

arm-angstrom-linux-gnueabi-as ARM assembly compiler;

[2]The median value was taken into consideration instead of the average so that the incoherent difference between the ninth and the eighth values (due to the register overflow) does not affect our result.

[3]The SD-card must have two partitions. The first one is a boot partition using the FAT file system to be read by the OMAP bootloader. The second partition is intended for the Linux root file system and uses an ext3 file system

[4]http://gcc.gnu.org/

arm-angstrom-linux-gnueabi-gdb application debugger; and

arm-angstrom-linux-gnueabi-objdump displays information about object files. In this project the disassembler was used to generate assembly code based upon executable files by means of the option `-d`.

In addition, this toolchain is compliant with the GNU Embedded Application Binary Interface (EABI). An ABI is a low-level description of the interface between the applications and the operating system hosting them. EABI extends a common ABI by improving [29]:

- the performance of floating point operations. The code of both software and hardware floating point operation can be mixed and the general performance is improved (for more details see [30]); and
- the conversion of several system calls.

On the DSP side, the BIOS used is the **DSP/BIOS 5.41.07.24**. The version of DSPLink used is the 1.65.00.03. The reason why a DSP/BIOS 5.x version was chosen rather than a DSP/BIOS 6.x one, is that currently DSPLink supports only DSP/BIOS 5.x on the DSP side when executing on an OMAP3530 platform. This can be verified during the DSPLink installation phase as specified in [31]. One of the first steps in the building and installation of the DSPLink, is the configuration and the building of the kernel object to be loaded at boot time in the target operating system. This is done by calling the perl script `dsplinkcgf.pl` in the following way:

```
perl dsplinkcfg.pl --platform=OMAP3530 --nodsp=1 --dspcfg_0=OMAP3530SHMEM
--dspos_0=DSPBIOS5XX --gppos=OMAPLSP  --comps=ponslrmc
```

As we can see, one of the arguments passed to the script is: `-dspos_0=DSPBIOS5XX`. If the user tries to set up the DSP/BIOS 6.x by typing the option `-dspos_0=DSPBIOS6XX`, the output from the script is:

```
****************** ERROR !!! ***************************
Please provide a valid DSP OS!
Following DSP OS are supported by OMAP3530 with Shared Memory Physical
Interface:
<ID>-->DSPBIOS5XX
DSP/BIOS (TM) Version 5.XX
```

Furthermore, the current version of DSPLink supports only one DSP and the only communication method for the OMAP3530 platform is shared memory. However, this limitation does not affect our analysis. In section 2.5.2 it was stated that DSP/BIOS 6.x offers a larger number of IPC mechanisms. Nevertheless, the IPC mechanisms provided by the operating system are those provided locally among the thread running locally on the DSP. The IPC mechanisms in which we are interested in this project, are those provided by DSPLink as these make possible the communication between threads on *different* processors (theoretically on a high number[5] of processors). In order to develop DSPLink applications, a Linux

[5]This number is theoretically limited to $2^x - 1$, where x is the number of address bits.

workstation was set up. The Linux workstation was running Fedora Release 12 (Constantine) operating system, based on Linux kernel 2.6.31. Besides the DSPLink and DSP/BIOS, other tools are needed. Below these tools along with the version number used for this project are listed:

Code Generation Tool 6.1.17 TI's collection of tools needed to build DSP applications. It contains an optimizing C compiler and assembly language tools (assembler, linker, archiver, hex conversion utilities, etc.);

Local Power Manager 1.23.01 used to create kernel modules for power management system services;

eXpress DSP Components (xdctools) 3.20.03.63 collection of components to create, test, deploy, and install RTSC components. RTSC is used to develop modules and components constituting distributed embedded applications.

TMS320C64x+ DSP Library v. 2.1.0 collection of optimized functions for signal processing on the C64x+ DSP; and

IQMath Library v.2.1.3 collection of libraries for supporting floating point operations on the DSP. This library is provided by TI for a Windows-based development environment. However, the code can be cross-compiled to target the operating system running on the GPP (Linux in our case)..

For some experiments and for debugging purposes, the Code Composer Studio tool (see appendix A), running on a Windows XP workstation, was used.. All tools needed (with the exception of DSPLink), are embedded in the Code Composer Studio Tool. The version of this tool used is CCS v4.2. After having built the application on the host machine, the executable files need to be loaded into the file system of the target machine and launched from a shell. Two methods were used to achieve this: downloading the code via the serial interface or via a network interface (via a *Secure Shell (SSH)* connection). In the first case, the *RS232* BeagleBoard port is connected to the host's serial port and the transfer is done. In the this case you need a RS232 flat ribbon cable to DB9 serial connector to connect the BeagleBoard to the serial port of the host machine. A remote serial console to the BeagleBoard is set up on the host computer using Minicom. Minicom is a text-based modem application that emulates a remote unix-like console session through a serial connection. The second method was to use a network SSH connection from the host to the target platform. Since the BeagleBoard does not provide an Ethernet interface, a D-Link DUB E-100 Ethernet - USB 2.0 adapter was used. This second method makes it possible to access the target machine via the network.

The table 3.2 summarizes hardware and software specifics of the GPP and DSP.

Table 3.2. SW and HW specifics of GPP and DSP (D.M.=Direct Mapped, S.A.=Set Associative)

	GPP	DSP
Clock frequency	500 MHz	360 MHz
Cache specifics	16 KB L1i 4-Way S.A. 16 KB L1d 4-Way S.A. 256 KB L2	32 KB L1i D.M. 80 KB L1d 2-Way S.A. 64 KB 4-Way S.A.
Operating system	Ångström Linux kernel 2.6.32	DSP/BIOS 5.41.07.24
Compiler	Ångström toolchain GCC 4.3.3	CGT 6.1.17

3.3 Software for performance testing

In order to test the performance of the system, test software was developed. This software should process signals and have similar functionality and structure to the code that would be used to implement an SDR. The aim of this software was to remove noise from a Waveform Audio File Format (WAVE) audio file. In the case of SDRs, the data source commonly is the USRP which provides data to the host machine via its USB interface. For our testing purposes an audio file simulates such stream of samples. The application provides the possibility of changing the input sample rate, the number of bits per sample, and the sample data type. The core of this software is a Finite Impulse Response (FIR) filter. As the audio signal is processed by the filter, the stream of output data is stored in an output WAVE file. In this way the correct functioning of the system can be easily tested by listening to the output file. Figure 3.1 is a block diagram of the test software. The input and output blocks are the same for every version of the test software. The FIR block varies noticeably depending on the configuration of the system being tested.

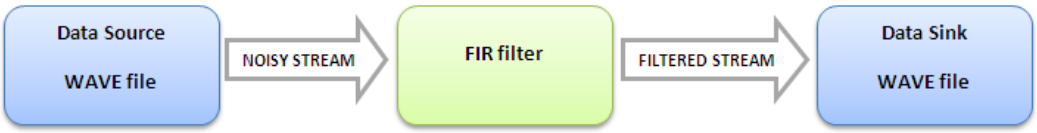

Figure 3.1. Testing software block diagram

3.3.1 The input WAVE file

The input WAVE file was chosen from a library of WAVE files freely available on the Internet. This file contains one channel of 44100 Hz data as signed 16-bit data samples. The size of the file is 822.6 KB and it contains 9.55 seconds of sound data. Such a file was chosen since it was a good trade-off between size and duration of sound data for our testing purposes. The WAVE format was selected since it is widely supported by many operating systems and by GNU Radio. The specifications for

this format are available at [32, page 55]. GNU Radio (see section 2.1.1) was used to insert noise into the input audio signal at a predefined frequency by using the block diagram shown in figure 3.2. When executing the block diagram in question, a sine wave of 10 kHz was added to the input WAVE signal, resulting in a whistle in the output signal. A noise of 10 kHz was selected since the main frequencies of the input file were in the range 0 - 5 kHz. This can be seen in figure 3.3 where both the input and the noisy signal are displayed. A low-pass Hamming FIR filter is applied to the noisy signal. The cut off frequency of this filter was set at 3 kHz with a transition width of 1 kHz.

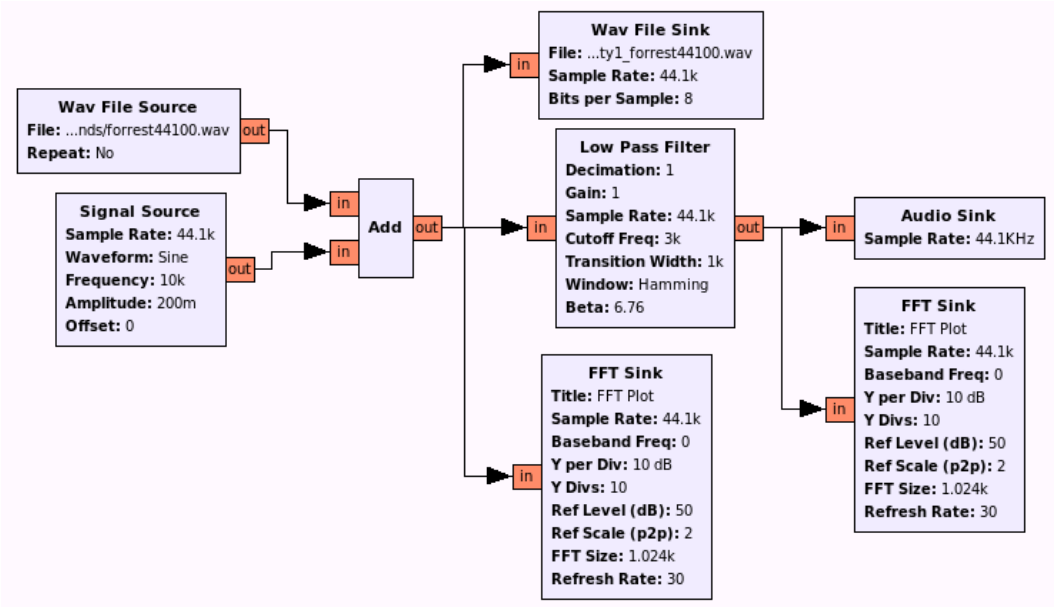

Figure 3.2. GNU Radio block diagram of the network used to insert noise and to display the FFT of the signal before and after processing with a low pass filter.

The frequency spectrum of the noisy signal, as generated by GNU Radio, is shown in figure 3.3. From the spectrum we can clearly see the frequency of the noise at 10 kHz and the frequencies of the input audio file, with the later at frequencies below 5 kHz.

In order to optimize transfer of the data to and from the GPP, a memory-mapped file was used. A memory-mapped file is not physically copied into the main memory. Instead, a segment of the virtual memory is set up to map the contents of the file stored on the disk into the address space of the processor. Each access to the file, is treated as it was in main memory and access happens via the memory-mapped file input/output routines. The code used to map the input WAVE file is showed in listing 3.2. In this code, the function `Print_0` is invoked. This function (as well as `Print_1`), is widely used in the rest of the report. This function is just a wrapper of the `printf()` with the addition of the flushing of the buffer. We note the use in this code of data type `Char16`. This type is equivalent to the `short` type

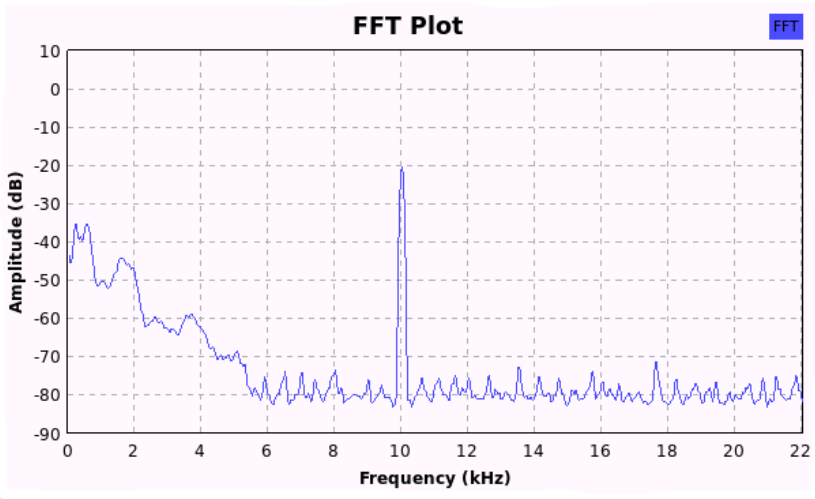

Figure 3.3. Frequency spectrum of the signal with 10 kHz noise added to it

and is defined by DSPLink in the header file `gpptype.h`. After the mmap call the contents of the file can be accessed via the "content" pointer, just as if the data were stored in an array in memory. This access is rapid and avoids the need to make repeated calls to the read() or fread() functions. If the operating system has sufficient memory available the contents of the file can remain buffered in memory once the file has been accessed once.

Listing 3.2. Memory-mapped input WAVE file

```
1  int fp_in;
2  Char16 *content;
3  int pagesize
4
5  ...
6
7  fp_in = open(inputFile, O_RDONLY);
8  if(fp_in == -1 )
9          Print_0("Error while opening input file\n");
10 pagesize = getpagesize();
11
12 ...
13
14 content= mmap((caddr_t) 0, header->dataLen, PROT_READ, MAP_SHARED, fileno(fp_in)↩
       , 0);
15 if(content == MAP_FAILED) {
16   printf("Error in mmap\n");
17   exit(1);
18 }
```

By using this technique, the I/O performance is increased by limiting the access in the file system since no system calls (`fread` and `fwrite`) are used, which are slow and blocking. Furthermore, no data must be copied from the kernel space to the user space. Linux implements *lazy loading* so portions of the file will be

transferred into main memory as needed. With read ahead, this can enable the file to be brought into memory before it needs to be accessed; while avoiding the need for the user to manage the actual file operations. The loaded portion is a multiple of a *page size*.

The performance of the reading and writing of the data file were evaluated using a simple test program consisting of the code in listing 3.3. Using this code a simply increment each 16 bit value and writing of the result into the output file revealed that the throughput was 24.95 Mbytes per second. By using system calls `fread` and `fwrite`, the resulting throughput was 13.90 Mbytes per second.

Listing 3.3. mmap test program

```
int fp_in;
int fp_out;
Char16 *data;
Char16 *dataOut;
int j;

...

inputFile = argv [1] ;
outputFile = argv [2] ;

ccnt_enable();
ccnt_start();

fp_in = open(inputFile, O_RDWR);
fp_out = open(outputFile, O_RDWR);

...

timer.startTime=ccnt_read();

data= (Char16*) mmap((caddr_t) 0, fileLen, PROT_READ, MAP_SHARED, fp_in, 0);
if(data == MAP_FAILED) {
  printf("Error in mmap in\n");
  exit(1);
}

dataOut= (Char16*) mmap((caddr_t) 0,fileLen, PROT_WRITE,  MAP_SHARED, fp_out, 0)↩
    ;
if(dataOut == MAP_FAILED) {
  printf("Error in mmap out\n");
  exit(1);
}

for (j=0; j< (fileLen >> 1); j++)
  dataOut[j]=data[j]+1;

munmap (data, fileLen);
munmap (dataOut, fileLen);

timer.endTime=ccnt_read();

...
```

3.3.2 The FIR filter

In electronics, a *filter* is "a system that removes components of the input signal based on frequency" [33, page 272]. In our application, the FIR filter block receives as input the noisy audio signal and produces as output the filtered audio stream. Since the noise frequency is 10 kHz and the significant part of the audio stream has frequencies lower than 5 kHz, the type of filter needed is a *low-pass* filter. Low-pass filters "pass low frequency and block or strongly attenuate high frequencies" [33]. An FIR filter is a digital filter whose output signal y is defined by the convolution:

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] \Rightarrow y[n] = b_0 x[n] + b_1 x[n-1] + ... + b_N x[n-N]$$

where x is the input signal, b is the array of *filter coefficients* or *taps*, and N is the filter order. The number of taps (length of b array) is $N + 1$. In such an FIR filter, the input signal is convolved with a filtering function. In my application this function is the *Hamming Window* defined as:

$$w(n) = 0.54 - 0.46cos(\frac{2\pi n}{N-1})$$

The coefficient array b was computed by means of the *Filter Design & Analysis Tool* integrated in MATLAB. The designed filter was a direct-form Hamming window low-pass FIR filter. The sample frequency parameter was set to 44100 Hz, the cut off frequency at 5 kHz with default attenuation at cut off frequencies of 6 dB. By means of this tool, filters with different orders N were designed and the respective coefficient array was exported as signed 16-bits integers and stored as array in C language header files to be included in our application. As figure 3.4 shows, in the upper part of the figure the magnitude response function of a filter of order 511 (512 taps), while in the lower part of the figure shows the filtered signal in the frequency domain. From this figure we can see that the noise is reduced by 80 dB. Frequencies of the input audio signal are lower than 5 kHz. For our scope, a low-pass FIR filter was sufficient to filter all frequencies above 5 kHz, by passing all frequencies of the input audio signal unaltered. Nevertheless, since the noise frequency is known (at 10 kHz), another possibility is to use a *notch filter*. A notch filter is a band-stop filter with a narrow stop-band. This filter passes most frequencies unaltered while reducing those frequencies in a specific range.

3.4 GPP + DSP Solution

The first objective was the evaluation of the platform while running the test software both on the GPP load, and the DSP. Specifically, we are interested in evaluating the performance of the software in terms of execution timing, GPP and DSP load as a function of the IPC mechanism used to communicate via DSPLink. In order to achieve this, the test software introduced in section 3.3 was modified to produce

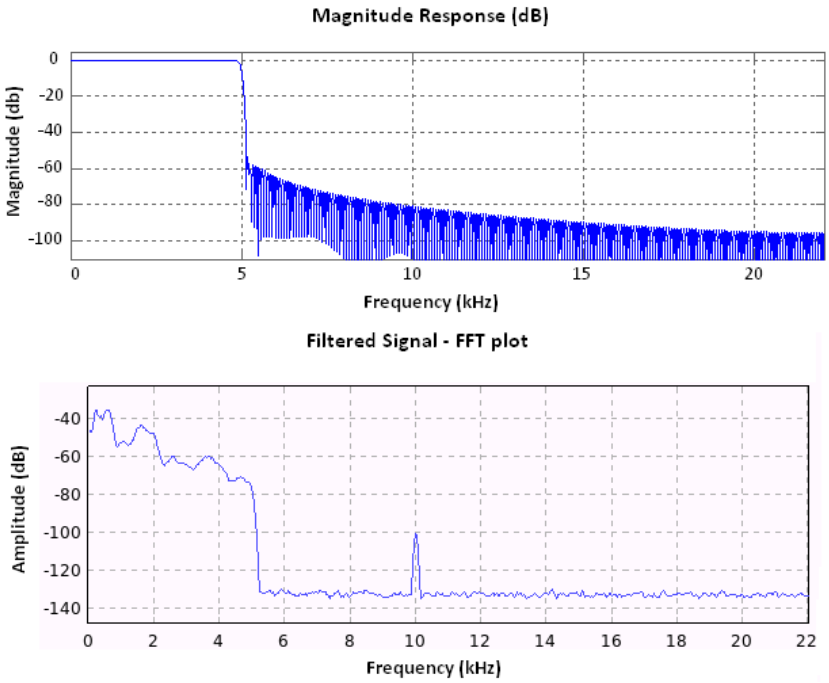

Figure 3.4. Magnitude response of a 511-order FIR filter and frequency spectrum of filtered signal

six different versions of the software (each using a different IPC protocol). Figure 3.5 shows how the blocks of the test software were distributed over the system's processors.

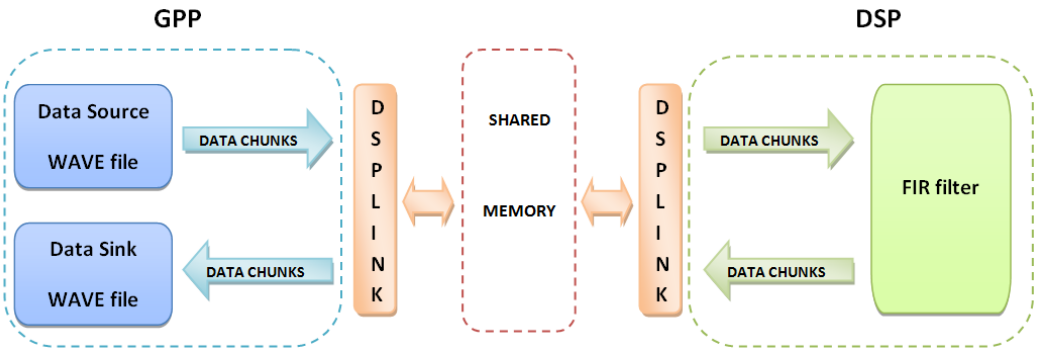

Figure 3.5. Test software block diagram: GPP + DSP

3.4.1 General description

The two processors communicate via a shared memory. The DSPLink software provides a wrapper for access to this memory and offers several IPC mechanisms to exchange data. Because of the limited amount of shared memory (the default size of shared memory in the OMAP3530 is 1 MB), the stream must be split in data chunks by the GPP, sent to the DSP for the filtering, received, and stored into an output file. The length of the chunks and the number of filter's taps are variable so that our analysis has varied both the length of the chunks and the number of the filter's taps (see chapter 4).

The DSPLink shared memory, according to the DSPLink design, is **cacheable** on the DSP side and non-cacheable on the GPP side. In fact, the shared memory regions are reserved in non-cached memory on the GPP side and no cache coherency protocol is implemented on the GPP . On the other hand, on the DSP side the shared memory is cached by default. There is the possibility to disable the cache by configuring the DSP/BIOS. On the DSP side the cache coherency is ensured for all modules with the exception of the PROC and MPLIST modules where the application must provide the data coherency.

3.4.1.1 GPP side

In all the six versions of the software, the main function on the GPP side is `GPP_Main()`. This function sequentially calls the other functions that will realize the GPP side functionality. In this function, the length of the data chunks (passed as a command line argument) is aligned to the length of the L1 cache word in the DSP (128 bytes in our case). In our analysis, the length of the chunks utilized will be multiples of 128 bytes. For software using the PROC, MPCS, CHNL, and MSGQ modules, the **`GPP_Main()`** sequentially calls `GPP_Create()`, `GPP_Execute()`, and `GPP_Delete()`. The function **`GPP_Create()`** creates the PROC object with `PROC_setup()`. Then the GPP is attached to the DSP with `PROC_attach()`. Then, one or more memory pools are allocated (via `POOL_alloc()`) after having set up the number and the size of the buffers to be allocated in the pool(s). Depending on the IPC module to be used, some address range in the GPP memory space is translated into the DSP memory space. The next step is to load the DSP with its executable file and some program specific arguments. This is done through the function `PROC_load()`. In this function the data structures needed by the IPC module in question are created and set up. The last step is to start the execution of the DSP by invoking `PROC_start()`. The function **`GPP_Execute()`** reads the stream of data from the GPP's memory and write it into the DSPLink memory space, splits the data into chunks, and sends the chunks in a loop via the chosen IPC protocol provided by DSPLink. The function returns a pointer to memory where the filtered data has been saved. The last step is **`GPP_Delete()`** which deallocates all the data and memory structures utilized by the previous functions. A common procedure is to stop the execution of the DSP with `PROC_stop()`, close the buffer

pool (`POOL_close()`), detach the GPP from the DSP with `PROC_detach()`, and finally to destroy the PROC object (`PROC_destroy()`).

In contrast, for MPLIST and RINGIO modules, the `GPP_Main()` first calls `GPP_Create()` and then `GPP_create_thread()`, a wrapper of the function `pthread_create()`[6], to create two parallel threads. One thread is the *writer*, which is in charge of sending data to the DSP; the other is the *reader* that receives the filtered data back. The function `GPP_Join_client()` waits for the threads to finish their execution and finally the function `GPP_Delete()` is invoked. This modification was made since these two modules are better suited for parallel data processing (see the specific software description paragraphs for further explanations).

The DSP boot mode chosen for the software in this project is the *normal boot mode* shown in figure 3.6.

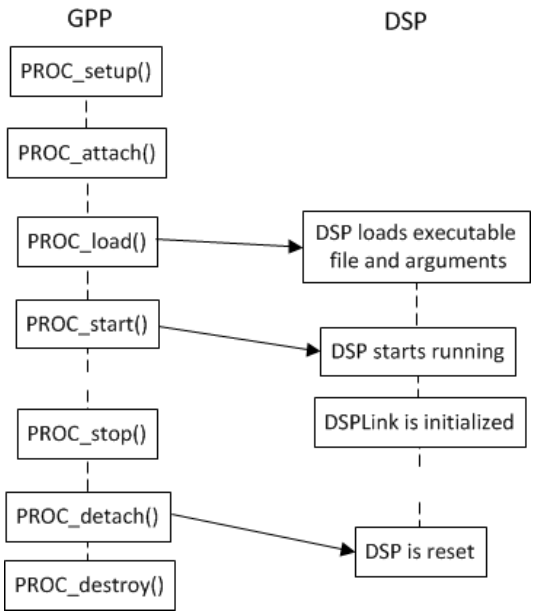

Figure 3.6. Normal boot mode (adapted from [24, page 21])

3.4.1.2 DSP side

On the DSP side, tasks (TSK module) are used instead of software interrupts (SWI). The reason for that is that tasks must be used when communicating via IPC protocols since tasks can be suspended. In contrast SWIs cannot yield the processor. Furthermore, the main advantage of software interrupts over tasks is that the context switching time is reduced. However, in our case, only two threads are running on the DSP: the main task and `tskDSP`. Since the main task is suspended

[6]From the Posix pthread library

after creating the tskDSP task, the benefits of using software interrupts would be limited.

The execution flow of the DSP starts when the function PROC_start() is invoked on the GPP side. The first operation to be done by the DSP is to receive and save the arguments sent by the GPP via the function PROC_load(). Also on the DSP side the chunk length is aligned with the cache word length. Then, the main task creates the task tskDSP with a priority level of 4. The flow of this task is very similar to the flow of the GPP program. The first function invoked is **TSKDSP_create()** that allocates the necessary areas of the DSP memory space and sets and initializes some transfer information. One aspect that is important to note is that all DSPLink structures are created, allocated, and initialized on the GPP side. The next step is to call the function **TSKDSP_execute()**. This function represents the main functional part of the DSP code since it receives data, filters the samples, and sends the resulting data back to the GPP. The next function to be invoked is **TSKDSP_delete()**, to deallocate all those structures created in the TSKDSP_create() function.

The code for the FIR filter is placed in the TSKDSP_execution() function. The code was optimized using functions from the DSPLIB and it is commont to all versions of the software. The code is showen in listing 3.4.

Listing 3.4. Code of FIR filter on the DSP side

```
1    ...
2   void DSP_fir_gen (
3     const short *restrict x,     /* Input array [nr+nh-1 elements] */
4     const short *restrict h,     /* Coeff array [nh elements]      */
5     short       *restrict r,     /* Output array [nr elements]     */
6     int nh,                      /* Number of coefficients         */
7     int nr                       /* Number of output samples       */
8   );
9
10  void DSP_blk_move_gen (
11  short * restrict x,      /* Source address */
12  short * restrict r,      /* Destination address */
13  int nx             /* Number of bytes to be transferred */
14  );
15
16  ...
17
18  #pragma DATA_ALIGN (filter_out, 4)
19    filter_out = MEM_calloc (DSPLINK_SEGID, numSamples * sizeof (short), 4) ;
20    history = MEM_calloc (DSPLINK_SEGID, (FILTER_ORDER) * sizeof (short), ↩
          DSPLINK_BUF_ALIGN) ;
21    filter_in = MEM_calloc (DSPLINK_SEGID, (numSamples + FILTER_ORDER) * sizeof ↩
          (short), DSPLINK_BUF_ALIGN) ;
22    first_filter_in = MEM_calloc (DSPLINK_SEGID, numSamples * sizeof (short), ↩
          DSPLINK_BUF_ALIGN) ;
23
24    for(i=0; i< numberOfChunks; i++) {
25      /* Do processing on this buffer */
26    if (status == SYS_OK) {
27        if (first_chunk == 0) {
28            DSP_blk_move_gen (history, filter_in, FILTER_ORDER);
29            memcpy(&filter_in[FILTER_ORDER], buffer_input, bufferSize);
```

```
30            DSP_fir_gen(filter_in, coeff, filter_out, NUM_TAPS, numSamples);
31              DSP_blk_move_gen (&filter_in[numSamples], history, FILTER_ORDER);
32          }
33          else {
34            memcpy(first_filter_in, buffer_input, bufferSize);
35            DSP_fir_gen(first_filter_in, coeff, &filter_out[NUM_TAPS], NUM_TAPS, ↩
                  numSamples - NUM_TAPS);
36            DSP_blk_move_gen (&first_filter_in[numSamples - FILTER_ORDER], history↩
                  , FILTER_ORDER);
37            first_chunk = 0;
38          }
39                memcpy(buffer_output, filter_out, bufferSize);
40      }
```

The two functions from the DSPLIB are:

DSP_fir_gen The FIR filter function. This function was written using intrinsics to optimize the speed of the code. From its prototype, shown in listing 3.4, we can see that it operates on signed 16-bit integers arrays. This function unrolls the inner loop by loading 8 bytes per time step from the memory (four samples) and computing four output samples per instruction cycle taking advantage of the VLIW architecture. For these reasons, the function requires that the output array must be aligned to a 4-byte boundary.

DSP_blk_move_gen This function moves data from the source address to the destination address exploiting the VLIW parallelism as the function above.

In the prototypes of the two functions, we can see that some pointers are declared using the keyword `restrict`. By means of this keyword, the programmer ensures to the compiler that there is only one pointer used to access this area of memory. In this way, the compiler assumes that the problem of pointer aliasing[7] is negligible hence enabling cache optimizations.

The following sections will explain in detail the software implemented for each IPC mechanism and how the module in question was integrated in the general test software framework. For details regarding IPC modules, refer to section 2.6.1. The timing code is not described in these sections but is explained in section 3.6.1.

3.4.2 PROC module

The test software exploiting the PROC module is called `proc_FIR` and the test command to run the program is:

```
./proc_FIR  <path of DSP executable> <input WAVE file> <output WAVE file> /
<shared memory address> <chunk size>
```

As we can see, the fourth input argument is the address of the shared memory from which the buffers for storing chunks will be allocated. As we will consider later,

[7] The situation when the same memory location can be accessed using different pointers, is called *pointer alising*. This situation implies that the compiler is unable to utilize code optimizations involving caches and may reduce the degree of instruction level parallelism.

this implies a deep understanding and knowledge of the memory needed by the program. For our specific case, the address should be set at 0x87EF0080 in normal execution mode and at 0x87EF0200 if in CPULOAD mode for measuring the DSP load (see section 3.6). These addresses point to the DSPLink shared memory and they are physical addresses (remember that DSPLink does not support the MMU on DSP side.

In this application, the GPP writes data directly into the DSP memory space of the shared memory. This is done through functions `PROC_read()` and `PROC_write()`. In order to synchronize the two processors, the NOTIFY module is needed. The application control is shown in the figure 3.7. On the GPP side, the last step of the `GPP_Create()` function is to wait on a binary semaphore for the DSP to complete the set up operations (done in the `TSKDSP_create()`) function on the the DSP side. The semaphore is provided by the hosting operating system and not by the DSPLink. In our case, the semaphore has the `sem_t` type, provided by the POSIX library and defined in the header file `semaphore.h`. By means of the NOTIFY module, a callback function is associated with a particular IPS event. After that the DSP finishes its set up, a notification is sent to the GPP via `NOTIFY_notify()` specifying the ID of the target GPP (in the case of a multiprocessor environment) and the ID of the event in the IPS table. On the GPP side the callback function is executed when such event happens. The callback function, in our case, is simply a *post* on the same semaphore on which the process on the GPP side was *pending*. In this way, the thread on the GPP can safely continue being sure that the DSP is operating and ready to continue the execution flow. The NOTIFY module provides the possibility to send 32-bits of data as a payload of a notification message. Exploiting this feature, the GPP communicates the addresses where the data must be exchanged in the DSP memory. Specifically, the first address (`dspAddr1`) is the address passed to the application via the command line argument and it is the address in which the GPP writes and the DSP reads data. The second address (`dspAddr2`) is the value of the first address incremented by the chunk size. It represents the address at which the DSP writes and GPP later reads data and it is contiguous to the first memory area. After that, the GPP starts iteratively sending chunks to the DSP. Data are written to the `dspAddr1` address via `PROC_write()` passing as arguments the ID of the target processor, the address where data must be stored, the size of the data, and the data pointer. At this point, the GPP sends a notification to the DSP to communicate that data are ready in the DSP memory area and waits on a semaphore for the data to be filtered. The DSP, which was waiting on a semaphore for the data, can now process data and write it back to the address `dspAddr2`. A notification is sent back to the GPP to communicate that data are filtered and now available. This loop is performed until no more chunk remains on the GPP side to be elaborated. A possible optimization of this schema is the *double buffering*. With this technique, there are two buffer areas, one pair where the DSP is processing data and writing its results and the other where the GPP is reading out data and then writing in new data to be processed in the next iteration. The role of the buffer pairs are swapped in each iteration. The GPP should write data into the spare buffer

before waiting on the semaphore.

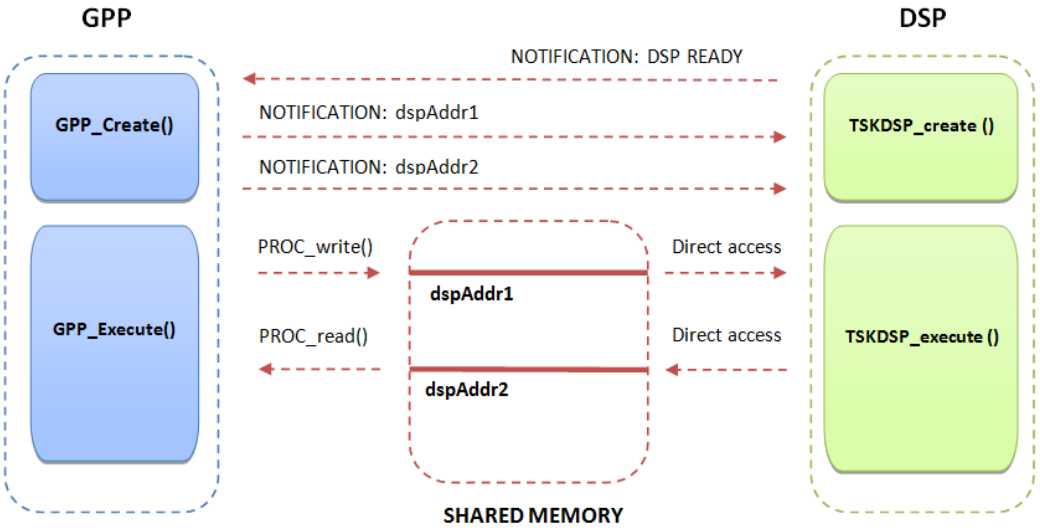

Figure 3.7. PROC module testing software

3.4.2.1 Advantages

The main advantage of the PROC module is that the communication protocol is straightforward. The complexity of the code for the communication purpose is very simple. Another advantage is that the *interrupts* are used to signal to the processes that updated data are available. In this way the processor load is lowered since the thread is suspended and the processor is yielded.

3.4.2.2 Shortcomings

An important drawback of the PROC module is that the addresses are manually decided by the programmer who must know in details the memory requirements of the application. This dramatically limits the scalability of the software. In fact, the programmer must evaluate if a change in the code affects the memory and has manually to avoid memory overlaps. With respect to parallelism, there are no memory protection mechanisms implemented by the PROC module. This means that data coherency problem can arise if concurrent processes operate on the same data.

3.4.3 MPCS module

Memory protection is provided by the MPCS module. The software using this module is called `mpcs_FIR` and is invoked with the command:

```
./mpcs_FIR  <path of DSP executable> <input WAVE file> <output WAVE file>/
```

```
<chunk size> <polling interval>
```

The communication schema in this application is similar to the PROC testing software. While in the PROC software the GPP writes data directly into the DSP memory, by using the MPCS module, data are exchanged by means of the POOL component while the MPCS module ensures the mutual exclusion while accessing data. By using `POOL_alloc()`, two buffers in the POOL memory space are allocated. In the first buffer a control structure is stored. This structure consists of a flag indicating whether the data are valid or not, the ID of the author of data, and the ID of the chunk currently present in the data structure which is the second buffer. The addresses of these two buffers must be translated by the GPP into the DSP memory space (using `POOL_translateAddr()`) and communicated to the DSP (since the DSP MMU system is not supported by DSPLink). The translated addresses are transmitted via notification messages as showed in figure 3.8. When the set up phase ends, the two processors start checking the control structure values in order to understand if data are updated or not. Access to the common buffers in shared memory is regulated by MPCS module. Before accessing the shared object, each processor invokes `MPCS_enter()`. This function blocks if another process has already gained access to the shared object. The lock is released by calling `MPCS_leave()`. If the author field of the control structure is different from the ID of the current processor, then the data are read and updated, and consequently, the control structure is updated. The thread is then suspended by means of the system call `usleep()` for the time period specified as fifth argument in the command line. This mechanism results in a *polling* behavior with the specified interval between pooling the status of the object. So, while in the PROC case interrupts were used to synchronize changes to data, in this case synchronization is achieved by using polling and a control structure.

Since the POOL module is used to handle data in the shared memory, is sufficient to copy data from the GPP memory space to the DSPLink memory segment through `memcpy()`. Nevertheless, to achieve data synchronization in the shared memory and to be sure to work with latest data on GPP side, the function `POOL_invalidate()` must be called before reading both control and data structure, and `POOL_writeback()` after having updated some data. Fgure 3.8 summarizes this communication pattern. On the DSP side the access to the buffers in the POOL memory space is done using the translated addresses. Also in this case the data synchronization must be forced by invalidating the cache before reading (`HAL_cacheInv()`) and writing back the cache updated content into the shared memory (`HAL_cacheWbInv()`).

3.4.3.1 Advantages

The MPCS module has the same advantages as the PROC module in terms of low programming complexity. In addition, one of the PROC drawbacks is solved, as MPCS provides the mutual exclusive access the shared memory. By using the

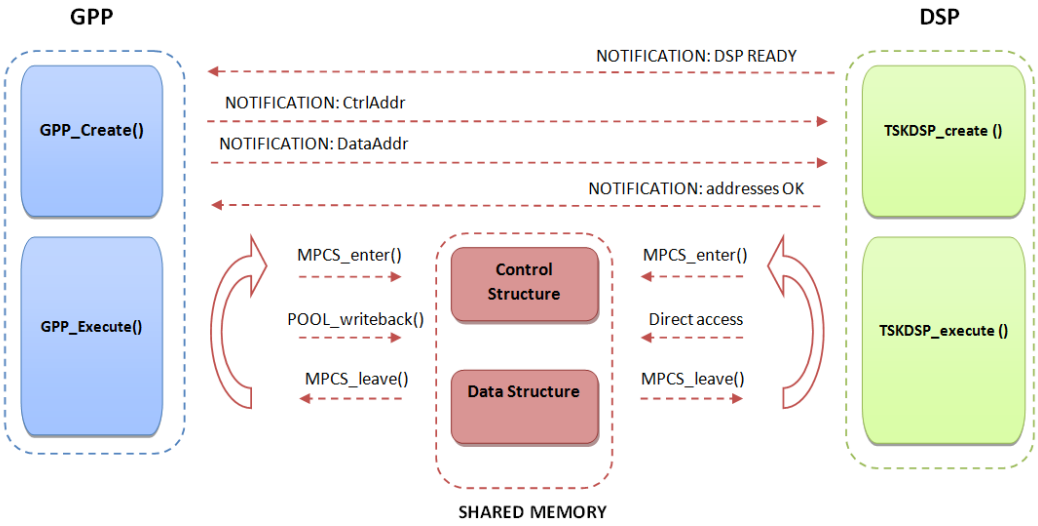

Figure 3.8. MPCS module testing software

MPCS, many concurrent processes can safely access to the shared memory without compromising data coherency.When utilizing the PROC module, the programmer is not forced to know in detail the memory allocation of the application and no manually allocated addresses must be defined a priori. Hence, a higher degree of scalability is reached.

3.4.3.2 Shortcomings

The most evident shortcoming in this application is the processor load. Since a *form of busy waiting* is used, the GPP process is not suspended (and the processor is not yielded) while waiting for the data to be elaborated. Another shortcoming is the need for a control structure to keep track of changes done to the data structure. This leads to a waste of limited shared memory. The amount of memory needed for the control structure depends on the application. The minimum control structure contains the ID of the last process that wrote data in the shared memory.

3.4.4 CHNL module

The `channel_FIR` uses the CHNL module for the communication. The application is invoked through the command:

```
./channel_FIR  <path of DSP executable> <input WAVE file> /
<output WAVE file> <chunk size>
```

The communication between processors is based on *unidirectional logical data channels*. Such channels permit communication between a single reader and a writer. In order to complete the exchange of data between the GPP and the DSP , two channels have been created using the function `CHNL_create()`. The channel

from the GPP to the DSP is called CHNL_ID_OUTPUT while the one from the DSP to the GPP is named CHNL_ID_INPUT, as shown in figure 3.9. One of the arguments of this function is an instance of the `ChannelAttrs` structure. The role of this structure, defined in the `chnl.h` header file provided by DSPLink, is to define some parameters of the channel - such as the endianess, the data size, and allocates the direction of the channel. The GPP allocates these channels on both directions and the buffers that should be transferred through the channels by invoking the function `CHNL_allocateBuffer()`. The association between these buffers and the channel is done by means of instances of the `ChannelIOInfo` structure. Some fields in this structure are the size and the pointer to the buffer. The data transfer is carried out according to the *issue-reclaim model*. According to this model, the channel created above is first opened. Then, a buffer for the I/O is issued. In case of a reader, an empty buffer is issued, while in the case of a writer, a filled one is issued. Afterwards, the client reclaims the buffer, this means that the client waits until the buffer is sent through the channel (writer) or that the buffer is filled (reader). The buffers are managed by the link driver, that is the interface between the application and the physical device, and the ISR module, for the interrupts. The reclaim function is blocking and the waiting time can be unlimited or a time out can be set. The functions to achieve this are `CHNL_issue()` and `CHNL_reclaim()` on the GPP side. On the DSP side the Serial I/O module (SIO) is exploited and the functions are `SIO_issue()` and `SIO_reclaim()`.

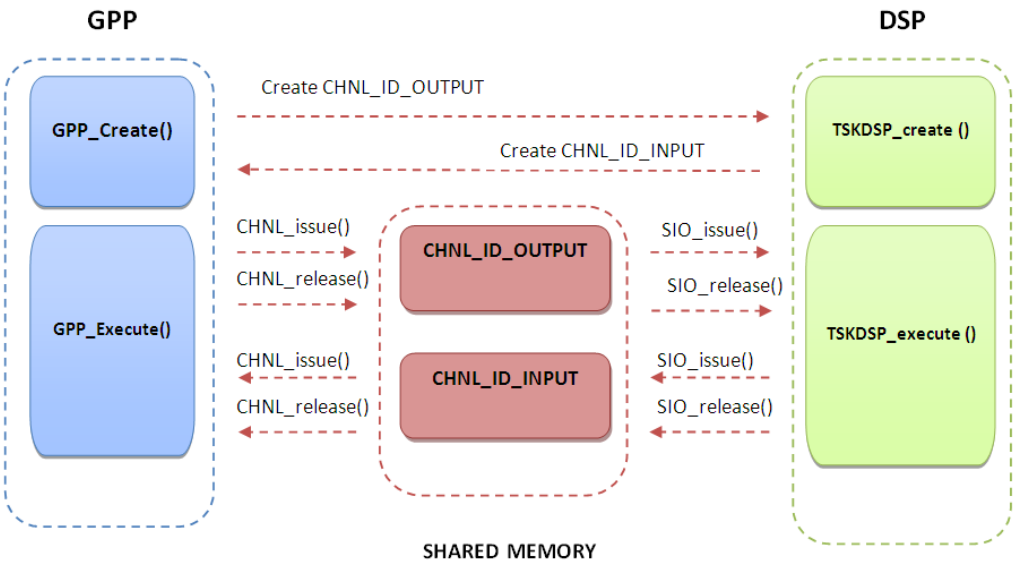

Figure 3.9. CHNL module testing software

3.4.4.1 Advantages

Firstly the concept of exchanging data through channels is very straightforward and easy to be understood. Furthermore, the buffer allocation is done by the DSPLink POOL module so that the programmer is not obliged to manually allocate the memory space. The CHNL module also offers the opportunity to allocate several buffers for one channel and send several consecutive buffers at the same time. Another advantage is the issue-reclaim model hides from the programmer the complexity of managing the buffer delivery and the interrupt mechanism.

3.4.4.2 Shortcomings

The first shortcoming is that the channels are unidirectional and so only one reader and one writer can use the channel at the same time. Furthermore, the two ends of the channel must be decided at the creation time and the channel direction cannot be changed dynamically. In case of multiple processors, N*(N-1) ** channels are needed to achieve the full connectivity.

3.4.5 MSGQ module

Another way of exchanging data among processors, is via messages. Each processor needs at least one *message queue* to be able to receive messages. A message queue has one reader and can have multiple writers. These capabilities are provided by DSPLink by means of the MSGQ module. The software exploiting this module is called `msgq_FIR` and the command to run it is:

```
./msgq_FIR  <path of DSP executable> <input WAVE file> /
<output WAVE file> <chunk size>
```

In order to be able to communicate, both the GPP and the DSP each have their own message queues. A message queue is an instance of the `MSGQ_Queue` type defined in `msgq.h`. This message queue must be created and opened on the reader side. The MSGQ module is functionally parallel on both GPP and DSP sides. In practice this means that the same DSPLink functions are used in the design even though different implementations are realized depending on the operating system. In fact, the DSPLink MSGQ module is based on the DSP/BIOS homonymous module on the DSP side.

The first step is to open the message queue referred by a unique system-level name using the function `MSGQ_open()`. This function returns a handle to the message queue. This handle is a 32-bit value made up of two 16-bit integers. The first integer is the processor ID, while the second field integer contains the ID of the message queue on the processor. DSPLink requires that a message queue for the handling of asynchronous error messages must be defined. Such a message queue can be the same as the one for the reception of application messages. A fundamental component of the system is the *message queue transport protocol*. The function of this component is to locate and deliver messages over the complete system. To locate the remote queues, the function `MSGQ_locate()` must be invoked passing

**From now on, N is the number of processors in a multi processors system.

in the name of the remote queue and the remote processor ID. The GPP waits for the DSP to set up the queue and for the queue to be located. The same operations are performed on the DSP side. An exchanged message is defined as follows:

```
typedef struct Message_content_tag {
    MSGQ_MsgHeader msgHeader;
    Void *dspSrcDataAddr;
    Char8 dataBuffer [CHUNK_L];
} Message_content ;
```

The message header `msgHeader` is mandatory in every message since it contains information necessary to deliver the message. Although according to the documentation DSPLink supports variable length payloads, errors occurred when variable length payloads were sent. To solve this problem, the payload `dataBuffer` was defined statically with the maximum application chunk size in the definition. However, when allocating the POOL buffer where the message should be stored (via `MSGQ_alloc()`), the effective chunk size is used. In this way a pseudo variable payload is achieved. Furthermore, the message size should be aligned to the DSP cache word size. The `dspSrcDataAddr` is the translated address of the payload to the DSP memory space. Then, in order to track messages, the message ID is set as to be the chunk ID. Messages are sent invoking `MSGQ_put()` specifying the destination queue and received on the local queue through `MSGQ_get()`. The API to send a message is deterministic and non-blocking. A timeout is specified for the receiving operation. If the message is not available and the queue is empty, the client waits for the timeout to expire. This timeout can be either zero or unlimited. In this way, both synchronous and asynchronous communication paradigms can be achieved.

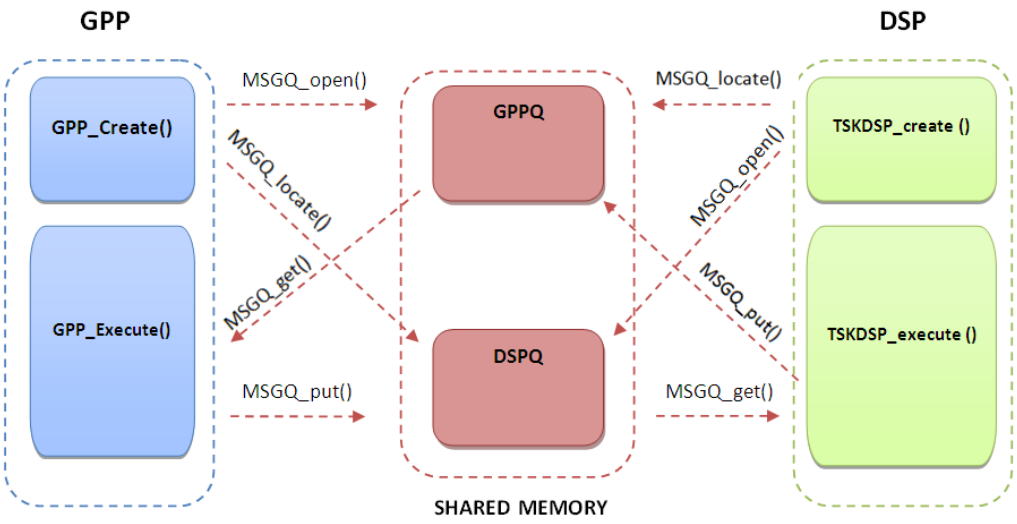

Figure 3.10. MSGQ module testing software

3.4.5.1 Advantages

MSGQ offers a good solution for realizing communication in a multi-processor system. Since a message queue can receive messages from different senders, N message queues are sufficient to enable communication between any processors. Another advantage is that the receiver does not need to know details of the sender in advance (e.g. the sender's address). All the necessary information is embedded in the message and can be retrieved by means of APIs. Furthermore, in the same message, different types of data can be sent at the same time.

3.4.5.2 Shortcomings

The first shortcoming is the need to define a maximum size payload for any message. A completely dynamically allocated payload is not achievable contrary to the DSPLink documentation. Another drawback is the need for a message header for each message. This control information wastes memory and requires additional overhead for every message that is transferred. An instance of the `MSGQ_MsgHeader` structure, requires 128 bytes of memory.

3.4.6 MPLIST module

The communication via *linked list* is achieved by using the MPLIST module in the application `mplist_FIR` invoked by the command:

```
./mplist_FIR  <path of DSP executable> <input WAVE file> /
<output WAVE file> <chunk size> <max number of elements>
```

By utilizing the linked lists provided by the MPLIST module, the programmer can operate on data out-of-order. To reflect this feature, the sequential schema of the previous application was broken and a different design was chosen. In this new program, on the GPP side two tasks are running: one writer, in charge of sending chunks to the DSP, and a reader, which receives the filtered data. The two tasks are synchronized together and with the task running on the DSP side by means of semaphores (as shown in figure 3.11). Two linked lists are created on the GPP side and they are characterized by a system-level unique name. In addition, three semaphores are used for synchronization and one MPCS object is utilized to protect a shared variable called `current`. This shared variable is the current number of chunks that are sent by the writer, but not yet received by the reader. Two semaphores, the `GPP_SemPtr_rd` and `GPP_SemPtr_wr` are used for synchronization of the GPP and DSP with the help of the NOTIFY module. Two callback functions waiting on these semaphores, are listening for two different IPS events. A third semaphore, `GPP_SemPtr_sync`, is used locally on the GPP side. When the `current` variable has reached the maximum number of elements in the list (passed as an argument to the program), the writer is suspended invoking a `SEM_pend()` on a semaphore. After the reader processes one element it wakes up the writer by means of a notification.

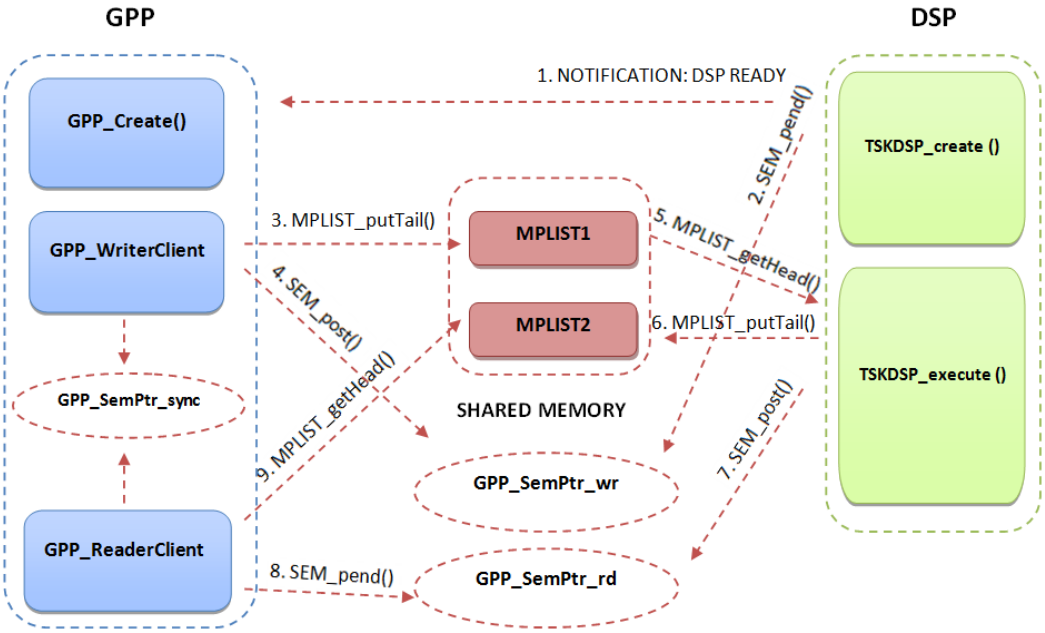

Figure 3.11. MPLIST module testing software

After the set up phase, the GPP waits for the DSP to finish its initial phase. When everything is ready, the DSP sends a notification to the GPP and waits on the semaphore `GPP_SemPtr_wr`. After that, the writer task starts sending chunks to the DSP. The chunks are appended to the tail of the MPLIST1 using the function `MPLIST_putTail()` and the variable `current` is updated after gaining access to the MPCS object. On the DSP side, two callback functions, associated with a particular IPS event ID, were defined. These functions make a post to the associated semaphore. After sending a chunk, the writer notifies the DSP so that a post to the `GPP_SemPtr_wr` is sent. The task on the DSP can now get data from the head of the linked list (`MPLIST_getHead()`), process this data, and append the output data to the tail of MPLIST2. Then, it sends a notification to the event associated with the semaphore `GPP_SemPtr_rd` and waits on the semaphore `GPP_SemPtr_wr`. The reader, which was waken up by the DSP, can now take the data from the head of the MPLIST2. When the number of `current` chunks is equal to the maximum number of elements in the linked lists (specified as an argument to the program), the writer waits on the semaphore `GPP_SemPtr_sync` and it is waken up by the reader by means of a post.

3.4.6.1 Advantages

An important advantage of the MPLIST module is the possibility to process data out-of-order. In our application, this feature is not important since we are dealing with a sequential stream of data. In a multi-processor environment, communication

can utilize a single doubly-linked list since MPLIST supports multiple readers and multiple writers. The identity of the senders can be retrieved by means of information included in the header of the messages.

3.4.6.2 Shortcomings

The MPLIST handles messages in a similar way as the MSGQ module. For this reason, the same disadvantages of the MSGQ module are also valid for the MPLIST. These drawbacks include the pseudo-dynamic payload length and the overhead of the message header. In case of many messages with a small payload, the data overhead may be significant.

3.4.7 RINGIO module

The last IPC mechanisms is the *circular buffer* provided by the RINGIO DSPLink module. The software utilizing this module is called `ringio_FIR` and the command to execute it is:

```
./ringio_FIR <path of DSP executable> <input WAVE file> /
<output WAVE file> <chunk size> <circular buffer size>
```

Two circular buffers are used for the communication: RINGIO1 and RINGIO2. The first buffer receives data from the GPP. The DSP reads data from RINGIO1 and writes to RINGIO2, where the GPP can collect the filtered data. As for the MPLIST case, two parallel threads are executed on the GPP: a writer and a reader. During the initialization phase, the GPP side creates the RINGIO1 buffer through the function `RingIO_create()` while the DSP creates the RINGIO2. After the creation of the circular buffer, the client needs to open it. This is done by using `RINGIO_open()`. Two important arguments to be passed to this function are:

- *openMode*: the mode with which the buffer is opened (reader or writer mode); and
- *flag*: flags to define the caching and notification behavior. On the GPP side the flag used is RINGIO_NOTIFICATION_ONCE. By means of this flag, the client gets a notification whenever an attempt to acquire a buffer from RINGIO fails. On the DSP side, since the shared memory is cacheable, the configuration of cache coherency protocol implemented by DSPLink is specified by means of the flags: RINGIO_DATABUF_CACHEUSE, RINGIO_ATTRBUF_CACHEUSE, and RINGIO_CONTROL_CACHEUSE. These buffers requires data coherency for the data, attribute, and control buffers.

The next step is to register the callback function that must be executed when an event happens on the associated RINGIO. This is not done by using the NOTIFY component, but rather this functionality is embedded in the RINGIO module in the form of the function `RingIO_setNotifier()`. The first step in

the communication is done by the writer, which sends attributes over RINGIO1 to the DSP (via `RingIO_setAttribute()`) which is initially suspended waiting on a semaphore. This attribute, RINGIO_DATA_START, causes the execution of the callback function associated with the RINGIO1 on the DSP side. This function posts to the semaphore on which the DSP task is suspended waiting on. The DSP sends the RINGIO_DATA_START on RINGIO2 to the reader. From now on, the writer writes data to the RINGIO2 and the reader collects data back by reading from RINGIO2. The data are sent via the circular buffer according to an *acquire-release* mechanism. When the writer wants to write data, it acquires an empty buffer of the desired size from RINGIO through `RingIO_acquire()`. This function blocks if no empty buffer is available. When the function returns the empty buffer, the writer fills the buffer and releases it by calling `RingIO_release()`. Conversely, the reader acquires filled buffers and releases empty ones. At the end of the iteration, when all chunks have been sent, the writer sets the attribute RINGIO_DATA_END that is received by the DSP and propagated to the reader on the GPP side.

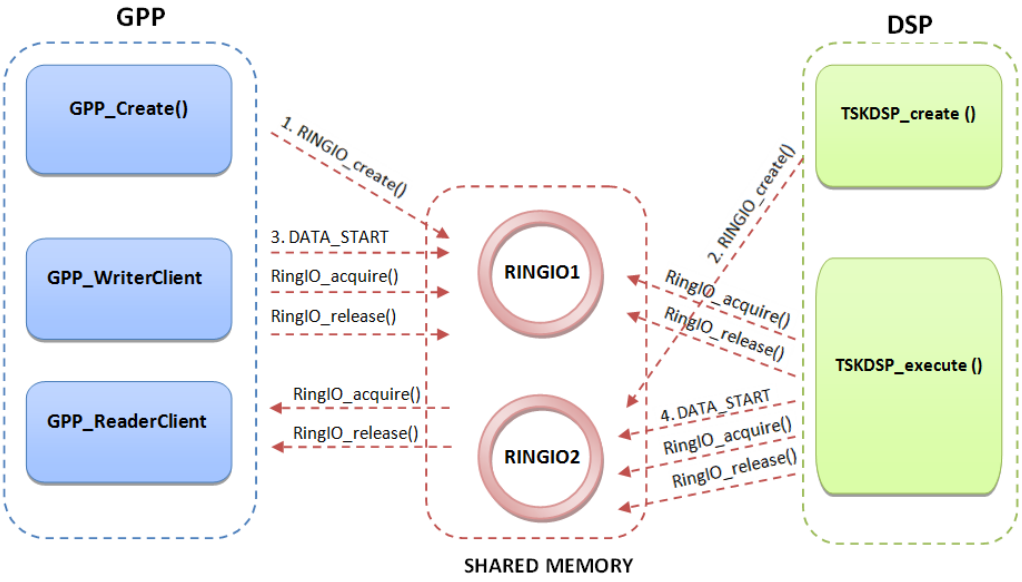

Figure 3.12. RINGIO module testing software

3.4.7.1 Advantages

RINGIO perfectly suits data streaming since is very flexible and processes data in order. Additionally, if a buffer of the required size is not available, a smaller buffer is retrieved so that communication may continue without evident interruptions. Nevertheless, this feature was not used for this project. In our case, only a buffer of the required size is acquired. RINGIO helps the programmer by embedding some of

the necessary synchronization structure. The NOTIFY and MPCS tools are used internally by the RINGIO component to ensure data coherence and synchronization.

3.4.7.2 Shortcomings

The main drawback of RINGIO is its high programming complexity. Although the RINGIO hides part of the synchronization matter, the programmer must handle the buffers' boundaries if partial buffers are sent. The complexity is even higher if variable length attributes are exploited.

3.5 GPP + NEON Solution

The first system configuration that was considered to port a SDR to the OMAP3530 platform, was to exploit the DSP. Another option is to exploit the NEON coprocessor provided by the ARM Cortex-A8. NEON is a SIMD coprocessor (also known as *vector processor*). Physically, it consists of a chip directly connected to the data and control bus of the ARM processor. A SIMD architecture is able to perform the same operation in parallel over different sets of data by using a single instruction, hence increasing the processor's performance. The level of parallelism depends on the algorithm and details of the instruction set. Since the largest NEON register is 128-bits, if the algorithm performs operations with 8-bit values, the coprocessor can execute at most 16 operations simultaneously. However, the instruction level parallelism can be limited by other constraints such as memory thoughtput and the loop overhead.

The hardware features of the NEON processors are (according to [34] and [35]):

- 16-entry instruction queue;
- dual-view register file: thirty-two 64-bit registers (D0...D31) or sixteen 128-bit registers (Q0...Q15);
- 6 stage execution pipeline (integer and single-precision floating point);
- 3 SIMD integer pipelines;
- a load-store/permute pipeline;
- 2 SIMD single-precision floating-point pipelines;
- a non-pipelined Vector Floating-Point unit (VFPLite); and
- 12-entry load data queue.

The NEON coprocessor is decoupled from the main processor by means of an instruction queue (as shown in figure 3.13). The Cortex-A8 can issue up to two valid instructions per clock cycle to the NEON unit. The NEON 10-stage pipeline (including a 6-stage execution pipeline), starts after the ARM integer pipeline in the case of a NEON instruction. All instructions are issued and retired in-order. The ARM processor makes NEON's job easier by solving all branch mispredictions

and exceptions *before* the NEON pipeline. More importantly, the ARM processor ensures NEON a zero-load penalty when using the L1 cache. In fact, all NEON addresses are computed by the processor before the NEON pipeline starts and data can be loaded in advance. The **integer** pipeline is made up of three pipelines: the *MAC* (multiply-accumulate), *ALU*, and the *shift*. The **loadstore/permute** pipeline handles all load and store operations and data transfers to the integer unit. Moreover, it performs data permute operations (interleave or de-interleave on memory accesses). The NEON **floating-point (NFP)** pipeline consists of two pipelines: *floating-point ADD* and *floating-point MUL*. The NEON coprocessor also contains the **VFPLite**[8] unit, this is a non-pipelined implementation of the ARM VFPv3 Floating-Point Specification (see section 2.4.1) for backward compatibility with the existing ARM floating-point code [35].

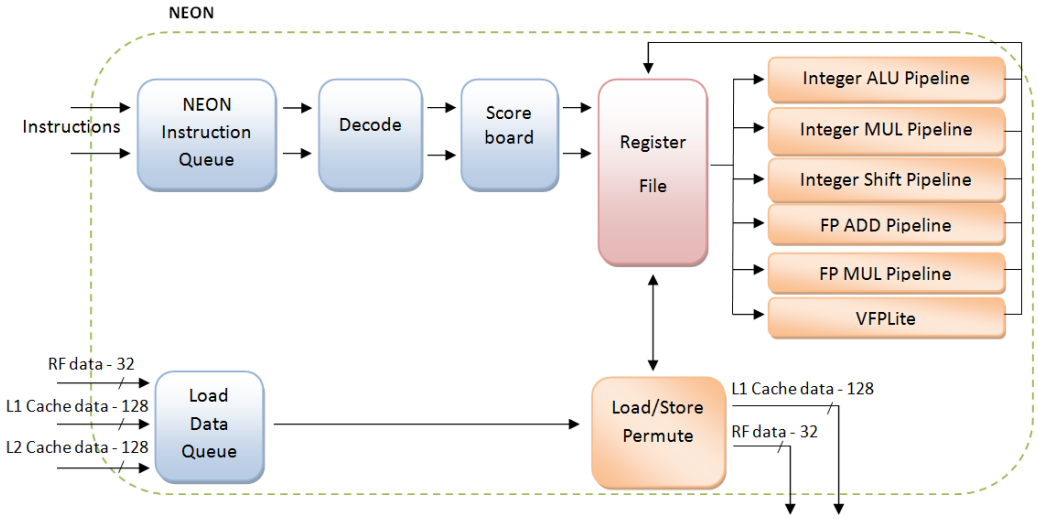

Figure 3.13. NEON block diagram (adapted from [34])

The **advantages** of NEON are:

- aligned and unaligned access to data for a better vectorization of the code;
- support for integer and floating-point operations;
- being a coprocessor, NEON is tightly coupled to the ARM core. This leads to a series of advantages: an unified view of memory and a single instruction execution flow. The latter feature simplifies the development tool, since only one development tool chain is needed for the two cores; and
- data through-put performance is enhanced thanks to an efficient handling of data by the register file. Due to the high data parallelism, the register file can load a large amount of data from memory with one access minimizing the number of accesses to memory.

[8]The term "lite" indicates a version not limited in functionality, but in area and performance.

There are three way to develop code for the NEON processor: using a vectorizing compiler, using NEON intrinsics, and writing NEON assembly code. Since the target was NEON, the test software was modified according for the three different ways of developing NEON code. In this way, a more complete analysis of the NEON performance could be made.

3.5.1 Vectorizing compiler

The first way of generating code for NEON is to exploit a vectorizing compiler. The main advantage of this method is that the programmer does not need to change their code. The vector code is generated by the compiler by means of compiler options. Listing 3.5 shows the straightforward C code to implement a FIR filter:

Listing 3.5. FIR Filter: compiler vectorization

```
1  Void FIR_core(int* __restrict longInput, short* __restrict dataOut, int  coeff↩
       [], int halfSize) {
2
3    int i = 0;
4    int j = 0;
5    int sum = 0;
6
7    for(j=0; j < halfSize; j++) {
8      sum = 0;
9      for (i=0; i < BL; i++) {
10       sum +=  longInput[i+j] * coeff[i];
11     }
12     dataOut[j] = (sum >> 15);
13   }
14 }
```

In our case, the compiler based on GCC (`arm-angstrom-linux-gnueabi-gcc`) is able to vectorize the code. In order to do it, the following options were used:

```
arm-angstrom-linux-gnueabi-gcc  -O3  -Wall -c  -mtune=cortex-a8 -mfpu=neon
-ftree-vectorize  -ftree-vectorizer-verbose=15 -mfloat-abi=softfp
-march=armv7-a
```

In order to generate vectorized code, the optimization level `O3` is required by the compiler. Furthermore, the target processor and the version of the ARM architecture need to be specified. The target coprocessor is specified by saying `-mfpu=neon`. The option `-ftree-vectorize` tells to the compiler to produce vectorized code and the output of this operation is shown thanks to the option `-ftree-vectorizer-verbose`. As we can see from listing 3.5, the keyword `restrict` was used in the array definitions (see section 3.4.1). Unfortunately, GCC poorly vectorizes the code. As can be seen by examining the generate code, very little change has been made to the code in comparison with the FIR code developed for the DSP. Specifically the FIR function deals with `int` arrays instead of `short` arrays. A conversion of the array, from `short` to `int` was mandatory in order for the compiler to vectorize the code. In fact, in GCC starting from version 4.2 there

is a bug[9] that does not let the compiler vectorize code in which a cast is done. Therefore the conversion is performed outside the FIR function and must be taken into account during the analysis of the NEON performance. This problem was not detected when using the ARM C compiler `armcc`. However, this compiler does a better job in vectorizing the code since is able to vectorize code with casting without problems. Unfortunately, `armcc` is not freeware and for the sake of coherency with the GPP + DSP solution, the `arm-angstrom-linux-gnueabi-gcc` was used above with the above workaround.

3.5.2 NEON Intrinsics

As the compiler is not always able to do a good job when vectorizing code, especially when dealing with complex functions, a second way of developing NEON code is using *intrinsics*. The NEON intrinsics are a combination of C and assembly code giving the programmer direct control over the SIMD functionalities. The programmer can still use a high level programming language (C language), but can directly control the processor's operations. Furthermore, the instrinsics are designed to limit errors related to data types mismatches. In order to use these intrinsics, the option `-mfpu=neon` must be passed to the compiler and the header file `arm_neon.h` must be included. Listing 3.6 shows the code for the FIR filter when using intrinsics to process four parallel vectors of shorts:

Listing 3.6. FIR Filter: NEON intrinsics

```
1  #include "arm_neon.h"
2
3  Void FIR_core(Char16* longInput, Char16* dataOut, int16_T coeff[], int BL, Int32↩
       halfSize) {
4
5    Int32 i = 0;
6    Int32 j = 0;
7    Int32 sum = 0;
8    int16x4_t coeff_vector;
9    int16x4_t input_vector;
10   int32x4_t result_vector;
11
12   for(j=0; j < halfSize; j++) {
13       sum = 0;
14       //Set all lanes to the same value 0
15       result_vector = vdupq_n_s32(0);
16
17       for (i=0; i < (BL >> 2) ; i++) {
18         coeff_vector = vld1_s16(&coeff[i*4]);
19         input_vector = vld1_s16(&longInput[i*4 + j]);
20         //vmlal: result_vector = result_vector + coeff_vector * input_vector
21         result_vector = vmlal_s16(result_vector, coeff_vector, input_vector);
22       }
23       sum += vgetq_lane_s32(result_vector, 0);
24       sum += vgetq_lane_s32(result_vector, 1);
25       sum += vgetq_lane_s32(result_vector, 2);
26       sum += vgetq_lane_s32(result_vector, 3);
```

[9] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=26128

```
27
28        if (BL % 4) {
29          for(i=BL - (BL % 4); i< BL; i++ )
30            sum += longInput[i+j] * coeff[i];
31        }
32        dataOut[j] = sum >> 15;
33    }
34  }
```

The code above was compiled with the command:

```
arm-angstrom-linux-gnueabi-gcc -O3 -Wall -c -mtune=cortex-a8 -mfpu=neon
-mfloat-abi=softfp -march=armv7-a
```

In the code above, the inner loop was manually unrolled with a factor of 4. In the lines 18 and 19, four values from the arrays `coeff` and `longInput` are loaded as 16-bit signed integers into variables of type `int16x4_t`. This variable type (defined in the `arm_neon.h`), is a vector of four `short` values. The multiply-accumulate operation is performed over these two vectors in the line 21. The result is a vector (`result_vector`) of four 32-bit integers. The last step (lines 23-26), is to sum up the value of every lane. The line 28 handles the remaining samples in case the coefficient array has a length which is not a multiple of the unrolling factor (4 in our case).

The list of all intrinsics is available in the `arm_neon.h` header file. For the mapping between intrinsics and ARM assembly, refer to [36].

Listing 3.7 shows part of the disassembled code of the listing 3.6.

Listing 3.7. NEON intrinsics: disassembled code

```
1  00008ae0 <FIR_core>:
2      8ae0: e1a0c00d  mov ip, sp
3      8ae4: e92ddff0  push  {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}
4    ...
5      8b24: e3a07000  mov r7, #0  ; 0x0
6      ...
7      8b6c: f461174f  vld1.16 {d17}, [r1]
8      8b70: f462074f  vld1.16 {d16}, [r2]
9      8b74: f2d128a0  vmlal.s16 q9, d17, d16
10     8b78: e2811008  add r1, r1, #8  ; 0x8
11     8b7c: e2822008  add r2, r2, #8  ; 0x8
12   ...
13     8b88: ee122b90  vmov.32 r2, d18[0]
14     8b8c: ee323b90  vmov.32 r3, d18[1]
15     8b90: e0833002  add r3, r3, r2
16     8b94: e35c0000  cmp ip, #0  ; 0x0
17     8b98: ee132b90  vmov.32 r2, d19[0]
18     8b9c: e0833002  add r3, r3, r2
19     8ba0: ee331b90  vmov.32 r1, d19[1]
20     8ba4: e0831001  add r1, r3, r1
21   ...
22     8c08: e89daff0  ldm sp, {r4, r5, r6, r7, r8, r9, sl, fp, sp, pc}
```

From this code the mapping between the instrinsics and the assembly language code is evident. The intrinsics function `vld1_s16` was mapped with the assembly language instruction `vld1.16`, the function `vmlal_s16` with the instruction

`vmlal.s16`, and the lines 23-26 of the listing 3.6 correspond to lines 13-20 of the disassembled code.

3.5.3 NEON assembly code

If neither the vectorizing compiler nor the usage of instrinsics yield the desired performance, then the last possibility is to develop NEON code is by writing in *assembly language*. Hand coding in assembly language is a way to maximize the code performance. Nonetheless, it requires lots of programming experience, a deep knowledge of the processor architecture, and has a considerable learning curve. NEON assembly instructions are listed and explained in [36] and [37, chapter 5]. The assembly language version of the FIR filter code is shown in listing 3.8.

Listing 3.8. FIR Filter: NEON assembly

```
1     .arch armv7-a
2     .fpu neon
3     .text
4     .align  2
5     .global FIR_core
6     .type FIR_core, %function
7
8  FIR_core:
9
10   push {r4-r12}      @ backup copy of registers
11                @ r0 <- longInput
12                @ r1 <- dataOut
13                @ r2 <- coeff
14                @ r3 <- BL
15   mov r4, #0         @ r4 <- j = 0
16   cmp r3, #512
17   blt .Lno512
18   ldr r11, [sp, #8]    @ r11 <- dataLen
19   b .Lafter512
20 .Lno512:
21   ldr r11, [sp, #4]
22
23 .Lafter512:
24   mov r9, #0         @
25   mov r10, r0        @ r10: backup copy of longInput address
26   mov r12, r2        @ r12: backup copy of coeff address
27
28 .Lstart_outer:
29   mov r6, #0         @ r6 <- sum = 0
30   vmov.i32 q0, #0      @ q0 <- result_vector = 0. q0 is a 128-bit vector, four ↩
         integers
31   mov r5, #0         @ r5 <- i = 0
32   mov r2, r12        @ restore original coeff address
33   mov r0, r10        @ restore original longInput address
34   add r0, r0, r4       @ longInput + j
35
36 .Lstart_inner:
37   vld1.16 d2, [r0]     @ load four short int from the longInput to d2 (64-bit ↩
         vector)
38   vld1.16 d3, [r2]     @ load four short int from the coeff to d3 (64-bit vector)
39   vmlal.s16 q0, d2, d3  @ multiply-accumulate: q0 = q0 + d2 * d3
40   add r5, r5, #4       @ i += 4
```

```
41    add r0, r0, #8      @ longInput += 8 (jump to the next 4 short int)
42    add r2, r2, #8      @ coeff += 8 (jump to the next 4 short int)
43    cmp r5, r3        @ compare i and BL
44    blt .Lstart_inner   @ if i < BL, restart inner loop, else quit the inner loop
45
46  .Lend_inner:
47    vmov.32 r7, d0[0]   @ r7 = first lane of q0
48    add r6, r6, r7      @ sum += first lane
49    vmov.32 r7, d0[1]   @ r7 = second lane of q0
50    add r6, r6, r7      @ sum += second lane
51    vmov.32 r7, d1[0]   @ r7 = third lane of q0
52    add r6, r6, r7      @ sum += third lane
53    vmov.32 r7, d1[1]   @ r7 = fourth lane of q0
54    add r6, r6, r7      @ sum += fourth lane
55
56    #case BL not multiple of 4
57    sub r9, r5, r3      @ r9 = BL % 4
58    cmp r9, #0        @ BL % 4 == 0
59    beq .L_multiple
60    eor r5, r5, r5      @ if BL % 4 != 0, a new loop starts. i = 0
61
62  .L_multiple_cycle:      @ loop for computing remaining samples
63    smlal r8, r6, r0, r2  @ multiply-accumulate: r8|r6 = r8|r6 + r0 * r2
64    add r5, r5, #1      @ i++
65    add r0, r0, #2      @ longInput += 2 (pointer to the next short int)
66    add r2, r2, #2      @ coeff += 2 (pointer to the next short int)
67    cmp r5, r9
68    blt .L_multiple_cycle @ if no more samples, quit the loop
69
70  .L_multiple:
71    #update dataOut
72    asr r6, r6, #15     @ sum = sum >> 15
73    strh r6, [r1]     @ dataOut = sum
74    add r4, r4, #2      @ j += 2
75    add r1, r1, #2      @ dataOut += 2
76    cmp r4, r11       @ j == dataLen
77    blt .Lstart_outer   @ if j == dataLen, quit the outer loop
78
79  .Lend_outer:
80    pop {r4-r12}      @ restore registers
81    mov pc, lr        @ restor program counter
```

The code was compiled by using the command:

```
arm-angstrom-linux-gnueabi-as  -mlittle-endian -mcpu=cortex-a8
-mfpu=neon -march=armv7-a
```

The directives in lines 1 and 2, indicates respectively the target architecture and floating point unit. The function prototype is:

```
extern Void FIR_core(Char16* longInput, Char16* dataOut, int16_T coeff[],
int BL, Int32 dataLen);
```

According to the ABI, the first four function arguments are passed through registers `r0...r3`, while the fifth argument (`dataLen`), is passed through the stack and retrieved in the line 18 or 21. Remember that `d` registers are 64-bit sized, while `q` registers have a 128-bit length. In lines 37 and 38, four 16-bit signed integers are loading starting from the address contained in `r2` and stored in a `d` vector. In the line 39, the multiply-accumulate operation is executed and the result, a 128-bit

register, is stored in the `q0` register. Please note that the register file has a dual-view. This means that the `q` register set `q0...q15` is physically the same as the `d` register set `d0...d31`. This means that `q0 = (d1 « 16) + d0`. In lines 47-54, the value of each lane is added up in the `r7` register (this is equivalent to the `sum` variable in the previous examples). The case in which the coefficient array has a length which is not a multiple of the unrolling factor 4, is handled in lines 56-68.

If we compare the assembly code generated by intrinsics (shown in listing 3.7) with the assembly code of the listing 3.8, we can notice that no relevant differences exist concerning the vector code.

3.6 Measuring tools

Our analysis consider three parameters in order to compare the performance of the different solutions when executing the test software. These parameters are **execution time**, **GPP load**, and **DSP load**. Clearly, the DSP load will only be used for the comparison of IPC protocols in the GPP+DSP system configuration.

3.6.1 Execution time

With respect to the timing, two different types of timing were measured depending on the system configuration:

- A *chunk's round-trip time*: this timing is only relevant for the GPP+DSP configuration since it measures the time spent when transmitting chunks over DSPLink to the DSP. It includes the transmission of chunks to the DSP and back to the GPP and the computation time of the DSP. A chunk's round-trip time ($t_{roundtrip}$) can be defined as:

$$t_{roundtrip} = 2t_{transmission} + t_{dsp}$$

- *Execution time*: for the GPP+DSP configuration, this time indicates the amount of time spent by the `GPP_Execute()` function. This includes copying of data (i.e., the chunks) from the user memory space to the DSPLink memory space and vice versa (when needed) and the chunk's round-trip time. The execution time ($t_{execution}$) can be defined as:

$$t_{execution} = 2t_{memory} + t_{roundtrip}$$

  In the GPP+NEON configuration, the execution time includes the time needed by the system to execute the FIR filter functions (the different versions were explained in section 3.5). No extra times are included in the measurements.

In order to measure the test software timing, two types of *software timers* were exploited: **fine-grained** and **coarse-grained** timer functions.

The fine-grained functions were used to measure the $t_{roundtrip}$. A high degree of precision is needed when measuring such times since the round-trip time can be really small in case of small chunks. The maximum degree of precision is achieved by directly accessing the so called Cycle Counter (CCNT) register in the ARM architecture. This register counts the number of clock cycles since the register was reset. It can be directly read by a user application, while some restrictions concern attempting to write it (e.g. the register must be disabled before being written to in order to ensure a deterministic behavior). The CCNT register is controlled by the Performance Monitor Control (PMNC) register. The functions used to handle the CCNT register are shown in listing 3.9.

Listing 3.9. CCNT register functions

```
1  /* Enable the CCNT register */
2  inline void ccnt_enable(void) {
3    __asm__ volatile ("mov r1, #0x80000000");
4    __asm__ volatile ("mcr p15, 0, r1, c9, c12, 1" ::: "r1");
5  }
6
7  /* The PMCR register is accessed. The following operations are performed:
8    - Disable Cycle count divider
9    - Reset cycle counter
10   - Enables all counters
11  */
12 inline void ccnt_start(void) {
13   __asm__ volatile ("mov r0, #5");
14   __asm__ volatile ("mcr p15, 0, r0, c9, c12, 0" ::: "r0");
15 }
16
17 /*  The value of the CCNT (cycle counter register) is returned */
18 inline Uint32 ccnt_read(void) {
19   Uint32 counter;
20   __asm__ volatile ("mrc p15, 0, %0, c9, c13, 0" : "=r"(counter));
21     return counter;
22 }
23
24 /* Stop the cycle counter */
25 inline void ccnt_stop(void) {
26   __asm__ volatile ("mcr p15, 0, %0, c9, c12, 0" :: "r"(1<<5));
27 }
```

The instructions `mrc` and `mcr` are called *coprocessor instructions*. This class of instructions causes the ARM processor to raise an undefined instruction trap. This happens when the processor is not able to interpret the instruction. When such a trap happens, the ARM processor starts a handshaking with the all coprocessors in order to establish if the instruction is a valid coprocessor instruction and if there is a coprocessor able to execute it (further details at [38, appendix A]). The `mrc` and `mcr` instructions, belong to a class of instructions used to transfer 32-bit values from the ARM processor to a coprocessor and vice versa. The instruction `mrc` transfers a value from the ARM processor to the processor. The `mcr` performs the opposite operation. Their syntax is:

```
MRC{cond} <cp#>,<op>,<ARM src>,<lhs>,<rhs>,{info}
MCR{cond} <cp#>,<op>,<ARM dest>,<lhs>,<rhs>,{info}
```

where cp# is the coprocessor ID (0-15), op is the operation code (0-7), ARMsrc / ARMdest is the ARM source/destination register (0-15), lhs and rhs are coprocessor registers (0-15), and info is the filed for extra information (0-7). The function ccnt_enable() enables the CCNT register by setting to 1 the enable bit in the PMNC register. The function ccnt_start() sets some fields in the PMNC register. It resets the value of the CCNT register, disables the cycle count divider[10], and enables other performance counters controlled by the PMNC register. The function ccnt_read() returns the current value of the cycle count while ccnt_stop() stops the counter.

Since the CCNT register is a 32-bit register and the GPP frequency is around 500 MHz, the cycle counter overflows every 8.59 seconds. In order to measure $t_{execution}$ and to be able to handle long latencies, coarse-grained timing functions were used. These functions are based on the function gettimeofday(). This function, defined in the header file sys/time.h, returns the current time with a precision of the order of microseconds. The time value is stored in a structure of the type timeval. These functions are coarse-grained since gettimeofday() introduces the overhead typical of the system calls into the measuring due to the context switching from the user space to the kernel space.

The time required to call the function gettimeofday was measured by using the fine-graded functions. The result is that the average number of clock cycles that gettimeofday requires to execute is 2294. This value corresponds to 4.588 μs for the processor used in our project.

3.6.2 GPP load

There are essentially two ways to measure the CPU load. The first one is to use measuring tools *internal* to the software by writing some code. This way usually gives the most accurate measurements. The drawback of this technique is that the programmer has to design and debug the code; and the scalability of the software is somewhat limited. The second way is to use *external* tools. These tools can be system monitors that control processes running on the operating system while producing statistics about system resources that these processes use. In order to measure the GPP load, an external system monitoring tool was chosen. The operating system running on the GPP side is a Linux distribution and a variety of system tools are readily available, are well tested, have been shown to be reliable. For all of these reasons, the **top** tool was utilized. The program top belongs to the procps package[11] and provides statistics about tasks managed by the kernel in a dynamic way in real-time. This command shows the percentage of CPU and memory usage of every process together with other information. The program top was run as background process while the test software was running. The command used to run this monitor tool was:

```
top -d 0.1 | grep <process name> | awk '{print $10"  "$13}' >> <out file>
```

[10] If the cycle count divider bit is set, the CCNT register is incremented every 64th processor clock cycle.

[11] The procps package collects some utilities that provide information about processes by using the /proc filesystem. The website of the project is: http://procps.sourceforge.net/

The -d option sets the sampling frequency. For this project, the sampling frequency was 10 Hz (a sample every 0.1 second). The output was filtered by grep so that only statistics about the process in question were taken into account. The small awk script prints just the columns of the CPU usage and process name and stores them into an output file. Another awk script was used to compute the average GPP load for each process. An important aspect that must be considered is that top consumes system resources as well. In particular, for a frequency of 10 Hz, the GPP workload (on the processor that was used) due to top itself was about 16%. This must be considered when analyzing the data.

Another profiler tool was considered: **oprofile**[12]. It is able to profile processes running on the operating system as a low overhead daemon. It supports the ARM architecture: with the option event=CPU_CYCLES:500:0:1:1, the tool samples the system every 500 clock cycles (minimum allowed value) by accessing the ARM CCNT register. However, this tool was not used to get the CPU usage in this project. The reason for this it that incoherent results were obtained during the testing. As a result of the oprofile analysis, the CPU load was constant and was not depending on the chunk size and on the number of filter's taps. This strange behavior was contradicted by top that was providing more realistic results. Nevetheless, the tool was useful in order to understand in detail which application libraries were using the CPU (by means of separate=library option). Furthermore, by using the option -vmlinux, information about CPU usage of kernel modules can be collected. An example of the output of the program when executing the msgq_FIR program with a 128-taps filter and chunk length of 2048 bytes with the -vmlinux option enabled, is listed in appendix C. The output report has been shortened for readability.

3.6.3 DSP load

The Real-time Analysis tool, integrated in the Code Composer Studio (see appendix A), can be used to analyze the DSP load. Unfortunately, the current version of the Real-time Analysis tool was not working properly with version 5.41 of DSP/BIOS. Hence, a custom solution was designed. This solution consists of creating an *idle TSK* that is executed whenever the principle tasks on the DSP side are suspended. On the DSP side, the main task creates the tskDSP (see section 3.4) with a priority of 4, the idle task idle_task (if in CPULOAD mode) with a priority of 2, and then ends its execution. Since the tskDSP has the highest priority, the idle task will run only when the tskDSP is suspended. The priority of idle_task was set higher than the priority of 0 of the system idle tasks (from IDL module) to be sure that the idle_task is not influenced by other tasks during its execution. The code for creating these tasks and the callback function of the idle_task are shown in listing 3.10.

[12] http://oprofile.sourceforge.net

Listing 3.10. Tasks for measuring DSP load

```
Void main(Int argc, Char *argv[])
{
  TSK_Handle tskDSPTask;
  TSK_Handle myIdle;

  /* Initialize DSP/BIOS LINK. */
  DSPLINK_init () ;
  ...
  /* Creating task for TSKDSP application */
  tskDSPTask = TSK_create(tskDSP, NULL, 0);
  TSK_setpri(tskDSPTask,4);

  #ifdef CPULOAD
  myIdle = TSK_create(idle_task, NULL, 0);
  TSK_setpri(myIdle, 2);
  #endif
}

static Int tskDSP()
{
  /* Create Phase */
  status = TSKDSP_create (&info);

  /* Execute Phase */
  if (status == SYS_OK) {
    #ifdef CPULOAD
    CLK_start();
    startTime = CLK_gethtime();
    #endif
    status = TSKDSP_execute (info);
    if (status != SYS_OK) {
    SET_FAILURE_REASON(status);
    }
  }
  #ifdef CPULOAD
  endTime = CLK_gethtime();
  *totalPtr = endTime - startTime;
  HAL_cacheWbInv ( totalPtr, intSize) ;
  *idlePtr  = idle;
  HAL_cacheWbInv ( idlePtr, intSize) ;
  CLK_stop();
  #endif

  /* Delete Phase */
  status = TSKDSP_delete (info);
  if (status != SYS_OK)
    SET_FAILURE_REASON(status);
  return status ;
}

static Int idle_task () {
  Uint32 t1 = 0;
  while(1) {
    TSK_disable();
    t1 = CLK_gethtime();
    idle += CLK_gethtime() - t1;
    TSK_enable();
  }
}
```

The percentage of the DSP CPU usage was computed with the following formula:

$$DSP_{workload} = 1 - \frac{t_{idle}}{t_{total}}$$

The total time is measured starting from line 30 of listing 3.10, before the starting of `TSKDSP_execute()` (section 3.4), the ending time is measured after its execution. These values are sent to the GPP to be showed as output by using DSPLink. Functions `TSK_disable()` and `TSK_enable()` in the `idle_task` deserve particular attention. The former function disables the DSP/BIOS task scheduler so that the current task continues its execution even if a higher priority task becomes ready to run. However if a HWI occurs, then the task is preempted. The latter function enables again the task scheduler. These functions are needed in order to avoid the situation in which a higher priority tasks is ready to run after `idle_task` has executed the line 58 and before the execution of the line 59. In this case, the execution time of the higher priority task will be included in the idle time measurement.

The method shown above is very simple, and has the disadvantage that in a complex operating system, the programmer cannot be sure that no other processes influence the execution of the idle task. Fortunately in our case, the DSP/BIOS is a simple BIOS and few tasks are executing and the tasks that are running are well known so this generic disadvantage is not relevant to us. Nonetheless, for testing purposes a further idle task with a priority of 1, i.e. lower than the `idle_task` was run. We can use this to detect if there is a malfunctioning in our thinking, because we expect that this task will never be run during the time our priority 4 and priority 2 tasks are executing, hence we can detect a problem if this additional idle task executes for a number of cycles different than zero.

3.7 Floating point operations

The last objective of this thesis project is to compare the performance of the two solutions (GPP+DSP and GPP+NEON) when executing floating point operations. Such an analysis is important for the porting of SDRs to the OMAP3530 platform since a considerable number of floating point operations are typically performed by SDRs per unit time. Hence, the FIR filter software was modified. The samples of the original input file were converted from 16-bit signed integers to 32-bit floating point to create a new input file. Consequently, the filter coefficients were also modified by computing a new set of coefficients using MATLAB's FDATool.

3.7.1 Floating point on the GPP

On the GPP side, the floating point operations are executed by the NEON coprocessor. Since the NEON coprocessor supports floating point operations (it has two floating point pipelines and a VFPLite unit, see section 3.5), the changes to the code of the FIR filter are straightforward. The main FIR filter function is

the same as shown in listing 3.5, with the only difference that the pointer types are `float` instead of `short`, thus the function prototype becomes:

```
Void FIR_core(float* __restrict longInput, float* __restrict dataOut,
float  coeff[], int halfSize);
```

The code was compiled with the command:

```
arm-angstrom-linux-gnueabi-gcc -O3  -Wall -c  -ffast-math
-fsingle-precision-constant -march=armv7-a -mtune=cortex-a8 -mfpu=neon
-ftree-vectorize -mfloat-abi=softfp -ftree-vectorizer-verbose=15
```

The cross compiler is able to generate vectorized floating point code. The options used for the vectorization are the same as given in section 3.5.1. Additionally, two more options were used:

-ffast-math : this option enables additional optimizations for floating point operations. This can lead to violations of the IEEE standard 754 or ISO C specifications for math functions. The GCC documentation explicitly states that this option must not be turned on when any level of optimization (`-O`) is active since incorrect results according to the IEEE floating point standard can occur. Nevertheless, the cross compiler is not able to vectorize the code without this option; and

-fsingle-precision-constant : since NEON supports only single-precision floating point, this option prevents the compiler from utilizing double precision operations and hence avoids the conversion of the data to double-precision.

By using the tool `arm-angstrom-linux-gnueabi-objdump`, the assembly code produced by the compiler was retrieved. This shows that the cross compiler has vectorized the floating point code by using the instructions: `vmul.f32`, `vadd.f32`, and `vmov.32`. This assembly code is shown in appendix D.

The option `-mfloat-abi` deserves a special attention. This GCC compiler option controls both the ABI and whether floating point operations should be used [39]. It can have three different values:

soft the floating point operations are emulated in software. The compiler will not produce any Floating Point Unit (FPU) instruction, hence the option `-mfpu`, indicating the target hardware floating point unit, is ignored. Integer registers are used to pass `float` arguments to the emulation routines.

softfp the floating point operations are done in hardware by using the `soft` ABI. However, the compiler can generate `soft` instructions to improve performance depending on the target FPU. The main drawback of this approach is that the floating point arguments are passed through integer registers. The copying of data from integer to floating point registers introduces a stall in the pipeline. This slowdown notably penalizes the performance (as shown in section 4.5).

hard the floating point operations are done in hardware. The `-mfpu` option is mandatory and floating point arguments are directly passed in floating point registers.

Although the best option in term of performance is `hard`, during this project the option `softfp` was used since at the time of writing, it is the only one supported by the Ångström compiler toolchain.

3.7.2 Floating point on the DSP

The TMS320C64x+ DSP (2.4.2) does not have a floating point unit. In order to perform floating point operations software emulation must be used. TI's IQmath Library for C64x+ ([21]) permits to the programmers to port of floating point algorithms to fixed point code. Furthermore, optimized routines are available in the library for common operations to facilitate the programmer producing high-performance code.

Listing 3.11 shows how the code in listing 3.4 was modifying for use with IQMath libraries.

Listing 3.11. Code of FIR filter using IQMath library

```
1  ...
2  float * history;
3  float * first_buffer;
4  float * complete_buffer;
5  _iq * IQ_coeffs;
6  _iq * IQ_first_buffer;
7  _iq * IQ_complete_buffer;
8
9  ...
10
11 filter_out = MEM_calloc (DSPLINK_SEGID, halfSize * sizeof (float), ↩
       DSPLINK_BUF_ALIGN) ;
12 history = MEM_calloc (DSPLINK_SEGID, (FILTER_ORDER) * sizeof (float), ↩
       DSPLINK_BUF_ALIGN) ;
13 complete_buffer = MEM_calloc (DSPLINK_SEGID, (halfSize + FILTER_ORDER) * sizeof ↩
       (float), DSPLINK_BUF_ALIGN) ;
14 IQ_complete_buffer = (_iq *) MEM_calloc (DSPLINK_SEGID, (halfSize + FILTER_ORDER↩
       ) * sizeof (_iq), DSPLINK_BUF_ALIGN) ;
15 first_buffer = MEM_calloc (DSPLINK_SEGID, halfSize * sizeof (float), ↩
       DSPLINK_BUF_ALIGN) ;
16 IQ_first_buffer = (_iq *) MEM_calloc(DSPLINK_SEGID, halfSize * sizeof(_iq), ↩
       DSPLINK_BUF_ALIGN);
17
18 //Convert coefficients to IQ type
19 IQ_coeffs = (_iq *) MEM_calloc(DSPLINK_SEGID, NUM_TAPS * sizeof(_iq), ↩
       DSPLINK_BUF_ALIGN);
20 for (i=0; i< NUM_TAPS; i++)
21   IQ_coeffs[i] = _FtoIQ(coeff[i]);
22
23 ...
24
25 if (first_chunk==0) {
26   memcpy(complete_buffer, history, FILTER_ORDER * sizeof(float));
27   memcpy(&complete_buffer[FILTER_ORDER], buffer_input, bufferSize);
```

```
28
29    for (i=0; i<(halfSize + FILTER_ORDER); i++)
30      IQ_complete_buffer[i] = _FtoIQ(complete_buffer[i]);
31    for (j = 0; j < halfSize ; j++) {
32      sum = 0;
33      for (i = 0; i < NUM_TAPS; i++)
34        sum = sum + _IQmpy(IQ_complete_buffer[i + j], IQ_coeffs[i]);
35      filter_out[j] = _IQtoF(sum);
36    }
37    memcpy(history, &complete_buffer[halfSize], FILTER_ORDER * sizeof(float));
38  }
39
40  ...
```

The `_iq` type represents a 32-bit fixed point number in *q format*. In order to convert a floating point array of data to fixed point, the function `_FtoIQ()` may be used. The function `_IQtoF()` performs the inverse conversion[13]. In the line 34, the multiply-accumulate operation is implemented by means of the function `_IQmpy()` in order to perform fixed point multiplication and addition.

The IQMath library provides two types of implementations of its functions: a C implementation and an assembly one by using inline functions. In order to use the former implementation, the header file `IQmath.h` must be included in the application. In order to use the latter one, the header file to be included is `IQmath_inline.h`. According to Texas Instruments' benchmarks[14], an application using the inline assembly implementation is typically about 10 times faster than one using the implementation in C.

[13]Note that because of the limited resolution of the fixed point representation converting a floating point value to a fixed point value and converting back may not result in the original floating point value.

[14]The benchmark program that they used consisted of running a FIR filter of 10 taps with 100 outputs. The result is presented in the `benchmark.doc` file contained in the program installation directory which TI provides.

Chapter 4

Analysis and Results

This chapter collects, presents, and analyzes results from testing the system performance for the configurations and with the tools explained in chapter 3.

Section 4.1 analyses and compares the results of the performance of the GPP+DSP solution as a function of the IPC module that was utilized. The performance of the GPP+NEON configuration is analyzed in section 4.2. Section 4.4 compares the two approaches above with the additional optimizations explained in the section 4.3. The floating point analysis is reported in section 4.5, while the final analysis results are given in section 4.6.

This chapter includes a number of graphs concerning the most important results. Additional graphs and data are included in appendices B, E, and F.

4.1 DSPLink analysis

The first configuration of the system to be analyzed is the GPP+DSP. The objective of this analysis is to find the best DSPLink module in terms of the performance metrics which were described in section 3.6. These performance parameters are: **execution time**, the **GPP load**, and the **DSP load**. The analysis of these parameters has been performed as functions of a *variable chunk size* and on a *variable filter complexity* (hence a variable number of filter coefficients).

The **chunk size** has the values 2^i where $i \in [7, 14]$. The main reason for this choice of values is related to the requirement for *data alignment*. As already stated in section 3.4.1.2, buffers of data exchanged through DSPLink, must have their size aligned with the size of the DSP L1 cache word. Since the minimum size of the DSP L1 cache word is 128 bytes, the minimum chunk size must be at least 128 bytes (2^7) and all other sizes must be a multiple of this value. The highest value is 16 KB in order to avoid the saturation of the DSPLink shared memory which is limited. It is important to note that multiple buffers of the chunk size may need to be allocated in the shared memory. According to the application, several buffers of the specified size

and their associated synchronization structures need to be allocated in the shared memory. Moreover, the chunk size should suit the concept of *data streaming*; hence values greater than 16 KB, depending on the data sample's type, can result in a too coarse-grained streaming (i.e., with too high a delay for the delivery of the media stream).

With respect to the **FIR filter coefficients**, the number of taps is 2^i with $i \in [4, 9]$. The reason for this choice is that 16 taps where simple and sufficient to achieve good results in filtering the input file. On the other hand, the maximum number (as a power of 2) of filter taps that Matlab's FDATool can generate is 512. Hence another tool would need to be used to generate more than 512 (2^9) filter coefficients.

4.1.1 Execution time analysis

Before starting our analysis of the collected data, some clarifications must be made regarding the arguments passed to test software. First of all, a preliminary analysis of the *polling time* of the MPCS software was made. The purpose was to discover the polling time that leads to the best software performance. For this purpose the MPCS software was executed in an average case, with a chunk size of 1024 bytes and 128 taps. This test was run 5 times and the average values are shown in figure 4.1. The best execution time was obtained with a polling time of 100 μs. This value as a standard deviation of 0.34 μs (the minimum value is 1.79614 ms, the maximum value is 1.79694 ms). Hence, the polling time of 100 μs was used for all the other experiments involving polling. However, as once case see from these results there is really very little difference between the polling times of 10 μs and 10 ms. In order to explain this behavior, the chunk's round-trip time must be considered (see later in this section). For this test software and in these conditions, the chunk's average time was 4.29 ms. This means that the data are available after this amount of time. So the software performance remains the same for polling times less than 10 ms since data is retrieved with a low delay. In contrast, if the polling time is greater than 10 ms, a delay will occur between the time when data are ready and the polling time, thus leading to performance degradation.

Additionally, the MPLIST and RINGIO software were designed to achieve a certain degree of parallelism (see sections 3.4.6 and 3.4.7). In order to compare them with the other software, they were executed with a parallelism level (PL) equal to 1. This means that for the MPLIST software, the maximum number of elements in the list was set to 1, while for the RINGIO software, the dimension of the circular buffer was equal to the chunk size. Additionally, an additional analysis comparing the performance of these two modules has been done with a parallelism level of 4 (see later in this section).

As explained in section 3.6.1, we are interested in two times: execution and chunk round-trip time. These can be expressed as:

$$t_{execution} = 2t_{memory} + t_{roundtrip}$$

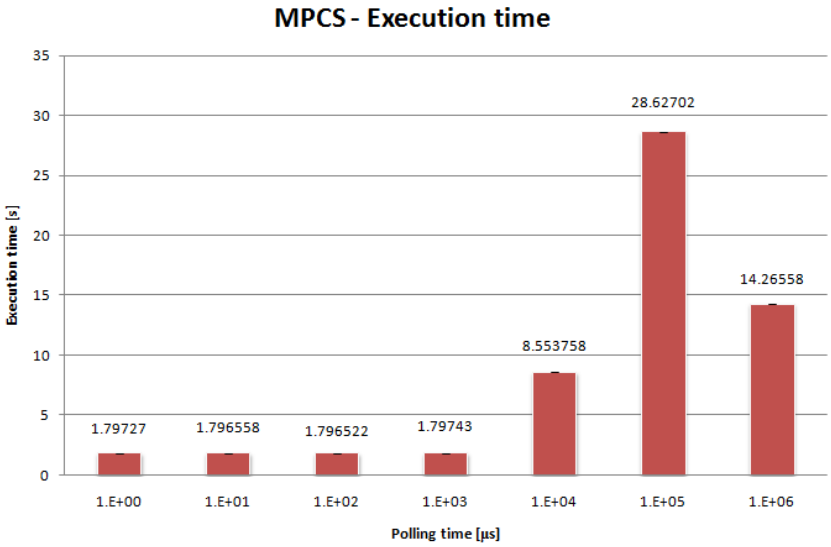

Figure 4.1. Execution time of MPCS as a function of the polling time

$$t_{roundtrip} = 2t_{transmission} + t_{dsp}$$

The graph of the execution times of each type of DSPLink module is shown in figure B.2. Figure 4.2 shows the average execution times of all these different modules. The vertical axis represents the average value of the execution times of all filter versions for a fixed chunk size. From this graph we can understand the performance of each module as a function of the chunk size and at the same time easily see which module optimizes the execution time. According to the graph, the best IPC mechanism in terms of execution time for all the values of the chunk size is MPCS. The reason for this is that this module exploites direct access to the shared memory and utilizes *polling*. The second best module is PROC, as it also exploits direct access to the shared memory but it uses *interrupts* instead of polling (thus there will be an added delay before the GPP knows that the processing of a chunk is finished). At the same time, MSGQ offers similar performance to both MPCS and PROC. The worse IPC mechanism is CHNL, which exhibits very poor performance for small chunk sizes. This is due to the issue-reclaim model that does not provide good performance since the process is suspended twice every data transmission (once while waiting for an available buffer, and once while waiting for the other end to provide or elaborate data). Furthermore, this communication model has a high communication overhead, particularly evident for small chunk sizes (hence a higher number of transfers). Additionally, the performances of MPLIST and RINGIO are worse than the first three modules, but better than CHNL. Furthermore, we can see that the execution time is similar for chunks with a size greater than 4 KB. For these values the $t_{transmission}$ time is negligible with respect to to the processing time t_{dsp}. For an audio signal at 44,100 samples per second, 4096 bytes represents

less than 47 ms of audio. In graphs referred to execution time error bars are shown. The error bars represent the precision of measurement done on the graph. In order to define the accuracy of the measurements of the execution times, a series of 5 measurements was done for each chunk length. The highest standard deviation was considered resulting in a precision of ±0.28%. This percentage was applied to all measurements concerning the execution times. Error bars in figure 4.2 represent the precision for the average of 6 values. The precision is so computed as the accuracy value of the single measurement (±0.28%) multiplied by 6, resulting in an accuracy of ±1.69%.

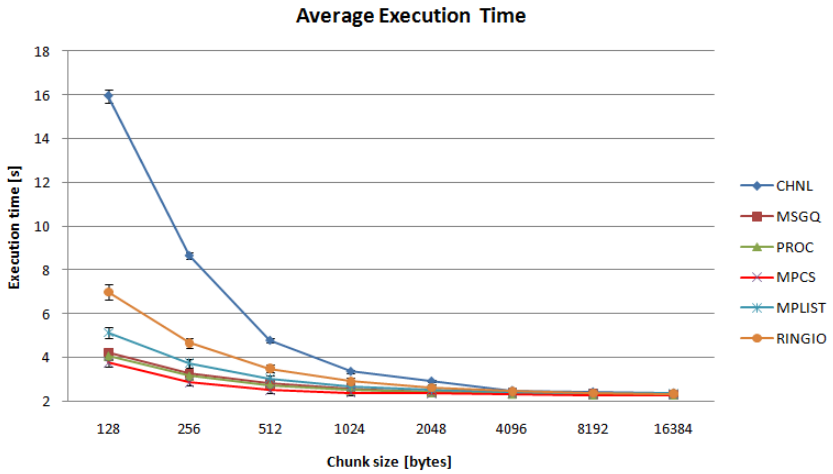

Figure 4.2. Average execution time of the different DSPLink modules for different chunk sizes

The graph shown in figure 4.3 is a complement to the previous figure. It shows the average execution time for all chunk sizes and numbers of taps. This average value has been considered to offer a simple overall comparison of the different IPC mechanisms. The graph confirms the results shown in figure 4.2, and highlights how there are two distinct classes of modules with respect to the execution time. The first group, made up of MPCS, PROC and MSGQ, has the best performance; while the second one, made up of MPLIST, RINGIO, and CHNL, should not be used if minimum execution time is the primary target of a system.

With respect to the total round-trip time, the graphs for all modules are shown in figure B.1. These figures show the total round-trip time computed as the sum of the chunk round-trip times of all sent chunks. Depending on the chunk size, a variable number of iterations are needed in order to send all the data. If we call this value *iterNumber*, then the average chunk round-trip time can be defined as:

$$t_{roundtrip\_avg} = \frac{t_{roundtrip}}{iterNumber}$$

This value represents how much time is spent by a chunk of data going from the GPP to the DSP and vice versa. Figure 4.4 shows, for each chunk size, the average

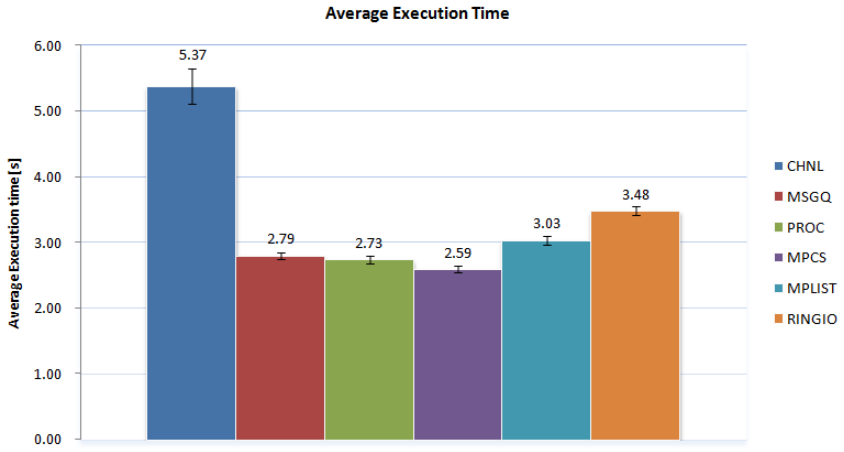

Figure 4.3. Average execution time of DSPLink modules

value of $t_{roundtrip\_avg}$ for all filter versions; while figure B.3 presents the value of $t_{roundtrip\_avg}$ for all the different DSPLink modules. From figure B.3, it is clear

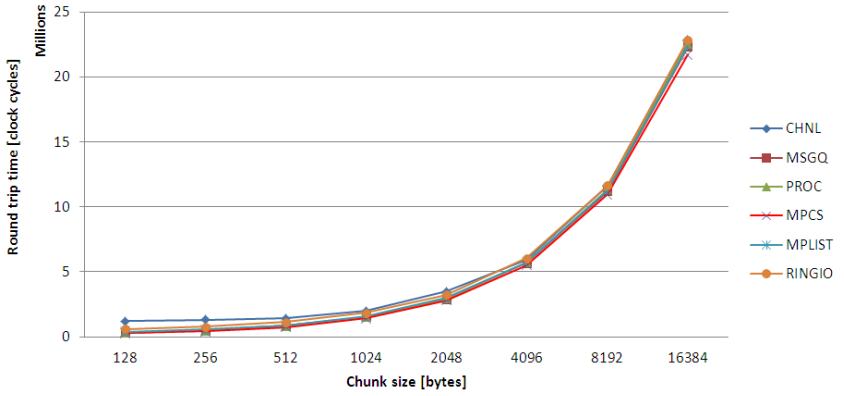

Figure 4.4. Average round-trip time of DSPLink modules for various chunk sizes

that the performance trend is similar for all modules, with the exception of the CHNL module. In general the round-trip time increases exponentially as the chunk size increases. This is due to the $t_{transmission}$ component which grows linearly as a function of the chunk size, and because t_{dsp} increases quadratically with the number of taps (since the FIR filter algorithm is $O(n^2)$, with n as the number of taps).

A better analysis can be carried out by using the graph shown in figure 4.5. As for figure 4.3, the value on the vertical axis represents the average of $t_{roundtrip\_avg}$ values for all chunk sizes and number of taps. From this graph we can see that MPCS has the smallest average chunk round-trip time and its performance is definitely better than the others. However, it is surprising that the MSGQ performance is better than the PROC one in terms of round-trip time. This means that the

interrupt mechanism of MSGQ is more efficient than the PROC one. Nevertheless, the execution time of PROC (as obtained from the previous analysis), is lower than the one of MSGQ. This is due to the fact that t_{memory} of MSGQ is greater than for PROC since PROC does *not* need to copy data from the user memory space to the shared memory since direct memory accesses are performed. Furthermore, more complex synchronization and data structures are needed by MSGQ, which decreases its performance. As seen in this graph, the performance of MPLIST in terms of round-trip time is comparable to the PROC one. In terms of execution times, PROC is slightly better than MPLIST, as was shown in figure 4.3. The complex communication protocols of CNHL and RINGIO limit their performance, so that their bad performance in terms of execution times is confirmed by this analysis of average round-trip times.

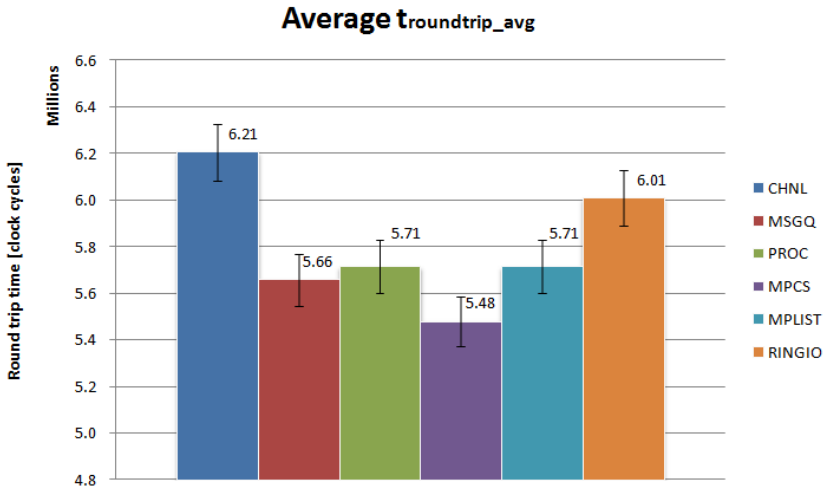

Figure 4.5. Average round-trip time of each of the DSPLink modules

As already mentioned, a comparison between RINGIO and MPLIST modules when exploiting a parallelism level of 4 has been done. The degree of parallelism has been limited by the available shared memory as the structures could only hold 16 KB of data. As a result, the maximum chunk size that I have analyzed was 4 KB (hence 16 KB of structure in this case and in the earlier analysis). This analysis aimed to determine if MPLIST is better than RINGIO when a level of parallelism greater than one is used. Results in terms of both the execution and average chunk round-trip time are shown in figure 4.6. From the graph, it is clear that MPLIST has a better performance than RINGIO in the case of both a parallelism of 1 and 4.

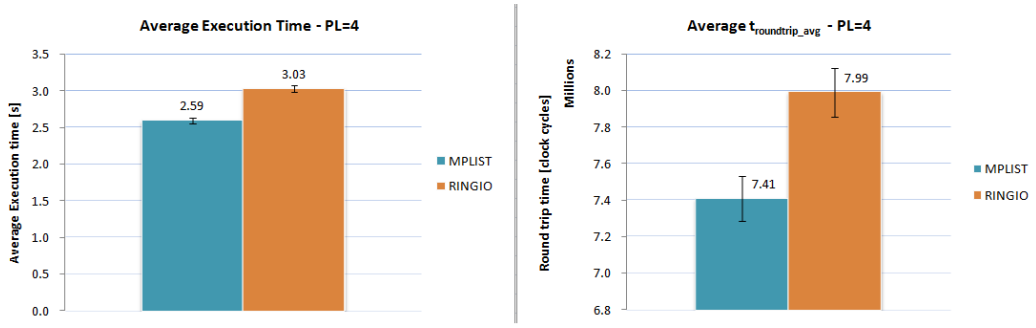

Figure 4.6. Performance of MPLIST and RINGIO with a parallelism level of 4

4.1.2 GPP load analysis

The GPP load for all the DSPLink modules are showed in the graph in figure B.4. From these graphs is possible to analyze the GPP workload as a function of the chunk size and the number of FIR filter taps. In order to compare these different modules, the graph in figure 4.7 can be used. This graph represents the average value of the GPP load for all the different modules. The vertical axis represents the average values of the GPP load for all chunk sizes; on the horizontal axis the filter order is represented. The most evident result is that while the GPP load of MPCS increases with the filter complexity, the GPP load of all other modules decreases. The reason for this is the usage of the *polling*. Since in the MPCS testing software the GPP does a busy form of waiting, the process waiting for data is suspended and restored frequently. On one hand, this waiting process is executed by the processor so that this processor performs some unnecessary work when it pools and the results are not ready. Additionally, a considerable number of unnecessary context switches occur - increasing the overhead on the processor. This explains why the trend of the MPCS's GPP load is opposite of the other IPC methods. The only way for the GPP to be suspended for a longer time than the polling time, is to wait on the MPCS object. This happens if the DSP is in the protected and shared region of code. For a lower number of coefficients, the probability that the DSP has acquired the shared region is higher than in the case of higher number of taps - since t_{dsp} is shorter[1]. Since t_{dsp} is shorter, the DSP will spend more time in the shared region to check if data were updated. This leads to a higher number of conflicts that causes the GPP to be suspended for a longer time. In the case of larger number of taps, the DSP spends a long time computing on a given chunk of data. During this time, the GPP always gets access to the shared region without being blocked; hence, the GPP load increases. For all the other modules that have their communication protocols based on interrupts, the smaller the number of taps, the smaller is the process waiting time, hence higher is the processor load. Moreover, the smaller the chunk size, the greater the number of iterations needed to send data. Hence, the process is woken

[1]The DSP makes its data computation outside the shared region protected by the MPCS object.

up more frequently. Regarding the other modules, the best module in terms of GPP load is MSGQ as it shows an efficient interrupting and synchronization mechanism. The performance of PROC is not very different from the MSGQ, but, starting from MPLIST, different performance is shown. Once again the CHNL module offers poor performance, in this case in terms of GPP workload. It is important to note the influence of the measurement tool `top` on these measurements. As stated in section 3.6.2, the GPP workload due to `top` is about 16%, hence no workload is 100% (as 16% of the GPP is being consumed by the `top` process).

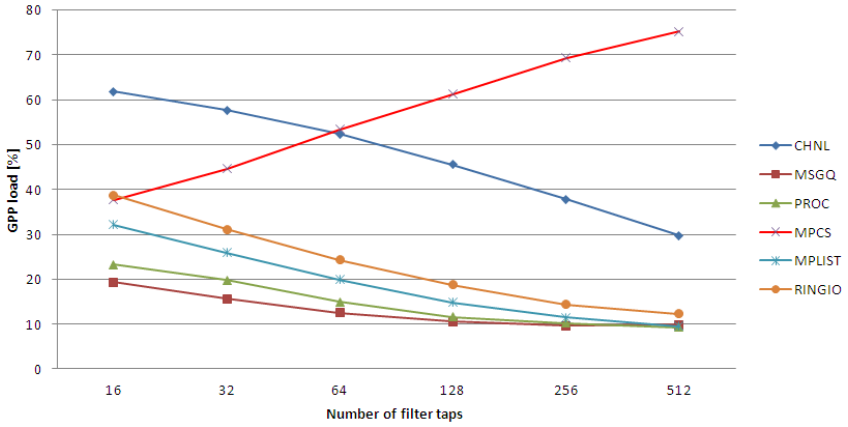

Figure 4.7. GPP workload of DSPLink modules

Figure 4.8 compares all the different DSPLink modules based on the GPP load computed as the average for all values of chunk size and number of filter taps. The results presented in the previous graph are highlighted in this column graph. The good performances of MSGQ and PROC are confirmed, while the performance of CHNL and MPCS are clearly quite poor.

4.1.3 DSP load analysis

Figure 4.9 shows the average DSP workload for all chunk sizes as a function of the filter order. From this graph two results are evident. The first result is that the DSP workload slowly increases as the filter complexity grows. This can be explained by the fact that as the data computation grows with the filter complexity, the DSP computation time (t_{dsp}) increases, while the $t_{transmission}$ and t_{memory} remain the same (the vary as a function of the chunk's size). The second result is the clear difference in behavior between MPCS and all other DSPLink modules. The reasons for this behavior of the MPCS module have already been explained in section 4.1.2. The processor load is roughly 100% since the overhead of any measuring tool does not affects the measurements results. Regarding the other protocols, there is not a well-defined difference in performance.

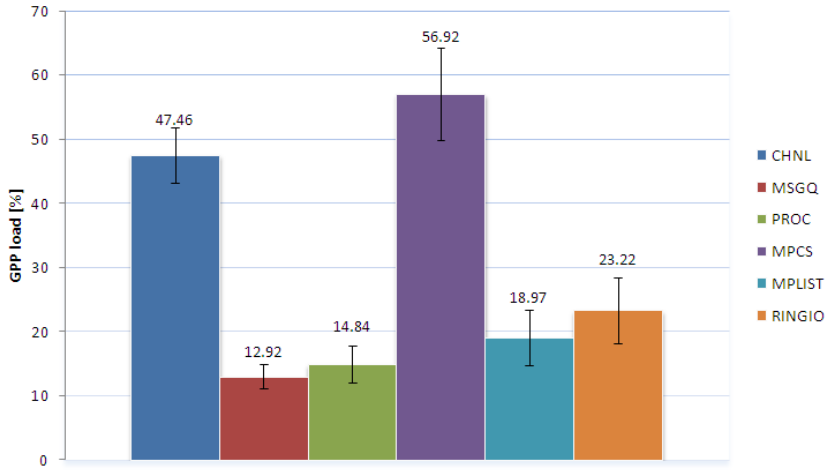

Figure 4.8. GPP workload of the different DSPLink modules

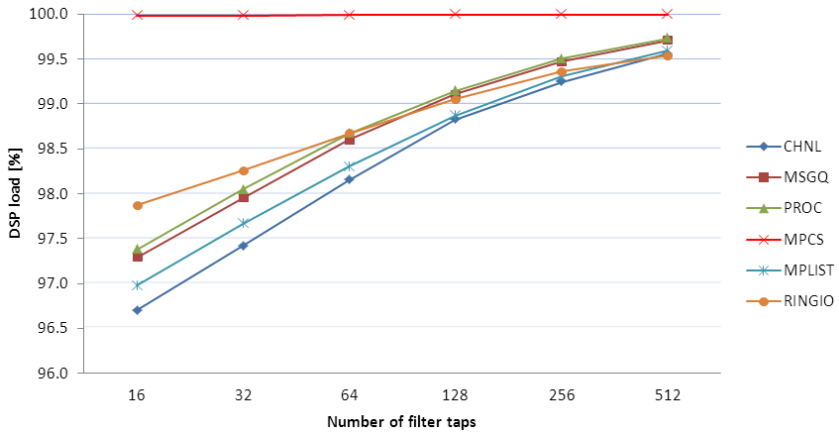

Figure 4.9. DSP workload of the different DSPLink modules

In order to understand which is the best IPC protocol with respect to the DSP load, the graph in figure 4.10 is useful. The values showed by this graph are the average for all chunk sizes and filter orders. From this graph, the bad performance of MPCS is highlighted, while no other module is clearly the best, although the module that offers the best performance in terms of DSP load is CHNL.

4.1.4 Conclusions

The outcome of the above analysis is summarized in this section. The best DSPLink IPC protocol in terms of execution time is MPCS, while PROC and MSGQ also offer good performance. Although MPCS is the best in terms of timing, it results in the worst GPP load. The best communication mechanism, in terms of minimal GPP

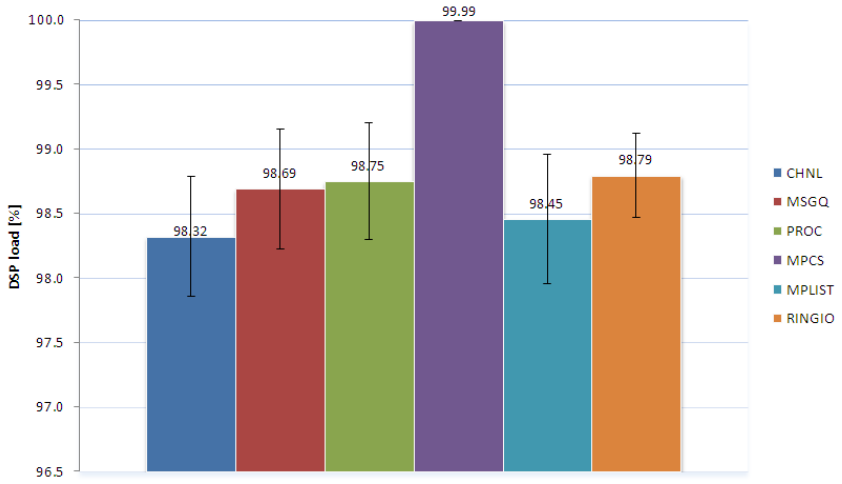

Figure 4.10. DSP workload of the different DSPLink modules

load, is MSGQ with PROC and MPLIST also offering decent performance. With respect to the DSP load, there is not module which is clear best (although PROC is slightly better than others). However, there is one module that is clearly the worst one in terms of DSP load: MPCS.

In conclusion, if the system designer is exclusively interested in timing optimization without concern for other system aspects, the module that should be used is MPCS. Nevertheless, the **MSGQ** module should be utilized to achieve the best system balance. In fact, by using MSGQ, we achieve good timing performance, with the GPP load optimum and the DSP load being average. This leads, on one hand to a lower power consumption since the processors can be put in a standby state while waiting for data and on the other side, the level of parallelism of the system is increased since other tasks can be executed by the GPP while waiting for data from DSP.

4.2 NEON analysis

After having analyzed the performance of the GPP+DSP solution, the next step is to analyze the performance of the system when the NEON coprocessor substitutes for the DSP . This analysis is done in terms of **execution time** and **processor load** (GPP load) using the tools explained in section 3.6. The performances of the three NEON test software versions (shown in section 3.5) have been compared with the performance of the test software running on the GPP only (without either DSP or NEON coprocessor). The results of the analysis in terms of the execution time are shown in figure 4.11.

From the graph the difference between the NEON-based solutions and the

GPP-only solution is clear. On one hand, the difference in performance between the intrinsics-based and assembly-based solutions is really slight. Nevertheless, the performance of the intrinsics-based software is consistently better than the assembly-based one. With respect to the compiler-based version, the performance is more similar to the GPP-only version than to the other two NEON versions. Additionally, the differences among versions are more evident for higher filter orders. In fact, the execution time grows exponentially as the FIR filter order grows. The execution time varies as:

- $k \cdot e^{0.648x}$ for GPP-only version;
- $k \cdot e^{0.629x}$ for compiler-based NEON version; and
- $k \cdot e^{0.416x}$ for intrinsics-based and assembly-based NEON versions

where x is the number of filter taps and k is a constant which differs for each function. From this analysis we can conclude that the compiler could be significantly improved - as there is potentially a factor of 4 to be gained for the case of 512 taps. In order to estimate the accuracy of the measurements shown in figure 4.11, the highest deviation standard for each data series as a percentage of the execution time average value. The deviation standards are ±0.09% for the *GPP_only* software version, ±0.05% for the *NEON_compiler*, ±0.12% for the *NEON_intrinsics*, and ±0.03% for the *NEON_assembly* software version.

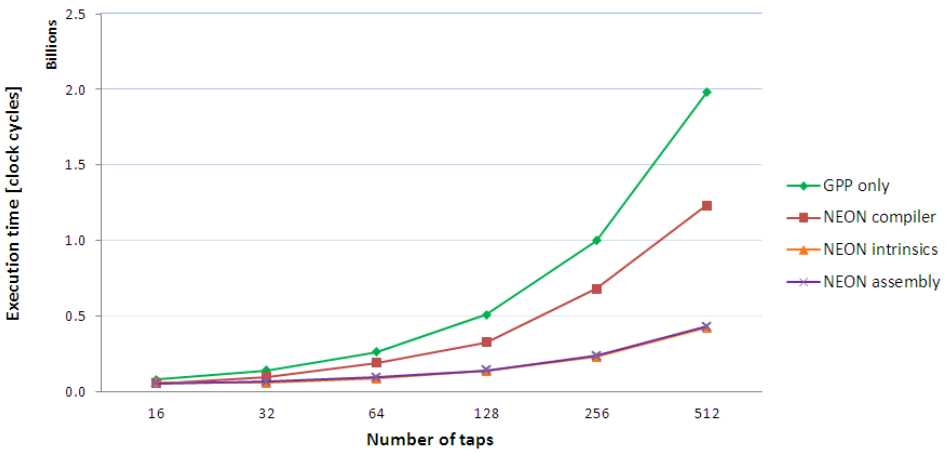

Figure 4.11. NEON execution time as a function of number of taps

The other parameter used in the performance evaluation is the processor load. In order to accurately measure this workload, three series of measurements have been made and the average of this series are considered. As we can see from figure 4.12, the growth of the processor workload is small as a function of the number of filter taps. Performances of the different software versions are similar. Nonetheless, we can notice the better performance of the intrinsics-based version for low-order

filters and the generally worse performance of the GPP-only solution for nearly all different numbers of taps. The average of all the workload values was computed and used to compare their performances, as shown in figure 4.13. This graph confirms the slight difference between the different versions. However, the best version is the compiler-based one, , but it is not clear that the difference is significant.

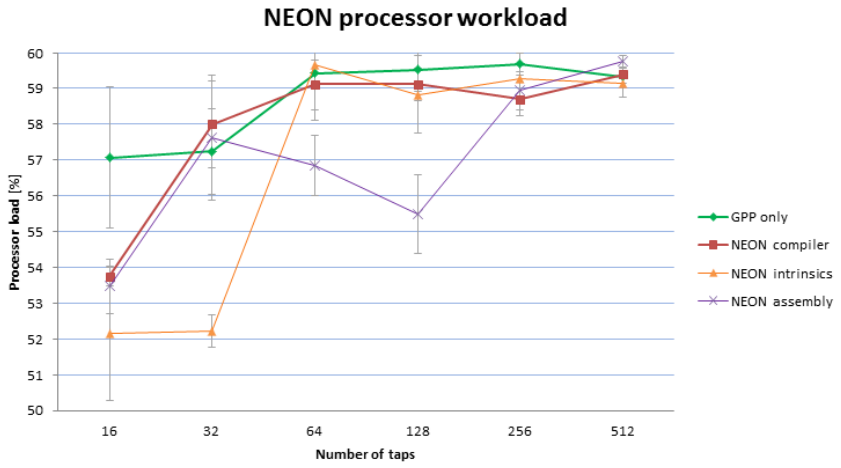

Figure 4.12. NEON processor workload as a function of the number of taps

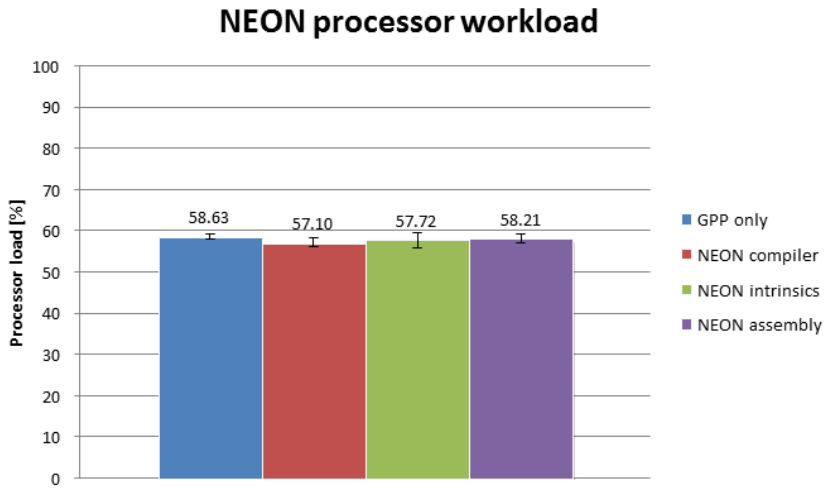

Figure 4.13. NEON processor workload for the different versions of the program

4.2.1 Conclusions

The analysis above has highlighted two important results. The first result is that the NEON coprocessor definitely increases the timing performance of the system in comparison to only using the GPP for computations. The amount of improvement depends on the complexity of the computation, but it shows an exponential growth in the number of taps. The best way of programming the NEON processor is using **intrinsics**. This solution offers both the best timing performance and a good trade-off between programming complexity and speed of programming. If the timing constraints are not very strict, the vectorizing compiler can be used to develop NEON code while offering some improvement in the timing performance. The assembly approach is not relevant as it achieves the same performance as the intrinsics, while introducing greater programming complexity and requiring a long learning curve.

The second result is that the NEON coprocessor does not significantly offload the processor. This is reasonable since NEON is a coprocessor and not an external processor. This is due to the same resources (memory, buses, etc.) being shared between the GPP and the NEON. Furthermore, we have to remember that the GPP is not suspended during the execution of NEON instructions (see section 3.5). Additionally, the test software version that improves the processor workload most is the compiler-based one. Nevertheless, the best solution of developing NEON code remains using the intrinsics, since the processor workload performance is very similar to the best achieved by the compiler-based version.

4.3 Code optimizations

This section explains the optimization made to the test software of the GPP+DSP solution in order to further improve the timing performance. The DSPLink analysis in the section 4.1, was carried without compiler optimizations. This was done so that the different modules performance would not be influenced by the compiler optimizations. This choice was suitable for an initial comparison of modules' performance. The next step of our analysis is to compare the two solutions taken into consideration with possible optimizations. Hence, an optimized version of the GPP+DSP solution code should be compared to the already optimized code for the NEON-based solution. Since the target of this optimization is improved timing performance, the best DSPLink module in terms of timing (MPCS) was further examined. The test software based on this IPC protocol, was optimized by using compiler optimization options and by changing some code to optimize memory transfer operations.

4.3.1 Compiler optimization

The first step in optimizing the software was to examine the compiler optimization options. The software for the GPP+DSP solution is distributed over the GPP and

DSP and so two different compilers must be used for the target software (one for each side). The software for the GPP side was compiled using the cross-compiler `arm-angstrom-linux-gnueabi-gcc` based on GCC 4.3.3. The compiler option `-O3` was used. This option activates some optimizations in the compiler in order to improve the resulting code's timing performance. However, a side effect of this optimization is that both the compiling time and the code size may be increased. This option is designed to optimize the code in terms of speed of execution while enabling only optimizations that are valid for all standards compliant programs.

The compiler used for the DSP software was `cl64`, the C compiler from the Code Generation Tool. This compiler is able to produce good performance by using optimization techniques. In order to emit optimized code, the programmer can specify the following options [40]:

-o1 minimal high-level optimizations are performed.

-o2 a low function-level optimizations are performed.

-o3 a low full optimization is performed.

If no optimization option is specified, the `-o0` level is implicitly adopted. These different options optimize the performance in terms of reducing execution time, but increase the code size. If the code size is an issue, the option `-ms` can be used. In order to achieve the maximum performance, the option `-g` must be avoided. This option enables full symbol debug, but limits code reordering and other optimizations. This causes a performance degradation ranging from 30% to 50% according to [40, page 8]. In this thesis project, the code was compiled with the options `-o3 -mv6400+` which applies optimizations target to the TI C64x+ DSP. Among the optimizations applied, two are especially important. The first one is *software pipelining*. Pipelining is disabled by default unless an optimization level greater than zero is specified. Pipelining can substantially increase the software performance due to the VLIW architecture implemented in the processor. Another optimization done by the compiler for the C64x+ architecture is the usage of a *loop buffer*. This buffer allows instructions in a loop to be fetched the first time the loop is invoked, rather than every iteration of loop. This technique saves power as there are fewer accesses to the instruction memory. Under certain conditions, timing performance is improved (see [40] for further details). In order to speed up the code, the `MUST_ITERATE` compiler directive was used where appropriate. This compiler directive is a pragma that informs the compile the minimum number of times the loop will be executed, the maximum number of times it we be execute, and the multiples of the minimum number of times that the loop will be executed.

4.3.2 Memory transfer optimizations

In order to transfer data from the GPP's memory space to the DSPLink shared memory, the `memcpy` function was exploited on both the GPP and DSP sides. As

discussed in [41], it is considered to be a time-consuming method for copying data. In fact, this function is blocking and does not return until the copying process is completed. So the processor's pipeline can be idle for many cycles. Fortunately, an optimized version of `memcpy` is provided by Texas Instruments to be used on the DSP , thus the performance degradation is limited. On the GPP side an alternative method should be implemented. From the analysis done in [42], the fastest way for copying data on a Cortex-A8 is via the NEON, in particular, by exploiting the NEON coprocessor's data preloading. *Data preloading* is a technique to improve data transfer performance by intelligently exploiting the memory hierarchy. It consists of loading data into the L2 cache in advance by exploiting the principle of locality. When a data miss does happen, data must not be retrieved from the main memory, but rather fetched directly from the L2 cache. In this way, the long access time to the main memory is hidden. The study in [42] shows that the performance achievable by only using NEON is the same as the `memcpy` one; in contrast, if NEON and the preloading are utilized, then the performance is increased by 49% with respect to `memcpy` implemented by the ARM processor alone.

A custom assembly function was implemented, called `custom_memcpy`. The code of this function is shown in listing 4.1.

Listing 4.1. Custom memcpy function

```
1     .arch armv7-a
2     .fpu neon
3     .text
4     .align  2
5     .global custom_memcpy
6     .type custom_memcpy, %function
7
8 custom_memcpy:
9     push {r4}
10    eor r3, r3, r3      @ initialize the index to 0
11 .Loop:
12    pld [r1, #0x100]    @ r1 = src address
13    vldm r1!, {d0-d7}       @ load into d0-d7 64-byte data (8 bytes per D ↩
          register)
14    vstm r0!, {d0-d7}   @ r0 = dst address. The previous loaded 64-byte data is ↩
          stored
15    add r3, r3, #64     @ update the index
16    cmp r3, r2
17    blt .Loop
18
19    #case length not multiple of 64 -> word copy
20    sub r2, r3, r2      @ check if the remainder is 0
21    cmp r2, #0
22    beq .Lmultiple
23    eor r3, r3, r3      @ initialize the index to 0
24 .Loop_word:
25    pld [r1, #0x100]
26    ldr r4, [r1], #4
27    str r4, [r0], #4
28    add r3, r3, #4
29    cmp r3, r2
30    blt .Loop_word
31
```

```
32 .Lmultiple:
33
34     pop {r4}
35     mov pc, lr        @ restore program counter
```

This function preloads in the L2 cache 256 bytes of data (four Q registers), then for every loop iteration, 64-bytes of data are read from the source address and moved to the destination address. If the data length is not a multiple of 64-bytes, this is handled by the code in lines 22-32.

Using the NEON coprocessor for copying data leads to another advantage that improves timing. The processor's integer registers are not corrupted (i.e., these registered do not have to be used, hence the values of these registers do not have to be saved before and restored after the copy operation) since the data are transferred via NEON's register file. This improves performance especially in the case of frequent data transfers, since the register restoring overhead is potentially eliminated. Another widely adopted option for copying data, is the use of a Direct Memory Access (DMA) engine. Such a coprocessor offloads the data transfer operations from the GPP. This enables the processor to perform other operation in parallel with the DMA engine copying data. The data are copied by the DMA engine using sub-blocks. The system performance can be further improved by implementing a pipeline. The DMA coprocessor provides a small block of data to the GPP so that the GPP can start its processing. In the meanwhile, other sub-blocks are loaded and passed to the GPP , rather than requiring the GPP to wait for the entire block of data to be loaded.

4.3.3 Performance analysis

The optimizations explained in the previous sections were applied to the test software and results of running tests with this software were collected. In this section only the testing software using the MPCS module is considered. The version of the software without any optimization is called *GPP+DSP v.1*. While the name *GPP+DSP v.2* indicates the version optimized by the compiler. Finally, the version with memory transfer optimizations is called *GPP+DSP v.3*.

Figure 4.14 shows the execution time as a function of chunk size and filter complexity of the final optimized version *GPP+DSP v.3*. This graph clearly shows that for chunk sizes smaller than 2 KB in size, the execution time does not depend on the number of taps. This is due to the fact that the DSP is sufficiently fast in computing that t_{dsp} is negligible, hence the transmission time dominant. For chunk sizes greater than 2 KB, the execution time differs but not significantly.

A comparison of the three versions of the test software is shown in figure 4.15. Firstly, the graph shows a significant gain in performance when optimizing the software. However, we note that the performance of the two optimized versions is similar. This behavior was expected since the DSPLink memory is not cacheable on the GPP side. Hence, data preloading is not applicable. The `custom_memcpy` function exploits only the NEON coprocessor without any preloading. This, as

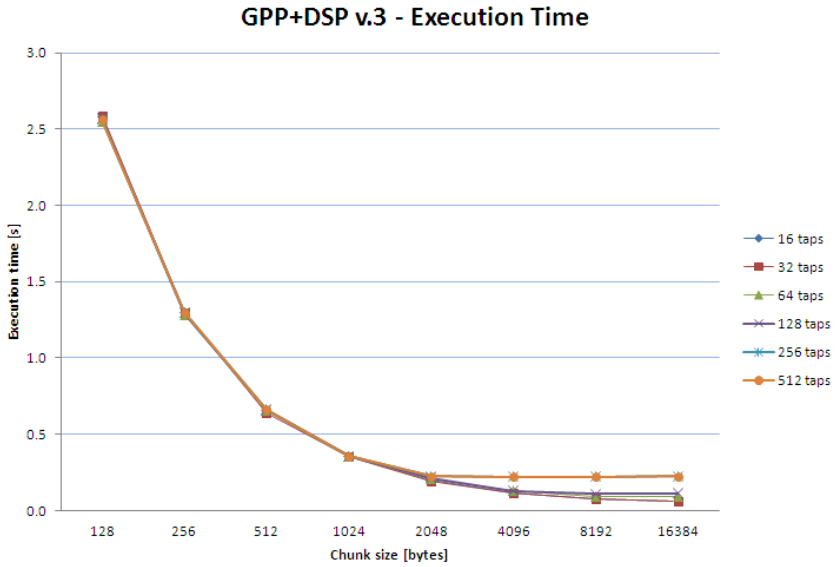

Figure 4.14. GPP+DSP v.3 execution time as a function of the chunk size and number of filter taps

discussed in the previous section, leads to performances comparable to that achieved by using `memcpy`. The usage of the `custom_memcpy` routine should be limited only to the GPP+NEON solution where it offers considerable performance improvement.

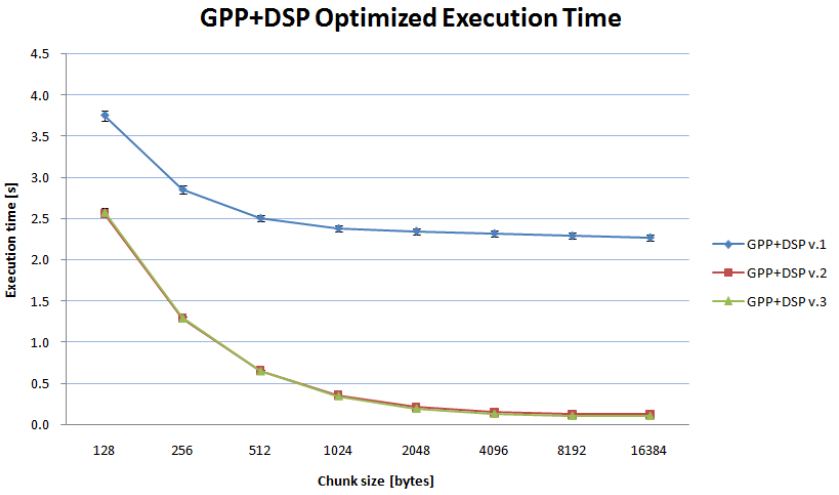

Figure 4.15. GPP+DSP optimized execution time

In order to quantify the improvement in performance achieved by *GPP+DSP v.3*, figure 4.16 is useful. This figure illustrates the average of all execution time

values. From this graph it can be determined that the combination of these optimizations achieve a 73.92% improvement.

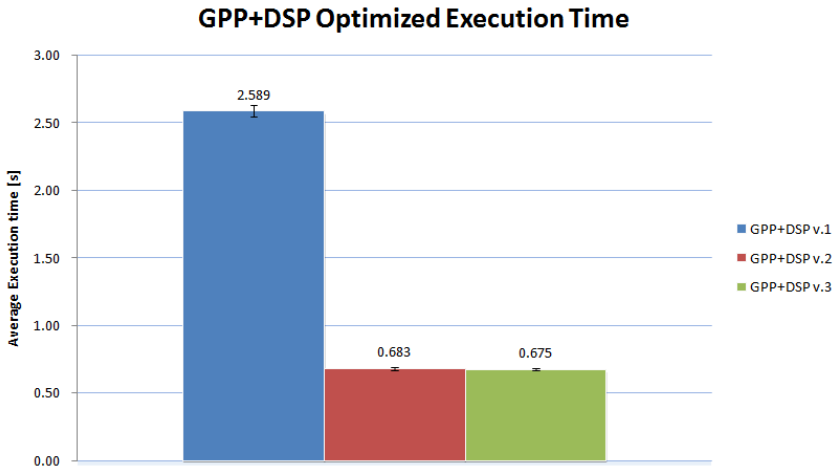

Figure 4.16. GPP+DSP optimized execution time

4.4 Comparison of GPP+DSP and GPP+NEON solutions

In this section, two earlier system configurations are compared in terms of execution time and GPP workload. The best performances achievable by exploiting the two configurations are considered. Hence, during the timing analysis, the performance of the GPP+DSP v.3 testing software (section 4.3.3) is compared to the NEON intrinsics-based version. Regarding the GPP workload analysis, the MSGQ testing software (the best GPP workload according to section 4.1.2) is compared with NEON compiler-based version.

4.4.1 Execution time analysis

The first step is to analyze the performance in terms of the execution time of the two solutions when executing one FIR filter. Figure 4.17 shows the execution time in terms of filter complexity for both solutions. Regarding the DSP-based solution, only the best results (achieved with larger chunk sizes) are showed for the sake of clarity. Before analyzing this graph, we have to take into account the trade-off between the execution speed and the *coarseness* of the streaming. For the GPP+DSP solution, the best performance is achieved with large sized data chunks (the maximum length considered in the analysis was 16 KB). Although the communication overhead is reduced by using larger chunk sizes, a coarse-grained data streaming results. This means that data buffering takes more time and processed data are only ready after a delay that is proportional to the chunk size.

During this thesis work, data streaming performance was considered for the DSP-based solution, while for the NEON-based solution the whole data set was filtered without being split. This means that the best execution time for the NEON-based solution was achieved. If we want to stream data in a similar manner as for the DSP-based solution, the NEON performance will be degraded due to the overhead of splitting data. From figure 4.17 is evident that the performance of the GPP+DSP solution is better than for the GPP+NEON solution for all filter complexities and for chunk sizes larger than 4 KB. Furthermore, the exponential growth of the execution time of the DSP-based solution is much slower than for the NEON-based solution.

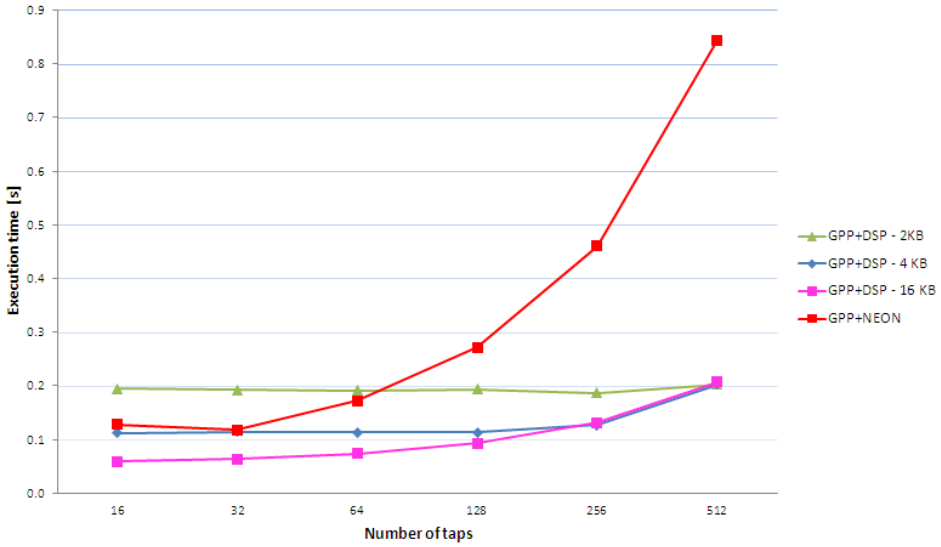

Figure 4.17. DSP versus NEON execution time

The next step is to analyze the performance when data is filtered by a chain of blocks. This analysis was done to simulate data computations in a SDR system when several blocks are processed in a series. In order to simplify the analysis, a chain of 512-tap FIR filters is considered. In the case of DSP-based solution, data was sent from the GPP to the DSP, processed by the chain of filters and sent back to the GPP. Figure 4.18 shows the result of the experiment's results in terms of execution time of different system configurations as a function of the number of blocks in the chain.

4.4.2 GPP load analysis

In this section the results of the GPP workload of the previous sections are merged. Figure 4.19 shows the average GPP load for all filter complexities (and chunk sizes in the case of the DSP-based solution). A big difference in behavior between the two solutions with regard to the GPP load is evident. As we can see, while the DSP

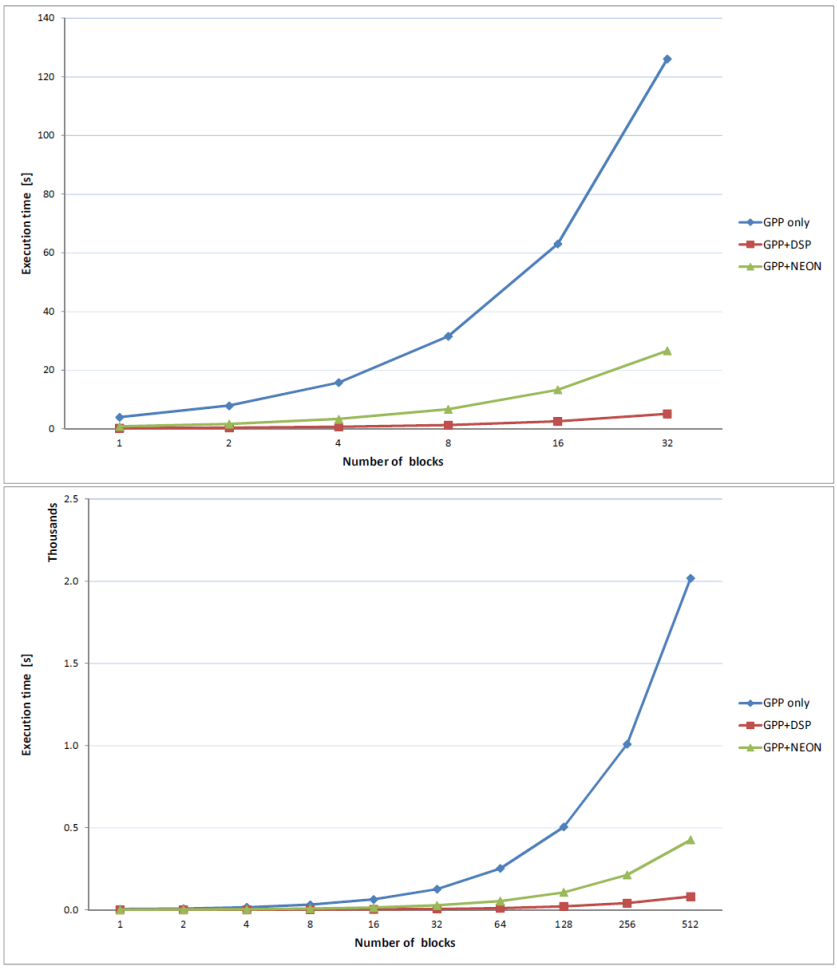

Figure 4.18. Execution time for a chain of blocks

offloads the GPP, the NEON coprocessor does not. The reasons for these behaviors have been discussed in sections 4.1.2 and 4.2.

4.5 Floating point analysis

In the section 3.7, the test software versions for floating point operations were introduced. In this section we analyze their performance with respect to their execution time. As already explained in section 3.7, the test software executed on the NEON coprocessor exploits the hardware floating point unit provided by NEON. On the DSP side, there are no hardware FPUs, but only a fixed point arithmetic and logic unit. Hence, TI's IQMath Library was used to convert floating point numbers into q-format fixed point numbers. The IPC protocol used in this test software is MPCS, as it was the best in terms of execution time performance. There are

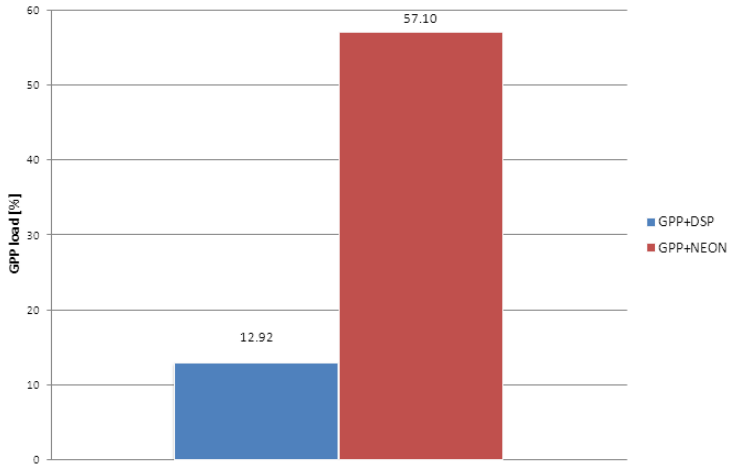

Figure 4.19. DSP versus NEON GPP workload

two possible implementations of the functions provided by IQMath Library: C and assembly implementations. During this analysis, the performance of the NEON software is compared with the performance of two versions of DSP software exploiting both libraries.

The performances of the three solutions are shown in figure 4.20.

From the graph it is clear that the best performance is achieved by using the DSP and the assembly implementation of the IQMath Library functions. The NEON performance is slightly better than the DSP performance when using the C implementation of the IQMath Library. Nevertheless, we must emphasize that:

1. the DSP performs all computations as fixed point computations. A precision loss may happen when using fixed point, in comparison to floating point. In order to reduce this loss, the programmer should carefully think about the data type to be used for q-format numbers. In particular, a preliminary analysis about the number of bits to be reserved for the integer and fractional parts must be done specifically for the application in question. This increases the programming complexity and the risk of obtaining incorrect results.

2. better performance can be achieved by the NEON coprocessor by using a compiler that supports the option `-mfloat-abi=hard`. The tested software was compiled with the option `-mfloat-abi=hard`, the only option supported by the utilized compiler (for further details, see section 3.7.1).

In order to verify the performance improvement achieved by the assembly implementation with respect to the C language implementation, the average value of the execution times for the 512-coefficients FIR filter for all chunk sizes was computed. This value is 17.59 seconds for the C implementation and 1.72 seconds for the assembly implementation. This confirms the TI's benchmark (see section

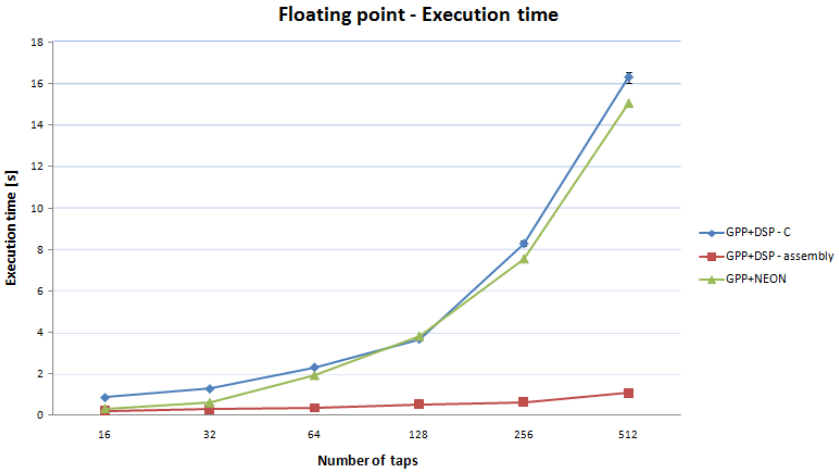

Figure 4.20. Floating point execution time

3.7.2) since simply by including `IQmath_inline.h` header instead of `IQmath.h`, the software executes 10.17 times faster. If we consider all FIR filters, the assembly version is on average 4.37 times faster.

4.6 Final results

From the analysis carried out in the previous section, the most important result to be highlighted is that the performance of the GPP+DSP solution is better than the GPP+NEON solution according to both the execution time and the GPP workload. Nevertheless, the two solutions are not mutually exclusive. This means that they can cooperate and be run in parallel on the same OMAP3530 platform. A suitable trade-off can be designed to achieve the maximum level of performance for such platform for a given computation.

Regarding **performance**, the best system configuration to optimize the *execution time* is when the GPP cooperates with the DSP and the IPC protocol used is MPCS. However, this IPC protocol leads to the fastest execution while exhibiting the worst performance in terms of the two *processors workload* (both GPP and DSP workloads). A high processor load leads to higher power consumption and reduces the parallelism of the system; as the GPP processor is not able to perform other tasks while waiting for the data to be ready. A more balanced solution is achieved by utilizing the MSGQ module. This IPC mechanism shows the best performance in terms of GPP workload, while exhibiting a reasonable result for the DSP workload. The loss of speed with respect to the MPCS module is 7.36% according to figure 4.3. From the point of view of the programming complexity, the MSGQ approach is very straightforward, since the concept of message-based communication is widely

used in many distributed systems. By utilizing the MSGQ IPC protocol, a more robust system is achieved since the MPCS does not tolerate deadlocks and priority inversions in DSP/BIOS 5.x (see 2.6.1.4). Concerning the GPP+NEON solution, the best timing performance is achieved by developing NEON code with *intrinsics*. This offers the best trade-off, since it merges the architecture control of the assembly language and high level programming capability of the C language. With respect to the processor workload, there are not any relevant differences among the different method for developing code for NEON. If the system should perform floating point operations, two possibilities can be exploited considering the design constraints. If the execution speed has priority over the loss of precision, the floating point operations can be executed by the DSP. Since the DSP does not provide a FPU, the programmer must convert floating point numbers to fixed point numbers by using the inline assembly implementation of the IQMath library. However, a preliminary study to minimize the loss of precision is needed according to the application constraints. If the loss of precision is not negligible, then the NEON coprocessor should be used since no precision loss will occur by actually performing floating point operations. The NEON performance related to floating point operations is better than the performance of the DSP when the C implementation of the IQMath library is used. The fact that the NEON coprocessor was not able to achieve the same performance as the DSP for the FIR filter considered in this thesis project, does not imply that this resource should not be exploited. Depending on the type of computations and on algorithms that must be executed, an efficient trade-off in distributing tasks over the system can be designed to maximize the platform performance. An example of design could utilize the NEON to compute floating point operations which have strict precision constraints while in parallel the DSP can execute integer or fixed point operations that can tolerate precision losses.

Another factor to be considered when choosing the proper system configuration is the system **determinism**. The deterministic behavior is referred to the timing. In particular, DSPLink introduces a communication layer that introduces some differences in times measured for different executions of the same algorithm. These differences are evident but still negligible for the case of a single-DSP system. Anyway, since DSPLink was designed to support a multi-DSP environment, this issue should be considered carefully when executing critical tasks in a real-time system. This non-deterministic behavior can be avoided by using NEON. Since the communication between GPP and NEON is directly done by accessing to the same memory space, any source of non-determinism is avoided.

Another design parameter is the desired degree of **parallelism**. The first aspect is the parallelism when exchanging data. If the designer wants to enable the communication layer to send several chunks at the same time via DSPLink, then the MSGQ module should not be used. In fact, the MSGQ mechanism limits the degree of parallelism of the system since a message queue can have only one reader - although it can have multiple writers. Instead, either MPLIST or RINGIO

module could be used. However, as showed in the previous sections, MPLIST offers better performance than RINGIO both regarding the timing and the processors workloads. With respect to the system-level parallelism, an implementation should exploit both DSP and NEON. As explained in section 4.4, long chains of blocks must be executed by the DSP which offers better long-term performance. However, sporadic tasks can be executed by NEON. If the GPP+DSP configuration is chosen, then the GPP load is low (if MSGQ module is used). Hence, the GPP can execute other tasks in parallel (e.g. input/output, control operations, etc.) while waiting for data from DSP. In case of highly parallelizable code, the DSPLink layer can reduce the degree of parallelism of the code since data are split. On the other hand, the NEON coprocessor must be used since its vector processor architecture obtains the maximum performance when executing parallel code.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The objective of this thesis project was to investigate the performance of a OMAP3530 platform when executing signal processing algorithms. This study aimed to facilitate the porting of SDR systems to embedded systems such as the OMAP platform.

The first goal has been the analysis of the performance of the system as a function of the IPC protocol used for communication between the GPP and the DSP. The system performance was analyzed in terms of timing and processor workloads. After the analysis of the DSP-based solution, a second goal was defined. This second goal was to find another system configuration that would further improve the system performance. Hence, the NEON vector coprocessor, embedded in the ARM Cortex-A8 GPP was exploited. Such a solution, based on NEON, was analyzed. Finally, the two solutions have been compared regarding both integer and floating point operations.

Results of the analysis have showed that the best system performance is achieved when using the DSP for signal processing computations. In particular, the optimal DSPLink setup was obtained with the MSGQ module. This module offers the best general system performance, although it is not able to guarantee the best timing performance. To guarantee the best timing performance requires use of the MPCS module, however, this leads to an extremely high processor (over)load. The MSGQ DSPLink module offers both good execution time performance and offloads the GPP. Using the DSP leads to two main benefits due to offloading the GPP. The first advantage is reduced power consumption of the system. The GPP can be put in a standby state while waiting for data from the DSP (if no other tasks need to be executed). The second benefit is that the degree of parallelism can be increased if other tasks are executed by the GPP while the DSP processes data. Exploiting this parallelism further decreases the power consumption and increases the performance

of the system.

Unfortunately, the NEON vector coprocessor is not able to guarantee the same performance as the DSP. However, its role in the system should not be underestimated. It can be used to execute highly parallelizable algorithms contemporaneously to the DSP execution. Additionally, NEON is the only system resource offering a FPU. Floating point operations can be performed in hardware via the NEON. Another way to execute floating point computations is to convert floating point numbers to fixed point representations, then perform the operations using the fixed point unit of the DSP. In order to perform such conversions, TI's IQMath library was utilized. The performance achieved by using this library is better than the performance offered by NEON. Unfortunately due to the use of a fixed point representation, a loss in precision can occur.

As a result of this thesis project I have developed a good mastery of DSPLink software. DSPLink was an excellent tool to exchange data between the two processor cores. It generally offers good performance in terms of timing and reliability. Nonetheless, there are some shortcomings in using this tool. The main drawback is the long learning curve. A large amount of time must be spent in study in order to deeply understand the IPC mechanisms which this software provides. In order to do so, I have found the sample programs provided with the DSPLink package to be really useful and fundamental to enabling my understanding of this software. Although these programs only show basic functionalities and capabilities of the tool, they are useful when starting to work with DSPLink. Furthermore, the designer must deeply know the target system, since the memory pools must be manually allocated by the programmer. Hence, the resources needed by the application must be manually allocated *a priori*. This implies a preliminary long evaluation study of the application and makes the system difficult to scale.

5.2 Future work

This thesis project has evaluated the performance of the OMAP3530. The focus of this project was mostly on improvements to the system in terms of timing performance. Another analysis could evaluate the power consumption of the two main system configurations: DSP-based and NEON-based configurations. Only a limited power consumption analysis was carried out during this thesis project. Power consumption plays an important role in the portability of SDR systems to embedded platforms and can be a future extension and further development of this project.

The analysis done in this project can be further improved in two directions that were not explored due to the limited time available for this thesis project. The first direction is to exploit the C compiler provided by ARM to replace the GCC-based

one for certain tasks. As already explained throughout this report, sometimes the cross compiler used in this project resulted in bad performance. Two examples are the ability of the compiler to vectorize the code for targeting NEON and the lack of support for the option `-mfloat-abi=hard` that would lead to optimized execution of floating point operations on the NEON FPU. It would be interesting to try to achieve these goals with the ARM `armcc`. The second direction is to further improve the memory transfer operations. In order to do this, two options are available:

- `cmem`: is an API and library to manage blocks of memory which are physically contiguous. This library can be used to improve the reliability and availability of the system by using its pool-based configuration to allocate shared memory between the GPP and the DSP. In fact, `cmem` ensures contiguous memory even after that the software has run for a long time; and

- DMA: the DMA engine can be used on the GPP processor to speed up memory transfers. Instead, on the DSP side is provided an Enhanced Direct Memory Access (EDMA) Controller for data transfers between the L2 cache and the device peripherals. This unit may speed up data transfer when peripheral devices are directly connected to the DSP.

In order to improve the performance of the code, two tools can be useful for future applications:

- OpenMAX™ a royalty-free, cross-platform API developed by the Khronos Group that provides a comprehensive media codec and application portability by enabling accelerated multimedia components to be developed, integrated, and programmed across multiple operating systems and silicon platforms[1]. OpenMAX is related to this master thesis project since an optimized implementation of the library to target the NEON coprocessor was developed by ARM;

- Embedded MATLAB offers a subset of the MATLAB language that supports efficient code generation for prototyping and deploying embedded systems, and accelerating fixed-point algorithms. It consists of more than 270 MATLAB operators and functions and more than 90 functions from the Fixed-Point Toolbox™ software[2]. This tool can be used in conjunction with the IQMath library to perform floating point operations on the DSP.

Finally, some consideration about the hardware tools must be reviewed. During this project, all the experiments were executed on a BeagleBoard Revision C3. This platform is not at the cutting edge, since its working frequency is slower than the leading edge products in the market. While such a platform was perfect for the project, it should not be used in future product development. There are

[1] from: http://www.arm.com/community/multimedia/standards-apis.php
[2] from: http://www.mathworks.com/help/toolbox/eml/gs/brqncq7-1.html

two possibilities to improve the hardware. The first possibility is to use the last model of BeagleBoard. This version of the board is called BeagleBoard-xM and, at the time of writing, the latest revision is A2. This board consists of a DM3730 instead of the OMAP3530. The DM3730 platform contains an ARM Cortex-A8 with a working frequency of up to 1 GHz and includes a NEON coprocessor. The DSP is a TMS320C64x+ 800 MHz DSP core. Using such a platform, enables the use of DSP/BIOS 6.x in conjuntion with DSPLink. Even more interesting would be analyzing the performance of TI's TMS320C6A8167 and TMS320C6A8168 processors. These processors both contain an ARM Cortex-A8 processor with up to a 1.5 GHz working frequency with NEON technology. The DSP is TI's TMS320C674x floating point VLIW DSP, fully compatible with the c64x+ DSP, used during this project.

Appendix A

Texas Instruments Development Tools

Texas Instruments provides a range of tool supporting the development of DSP/BIOS applications.

- **Code Composer Studio (CCStudio) IDE**: an integrated development environment for programming Texas Instruments DSP families. It is based on the Eclipse open source software framework. The tools offers to programmers compilers for every TI DSP, debuggers, profilers and simulators;
- **Kernel Object Viewer**: a part of CCStudio framework. This viewer is used for debugging purposes and enables developers to check the status of OS objects such as tasks, semaphores, and mailboxes. A task can have three possible statuses: ready, running, and blocked. In the ready status, a task is ready to be executed, the system has already checked the availability of resources but is waiting for the processor. In the running state, a task is being executed by the processor. In the blocked state, the task is waiting on an IPC object (semaphore, mailbox, or event);
- **Real-time Analysis**: another part of the CCStudio framework. This tools graphically displays the threads execution and switching sequence. Additional information is also presented, such as the average and maximum execution time per thread and system CPU load.

Appendix B

Performance of DSPLink modules

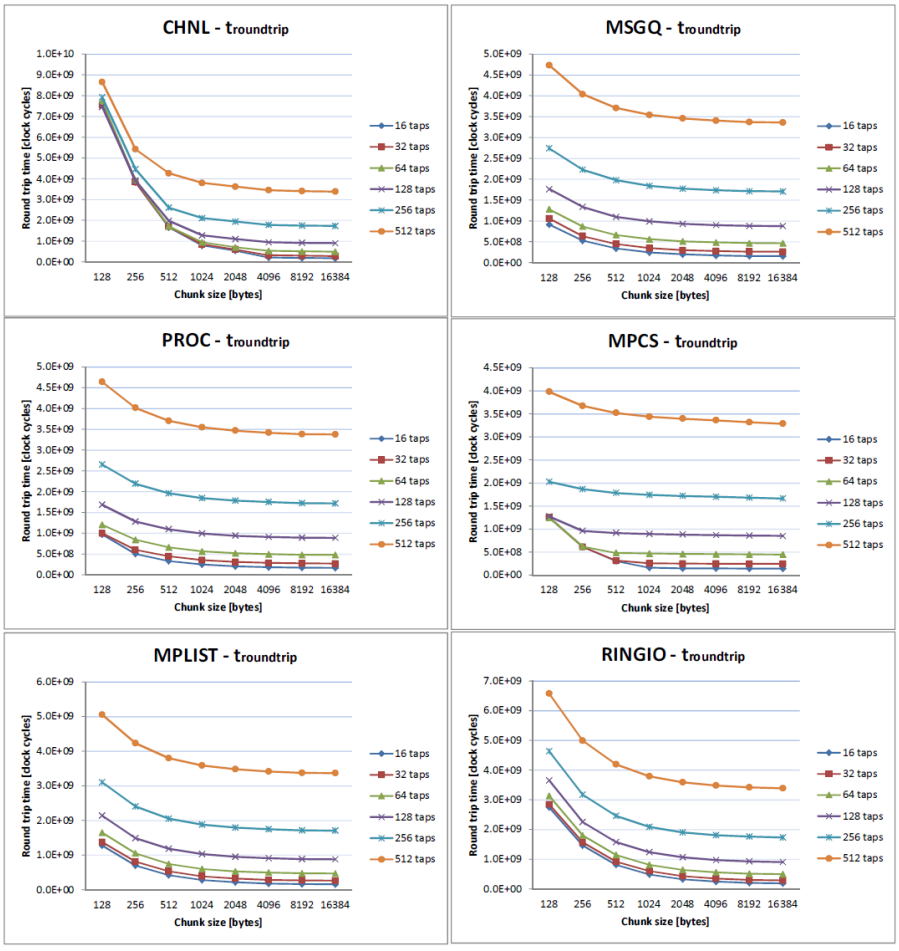

Figure B.1. Chunks round trip time of DSPLink modules

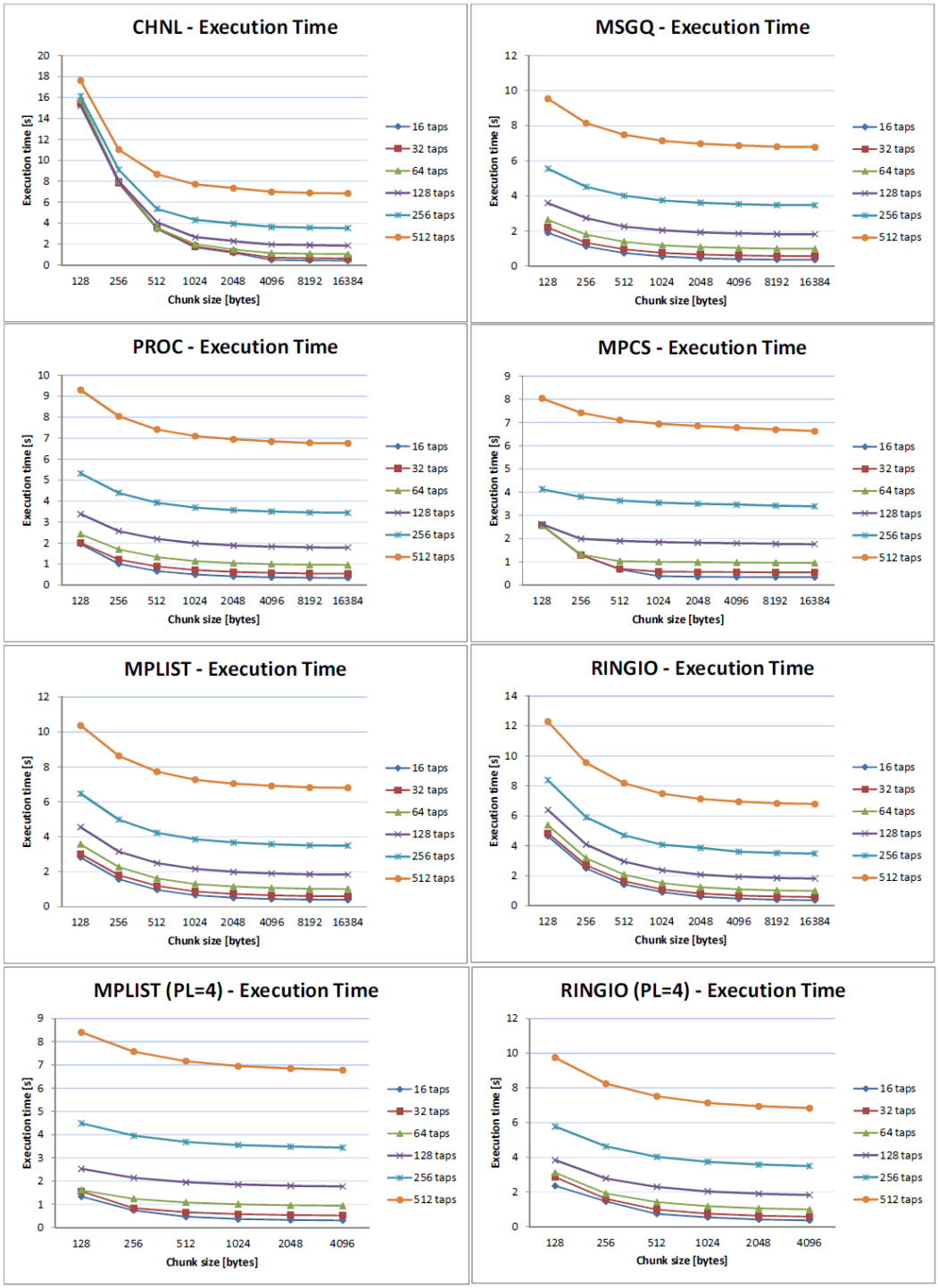

Figure B.2. Execution time of DSPLink modules

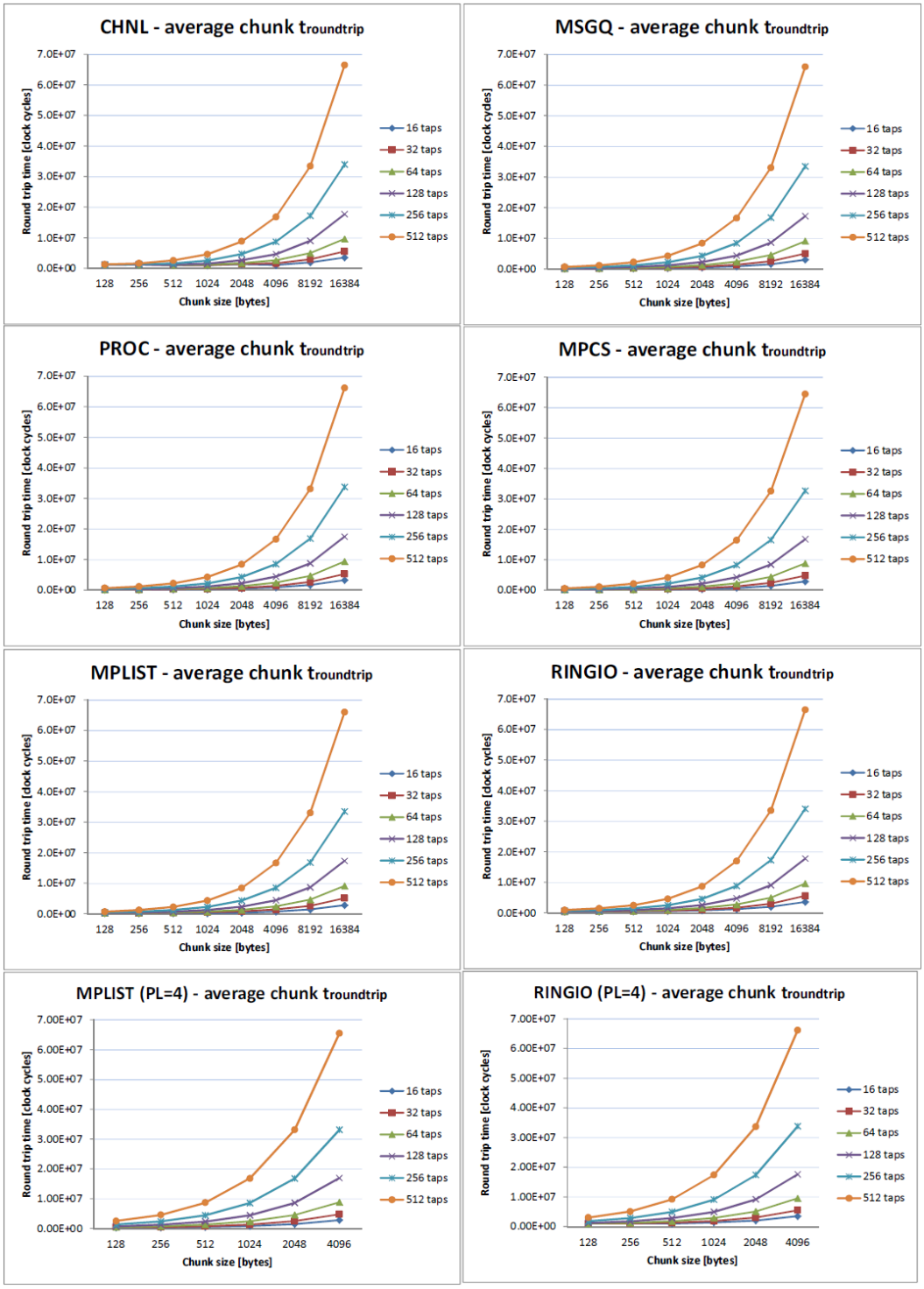

Figure B.3. Average chunk round trip time of DSPLink modules

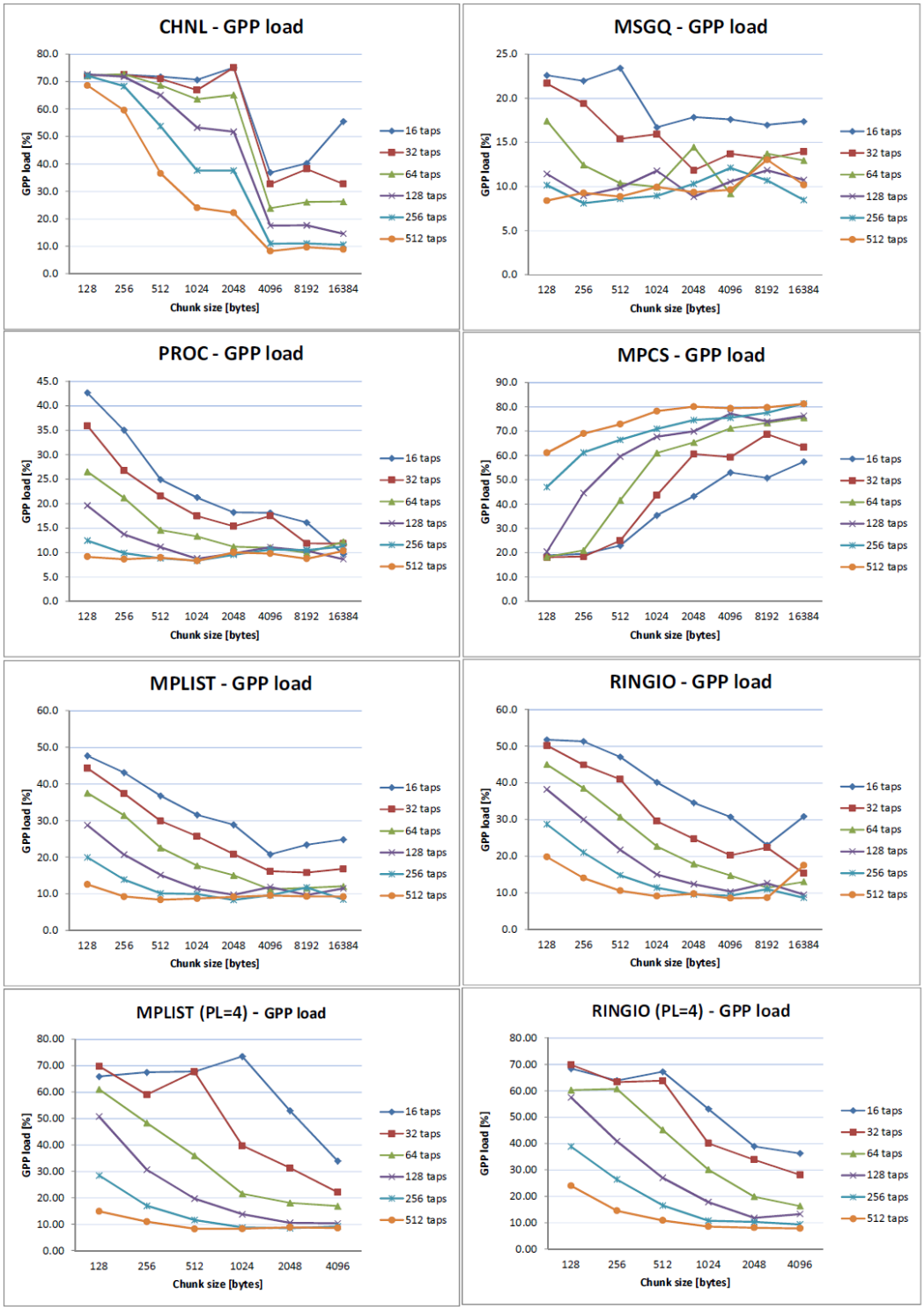

Figure B.4. GPP load of DSPLink modules

Appendix C

Example of OProfile output report

```
CPU: ARM V7 PMNC, speed 0 MHz (estimated)
Counted CPU_CYCLES events (Number of CPU cycles)
with a unit mask of 0x00 (No unit mask) count 500

samples  %        image name          app name    symbol name

432      46.7027  vmlinux             vmlinux     omap3_pm_idle
157      16.9730  vmlinux             vmlinux     cpu_idle
22        2.3784  ld-2.9.so           msgq_FIR    /lib/ld-2.9.so
21        2.2703  ld-2.9.so           busybox     /lib/ld-2.9.so
12        1.2973  vmlinux             vmlinux     schedule
11        1.1892  libc-2.9.so         msgq_FIR    /lib/libc-2.9.so
11        1.1892  vmlinux             vmlinux     v7_flush_kern_dcache_page
10        1.0811  vmlinux             vmlinux     sub_preempt_count
10        1.0811  vmlinux             vmlinux     unmap_vmas
9         0.9730  vmlinux             vmlinux     copy_page
8         0.8649  vmlinux             vmlinux     __do_softirq
7         0.7568  vmlinux             vmlinux     cpu_v7_dcache_clean_area
7         0.7568  vmlinux             vmlinux     kmem_cache_alloc
7         0.7568  vmlinux             vmlinux     mmc_blk_issue_rq
7         0.7568  vmlinux             vmlinux     omap_hsmmc_start_command
6         0.6486  busybox             busybox     /bin/busybox
6         0.6486  vmlinux             vmlinux     __memzero
5         0.5405  vmlinux             vmlinux     __up_read
5         0.5405  vmlinux             vmlinux     filemap_fault
5         0.5405  vmlinux             vmlinux     update_mmu_cache
4         0.4324  libc-2.9.so         busybox     /lib/libc-2.9.so
4         0.4324  vmlinux             vmlinux     __d_lookup
4         0.4324  vmlinux             vmlinux     __down_read_trylock
4         0.4324  vmlinux             vmlinux     add_preempt_count
4         0.4324  vmlinux             vmlinux     cpu_v7_set_pte_ext
4         0.4324  vmlinux             vmlinux     get_page_from_freelist
4         0.4324  vmlinux             vmlinux     pfn_valid
3         0.3243  vmlinux             vmlinux     __do_fault
3         0.3243  vmlinux             vmlinux     __make_request
3         0.3243  vmlinux             vmlinux     __pabt_usr
3         0.3243  vmlinux             vmlinux     __wake_up_bit
3         0.3243  vmlinux             vmlinux     do_page_fault
```

```
3         0.3243  vmlinux            vmlinux     find_get_page
3         0.3243  vmlinux            vmlinux     handle_mm_fault
2         0.2162  vmlinux            vmlinux     do_filp_open
2         0.2162  vmlinux            vmlinux     do_mmap_pgoff
2         0.2162  vmlinux            vmlinux     find_vma
2         0.2162  vmlinux            vmlinux     get_request
2         0.2162  vmlinux            vmlinux     load_elf_binary
2         0.2162  vmlinux            vmlinux     mutex_lock
2         0.2162  vmlinux            vmlinux     number
2         0.2162  vmlinux            vmlinux     page_add_file_rmap
2         0.2162  vmlinux            vmlinux     sys_mmap_pgoff
2         0.2162  vmlinux            vmlinux     tick_nohz_restart_sched_tick
2         0.2162  vmlinux            vmlinux     tick_nohz_stop_sched_tick
2         0.2162  vmlinux            vmlinux     uart_start
2         0.2162  vmlinux            vmlinux     unlock_page
2         0.2162  vmlinux            vmlinux     vector_swi
2         0.2162  vmlinux            vmlinux     vm_normal_page
1         0.1081  ld-2.9.so          oprofiled   /lib/ld-2.9.so
1         0.1081  libpthread-2.9.so  msgq_FIR    /lib/libpthread-2.9.so
1         0.1081  vmlinux            vmlinux     ____pagevec_lru_add
1         0.1081  vmlinux            vmlinux     __aeabi_uidivmod
1         0.1081  vmlinux            vmlinux     __clear_user_std
1         0.1081  vmlinux            vmlinux     __dabt_usr
1         0.1081  vmlinux            vmlinux     __dentry_open
1         0.1081  vmlinux            vmlinux     _down_read
```

Appendix D

NEON disassembled floating point code

Listing D.1. NEON disassembled floating point code

```
00008a90 <FIR_core>:
    8a90: e92d07f0  push  {r4, r5, r6, r7, r8, r9, sl}
    8a94: e2535000  subs  r5, r3, #0  ; 0x0
    8a98: ed2d8b0c  vstmdb  sp!, {d8-d13}
    8a9c: e1a04001  mov r4, r1
    8aa0: da000068  ble 8c48 <FIR_core+0x1b8>
    8aa4: e2828008  add r8, r2, #8  ; 0x8
    8aa8: e288a008  add sl, r8, #8  ; 0x8
    8aac: e28a9008  add r9, sl, #8  ; 0x8
    8ab0: e2896008  add r6, r9, #8  ; 0x8
    8ab4: e2867008  add r7, r6, #8  ; 0x8
    8ab8: e3a0c000  mov ip, #0  ; 0x0
    8abc: e2871008  add r1, r7, #8  ; 0x8
    8ac0: ea000027  b 8b64 <FIR_core+0xd4>
    8ac4: edd02b00  vldr  d18, [r0]
    8ac8: edd21b00  vldr  d17, [r2]
    8acc: edd00b02  vldr  d16, [r0, #8]
    8ad0: f3422db1  vmul.f32  d18, d18, d17
    8ad4: edd81b00  vldr  d17, [r8]
    8ad8: f3400db1  vmul.f32  d16, d16, d17
    8adc: f2422da0  vadd.f32  d18, d18, d16
    8ae0: edd01b04  vldr  d17, [r0, #16]
    8ae4: edda0b00  vldr  d16, [sl]
    8ae8: f3411db0  vmul.f32  d17, d17, d16
    8aec: f2422da1  vadd.f32  d18, d18, d17
    8af0: edd00b06  vldr  d16, [r0, #24]
    8af4: edd91b00  vldr  d17, [r9]
    8af8: f3400db1  vmul.f32  d16, d16, d17
    8afc: f2422da0  vadd.f32  d18, d18, d16
    8b00: edd01b08  vldr  d17, [r0, #32]
    8b04: edd60b00  vldr  d16, [r6]
    8b08: f3411db0  vmul.f32  d17, d17, d16
    8b0c: f2422da1  vadd.f32  d18, d18, d17
    8b10: edd00b0a  vldr  d16, [r0, #40]
    8b14: edd71b00  vldr  d17, [r7]
    8b18: f3400db1  vmul.f32  d16, d16, d17
    8b1c: f2422da0  vadd.f32  d18, d18, d16
    8b20: edd01b0c  vldr  d17, [r0, #48]
    8b24: edd10b00  vldr  d16, [r1]
    8b28: f3411db0  vmul.f32  d17, d17, d16
```

```
    8b2c: f2422da1  vadd.f32  d18, d18, d17
    8b30: edd00b0e  vldr  d16, [r0, #56]
    8b34: edd11b02  vldr  d17, [r1, #8]
    8b38: f3400db1  vmul.f32  d16, d16, d17
    8b3c: f2400da2  vadd.f32  d16, d16, d18
    8b40: f3400da0  vpadd.f32 d16, d16, d16
    8b44: ee103b90  vmov.32 r3, d16[0]
    8b48: ee0d3a10  fmsr  s26, r3
    8b4c: ee1d3a10  fmrs  r3, s26
    8b50: e784310c  str r3, [r4, ip, lsl #2]
    8b54: e28cc001  add ip, ip, #1  ; 0x1
    8b58: e155000c  cmp r5, ip
    8b5c: e2800004  add r0, r0, #4  ; 0x4
    8b60: da000038  ble 8c48 <FIR_core+0x1b8>
    8b64: e1823000  orr r3, r2, r0
    8b68: e3130007  tst r3, #7  ; 0x7
    8b6c: 0afffd4  beq 8ac4 <FIR_core+0x34>
    8b70: ed927a01  flds  s14, [r2, #4]
    8b74: edd07a01  flds  s15, [r0, #4]
    8b78: ee27da27  fmuls s26, s14, s15
    8b7c: edd26a00  flds  s13, [r2]
    8b80: edd07a00  flds  s15, [r0]
    8b84: ed927a02  flds  s14, [r2, #8]
    8b88: ed906a02  flds  s12, [r0, #8]
    8b8c: edd28a03  flds  s17, [r2, #12]
    8b90: ed909a03  flds  s18, [r0, #12]
    8b94: edd29a04  flds  s19, [r2, #16]
    8b98: ed90aa04  flds  s20, [r0, #16]
    8b9c: edd2aa05  flds  s21, [r2, #20]
    8ba0: ee06daa7  fmacs s26, s13, s15
    8ba4: ed90ba05  flds  s22, [r0, #20]
    8ba8: edd2ba06  flds  s23, [r2, #24]
    8bac: ed90ca06  flds  s24, [r0, #24]
    8bb0: edd2ca07  flds  s25, [r2, #28]
    8bb4: ed902a07  flds  s4, [r0, #28]
    8bb8: edd21a08  flds  s3, [r2, #32]
    8bbc: ed901a08  flds  s2, [r0, #32]
    8bc0: edd20a09  flds  s1, [r2, #36]
    8bc4: ed900a09  flds  s0, [r0, #36]
    8bc8: ee07da06  fmacs s26, s14, s12
    8bcc: ed928a0a  flds  s16, [r2, #40]
    8bd0: ed905a0a  flds  s10, [r0, #40]
    8bd4: edd24a0b  flds  s9, [r2, #44]
    8bd8: ed904a0b  flds  s8, [r0, #44]
    8bdc: edd23a0c  flds  s7, [r2, #48]
    8be0: ed903a0c  flds  s6, [r0, #48]
    8be4: edd22a0d  flds  s5, [r2, #52]
    8be8: ed906a0d  flds  s12, [r0, #52]
    8bec: edd25a0e  flds  s11, [r2, #56]
    8bf0: ee08da89  fmacs s26, s17, s18
    8bf4: ed907a0e  flds  s14, [r0, #56]
    8bf8: edd26a0f  flds  s13, [r2, #60]
    8bfc: edd07a0f  flds  s15, [r0, #60]
    8c00: e2800004  add r0, r0, #4  ; 0x4
    8c04: ee09da8a  fmacs s26, s19, s20
    8c08: ee0ada8b  fmacs s26, s21, s22
    8c0c: ee0bda8c  fmacs s26, s23, s24
    8c10: ee0cda82  fmacs s26, s25, s4
    8c14: ee01da81  fmacs s26, s3, s2
    8c18: ee00da80  fmacs s26, s1, s0
    8c1c: ee08da05  fmacs s26, s16, s10
    8c20: ee04da84  fmacs s26, s9, s8
```

```
    8c24: ee03da83  fmacs s26, s7, s6
    8c28: ee02da86  fmacs s26, s5, s12
    8c2c: ee05da87  fmacs s26, s11, s14
    8c30: ee06daa7  fmacs s26, s13, s15
    8c34: ee1d3a10  fmrs  r3, s26
    8c38: e784310c  str r3, [r4, ip, lsl #2]
    8c3c: e28cc001  add ip, ip, #1  ; 0x1
    8c40: e155000c  cmp r5, ip
    8c44: cafffc6  bgt 8b64 <FIR_core+0xd4>
    8c48: ecbd8b0c  vldmia  sp!, {d8-d13}
    8c4c: e8bd07f0  pop {r4, r5, r6, r7, r8, r9, sl}
    8c50: e12fff1e  bx  lr
```

Appendix E

Data of DSPLink performance

CHNL module

HAMMING16								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	1163886.64	7.66E+09	15.573200	72.19	1654566052	91478005	94.47
256	3290	1166488.82	3.84E+09	7.845490	72.42	832307918	43090321	94.82
512	1645	1014267.44	1.67E+09	3.467220	71.73	442314088	20868286	95.28
1024	822	973101.46	8.00E+08	1.695430	70.62	241105008	8430835	96.50
2048	411	1334147.36	5.48E+08	1.171880	75.08	173096276	4406984	97.45
4096	205	1058038.84	2.17E+08	0.502381	36.85	139884192	2714547	98.06
8192	102	1915293.21	1.95E+08	0.453369	40.23	124435612	1964158	98.42
16384	51	3514570.39	1.79E+08	0.416779	55.50	117249136	1659750	98.58
HAMMING32								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	1149658.17	7.56E+09	15.424700	72.08	1659835826	86944673	94.76
256	3290	1168361.19	3.84E+09	7.864080	72.39	860696074	39914915	95.36
512	1645	1035401.68	1.70E+09	3.528810	71.03	456709496	15323356	96.64
1024	822	1032388.67	8.49E+08	1.790370	66.87	315574434	7800292	97.53
2048	411	1464457.06	6.02E+08	1.276400	75.12	248563564	4513596	98.18
4096	205	1579315.15	3.24E+08	0.717712	32.65	215856608	2823000	98.69
8192	102	2944369.85	3.00E+08	0.663208	38.14	198804946	1954119	99.02
16384	51	5538135.69	2.82E+08	0.622772	32.68	191665030	1594626	99.17
HAMMING64								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	1175603.06	7.74E+09	15.771700	72.17	1701049982	78013916	95.41
256	3290	1199107.17	3.95E+09	8.066250	72.64	890318494	29106145	96.73
512	1645	1045953.85	1.72E+09	3.562160	68.68	600519238	13966661	97.67
1024	822	1159670.17	9.53E+08	2.001770	63.58	467404820	7631266	98.37
2048	411	1721251.82	7.07E+08	1.485320	65.10	401492492	4498526	98.88
4096	205	2601986.85	5.33E+08	1.136140	23.90	366215244	2862913	99.22
8192	102	4989408.51	5.09E+08	1.080290	26.10	348626304	2066200	99.41
16384	51	9629394.51	4.91E+08	1.040310	26.28	341136084	1612782	99.53
HAMMING128								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	1134465.70	7.46E+09	15.213000	72.61	1780609894	53255977	97.01
256	3290	1191124.57	3.92E+09	8.013460	71.79	1201015240	26910605	97.76
512	1645	1202746.38	1.98E+09	4.089230	65.03	916378702	14219676	98.45
1024	822	1562616.45	1.28E+09	2.666020	53.25	777574302	7699846	99.01
2048	411	2686402.74	1.10E+09	2.286040	51.73	705515716	4501861	99.36
4096	205	4636977.05	9.51E+08	1.970340	17.55	668054160	2940752	99.56
8192	102	9049752.27	9.23E+08	1.909000	17.64	647210254	1954832	99.70
16384	51	17740479.88	9.05E+08	1.867430	14.61	639447926	1552460	99.76
HAMMING256								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	1204035.64	7.92E+09	16.106400	72.11	2472607400	53277763	97.85
256	3290	1356120.70	4.46E+09	9.102200	68.31	1858813098	27402888	98.53
512	1645	1590740.07	2.62E+09	5.367430	53.73	1545505872	14324249	99.07
1024	822	2565480.83	2.11E+09	4.313810	37.63	1391161866	8013984	99.42
2048	411	4728038.76	1.94E+09	3.961240	37.59	1312280646	4647202	99.65
4096	205	8719253.20	1.79E+09	3.644070	11.00	1269867486	2862795	99.77
8192	102	17182609.98	1.75E+09	3.567780	11.05	1246670408	2183128	99.82
16384	51	33984412.98	1.73E+09	3.524870	10.57	1237043382	1674896	99.86
HAMMING512								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	1315944.52	8.66E+09	17.597100	68.57	3894761182	53748211	98.62
256	3290	1648249.14	5.42E+09	11.018400	59.53	3175262238	28171145	99.11
512	1645	2595076.07	4.27E+09	8.668210	36.57	2801361482	14377868	99.49
1024	822	4630169.97	3.81E+09	7.708010	24.05	2618016420	7701462	99.71
2048	411	8832145.21	3.63E+09	7.336180	22.23	2527052918	4595500	99.82
4096	205	16861801.63	3.46E+09	6.982090	8.26	2473334416	2761312	99.89
8192	102	33439134.24	3.41E+09	6.884220	9.66	2440585986	2085773	99.91
16384	51	66457100.55	3.39E+09	6.835970	8.93	2430044914	1554349	99.94

MSGQ module

HAMMING16								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	139059.67	9.15E+008	1.899990	22.60	666942834	30698370	95.40
256	3290	161068.56	5.30E+008	1.121150	21.97	369273080	15116185	95.91
512	1645	207753.91	3.42E+008	0.739380	23.42	238639536	8224969	96.55
1024	822	299039.39	2.46E+008	0.545563	16.70	173400370	4890330	97.18
2048	411	478855.56	1.97E+008	0.446656	17.85	139511942	3132795	97.75
4096	205	830374.98	1.70E+008	0.392609	17.60	121972100	2111625	98.27
8192	102	1543929.95	1.57E+008	0.365692	16.98	112475047	1625751	98.55
16384	51	3025722.41	1.54E+008	0.358581	17.38	110464729	1431068	98.70
HAMMING32								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	160856.47	1.06E+009	2.187500	21.67	738223392	29994513	95.94
256	3290	193220.61	6.36E+008	1.331940	19.40	453025148	15454288	96.59
512	1645	273274.59	4.50E+008	0.956147	15.37	324252004	8887503	97.26
1024	822	421716.12	3.47E+008	0.747192	15.92	249210220	4871237	98.05
2048	411	728376.06	2.99E+008	0.651214	11.83	214650898	3087640	98.56
4096	205	1340640.72	2.75E+008	0.601532	13.70	197135557	2219278	98.87
8192	102	2562342.24	2.61E+008	0.573425	13.16	187389259	1690481	99.10
16384	51	5061431.29	2.58E+008	0.565674	13.94	185419235	1385941	99.25
HAMMING64								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	194864.21	1.28E+009	2.633850	17.43	905283972	29240832	96.77
256	3290	264045.71	8.69E+008	1.798000	12.43	619267128	15079691	97.56
512	1645	403896.04	6.64E+008	1.385220	10.37	481233456	8850916	98.16
1024	822	682701.72	5.61E+008	1.176090	9.96	407157016	5193599	98.72
2048	411	1240602.84	5.10E+008	1.072630	14.47	369955896	3244501	99.12
4096	205	2363118.45	4.84E+008	1.020870	9.18	348120720	2146011	99.38
8192	102	4596643.48	4.69E+008	0.988312	13.72	336829165	1690119	99.50
16384	51	9124340.69	4.65E+008	0.980530	12.95	334234087	1390357	99.58
HAMMING128								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	267858.68	1.76E+009	3.594910	11.43	1275877702	30908298	97.58
256	3290	406427.77	1.34E+009	2.734860	8.95	956700212	15942325	98.33
512	1645	666997.14	1.10E+009	2.250520	9.87	793200964	8867080	98.88
1024	822	1206058.14	9.91E+008	2.036740	11.78	709600386	4739535	99.33
2048	411	2268792.07	9.32E+008	1.918550	8.83	671165828	3036772	99.55
4096	205	4400292.43	9.02E+008	1.856050	10.55	649167005	2135343	99.67
8192	102	8661034.40	8.83E+008	1.817870	11.84	635581057	1698998	99.73
16384	51	17247590.33	8.80E+008	1.809200	10.74	633057019	1473796	99.77
HAMMING256								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	416745.26	2.74E+009	5.554440	10.15	1986074390	31197926	98.43
256	3290	677537.54	2.23E+009	4.518560	8.10	1611189012	16298523	98.99
512	1645	1202933.70	1.98E+009	4.014010	8.59	1415906570	8420153	99.41
1024	822	2239484.42	1.84E+009	3.735570	8.94	1322898214	5044506	99.62
2048	411	4320697.30	1.78E+009	3.604770	10.31	1278733266	3209929	99.75
4096	205	8473615.36	1.74E+009	3.526340	12.11	1250825785	2096453	99.83
8192	102	16791145.45	1.71E+009	3.476070	10.68	1232876991	1707364	99.86
16384	51	33485807.47	1.71E+009	3.465880	8.46	1229897446	1406760	99.89
HAMMING512								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	719602.42	4.73E+009	9.540160	8.40	3376630114	29833914	99.12
256	3290	1227791.84	4.04E+009	8.140080	9.28	2904685360	15637693	99.46
512	1645	2255174.85	3.71E+009	7.475740	8.86	2676262726	9121920	99.66
1024	822	4309792.65	3.54E+009	7.139190	9.93	2550027712	4917625	99.81
2048	411	8411016.98	3.46E+009	6.966770	9.35	2490236885	3117905	99.87
4096	205	16618969.99	3.41E+009	6.866030	9.63	2453996411	2283180	99.91
8192	102	33040263.53	3.37E+009	6.791080	13.03	2427145001	1713380	99.93
16384	51	65940143.86	3.36E+009	6.775240	10.17	2422415252	1400983	99.94

PROC module

HAMMING16								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	147954.45	9.74E+08	1.959380	42.65	693032903	36117505	94.79
256	3290	153571.08	5.05E+08	1.016880	35.03	365363289	16441759	95.50
512	1645	200842.36	3.30E+08	0.664276	24.92	223692904	8044795	96.40
1024	822	302651.84	2.49E+08	0.499451	21.25	156643753	4084386	97.39
2048	411	496655.60	2.04E+08	0.409424	18.20	127424207	2380528	98.13
4096	205	889403.56	1.82E+08	0.365357	18.10	112362323	1536070	98.63
8192	102	1676751.53	1.71E+08	0.342559	16.13	104441871	1055622	98.99
16384	51	3285959.06	1.68E+08	0.335602	9.67	102450423	839089	99.18
HAMMING32								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	151847.10	9.99E+08	2.010930	35.88	726968277	32783830	95.49
256	3290	183369.33	6.03E+08	1.212830	26.76	412575185	14379198	96.51
512	1645	270939.13	4.46E+08	0.894776	21.57	294384652	7589677	97.42
1024	822	430564.04	3.54E+08	0.709717	17.47	241072697	4635771	98.08
2048	411	754271.17	3.10E+08	0.621155	15.36	204301599	2489200	98.78
4096	205	1402016.29	2.87E+08	0.575500	17.50	187781043	1551426	99.17
8192	102	2694095.11	2.75E+08	0.550141	11.83	180148587	1124190	99.38
16384	51	5317799.39	2.71E+08	0.542816	11.83	177025757	832592	99.53
HAMMING64								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	183246.08	1.21E+09	2.423830	26.51	854889157	29852886	96.51
256	3290	256951.53	8.45E+08	1.697270	21.18	581973225	14684711	97.48
512	1645	403898.04	6.64E+08	1.332150	14.57	455620391	7858429	98.28
1024	822	686187.32	5.64E+08	1.129940	13.31	389756749	4363272	98.88
2048	411	1269228.33	5.22E+08	1.044400	11.19	356121005	2480368	99.30
4096	205	2422406.60	4.97E+08	0.993897	10.96	338902903	1603642	99.53
8192	102	4729499.94	4.82E+08	0.965301	9.92	329087415	1077975	99.67
16384	51	9379934.47	4.78E+08	0.957153	12.17	326154035	834304	99.74
HAMMING128								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	256246.14	1.69E+09	3.384310	19.58	1215565409	30399210	97.50
256	3290	390004.76	1.28E+09	2.572450	13.71	917863045	15313291	98.33
512	1645	666997.65	1.10E+09	2.197780	11.11	775845731	8310187	98.93
1024	822	1208460.60	9.93E+08	1.988530	8.75	697961019	4416826	99.37
2048	411	2292675.80	9.42E+08	1.885650	9.81	660630743	2541917	99.62
4096	205	4461638.49	9.15E+08	1.829990	11.09	640077275	1631334	99.75
8192	102	8795364.49	8.97E+08	1.794740	10.38	627911885	1093388	99.83
16384	51	17503728.22	8.93E+08	1.785770	8.65	624384385	820271	99.87
HAMMING256								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	403277.62	2.65E+09	5.319240	12.43	1917261577	30080740	98.43
256	3290	666201.61	2.19E+09	4.389980	9.86	1575890205	15635218	99.01
512	1645	1192725.12	1.96E+09	3.927340	8.83	1398791335	8121588	99.42
1024	822	2243955.33	1.84E+09	3.690890	8.35	1311946899	4457832	99.66
2048	411	4344147.55	1.79E+09	3.571990	9.55	1267177219	2560160	99.80
4096	205	8539242.51	1.75E+09	3.501770	10.62	1242084939	1652336	99.87
8192	102	16922130.33	1.73E+09	3.452610	10.52	1224995969	1101662	99.91
16384	51	33740964.92	1.72E+09	3.441990	11.23	1221030847	839935	99.93
HAMMING512								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	705273.57	4.64E+09	9.293760	9.13	3309539025	28926458	99.13
256	3290	1220871.49	4.02E+09	8.039670	8.61	2870269753	15022681	99.48
512	1645	2252802.64	3.71E+09	7.415040	8.95	2654761343	8348982	99.69
1024	822	4315212.34	3.55E+09	7.096070	8.28	2538091639	4546385	99.82
2048	411	8440008.05	3.47E+09	6.938810	10.03	2479436443	2590168	99.90
4096	205	16679988.60	3.42E+09	6.839480	9.76	2444825603	1629082	99.93
8192	102	33169916.58	3.38E+09	6.767150	8.74	2418538923	1102923	99.95
16384	51	66194928.76	3.38E+09	6.752260	10.32	2413292989	868389	99.96

MPCS module

HAMMING16								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	190392.14	1.25E+09	2.578000	18.91	9.23E+08	26182	100.00
256	3290	187166.68	6.16E+08	1.296020	19.48	4.64E+08	28970	99.99
512	1645	185994.23	3.06E+08	0.671875	22.90	2.37E+08	26522	99.99
1024	822	194844.87	1.60E+08	0.377991	35.43	1.34E+08	26964	99.98
2048	411	358821.97	1.47E+08	0.352509	43.27	1.29E+08	26386	99.98
4096	205	707762.01	1.45E+08	0.346619	53.03	1.25E+08	27134	99.98
8192	102	1404021.81	1.43E+08	0.341614	50.78	1.23E+08	25264	99.98
16384	51	2821065.61	1.44E+08	0.341614	57.43	1.25E+08	24941	99.98
HAMMING32								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	190512.19	1.25E+09	2.579590	18.08	9.25E+08	25927	100.00
256	3290	186604.25	6.14E+08	1.291990	18.40	4.65E+08	25502	99.99
512	1645	191214.70	3.15E+08	0.689270	24.94	2.47E+08	26541	99.99
1024	822	312591.79	2.57E+08	0.572082	43.70	2.06E+08	35534	99.98
2048	411	614696.28	2.53E+08	0.562653	60.52	2.02E+08	26556	99.99
4096	205	1214288.31	2.49E+08	0.554138	59.27	2E+08	47109	99.98
8192	102	2409581.93	2.46E+08	0.546998	68.74	1.98E+08	24992	99.99
16384	51	4810967.51	2.45E+08	0.544952	63.46	1.99E+08	25485	99.99
HAMMING64								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	190157.73	1.25E+09	2.576450	18.29	9.24E+08	26063	100.00
256	3290	189571.39	6.24E+08	1.312680	20.98	4.72E+08	26097	99.99
512	1645	293968.98	4.84E+08	1.028350	41.56	3.7E+08	25519	99.99
1024	822	570996.91	4.69E+08	0.997070	61.06	3.59E+08	26301	99.99
2048	411	1124747.47	4.62E+08	0.981720	65.36	3.54E+08	25570	99.99
4096	205	2227841.84	4.57E+08	0.969757	71.16	3.5E+08	25011	99.99
8192	102	4421332.87	4.51E+08	0.958038	73.41	3.47E+08	29142	99.99
16384	51	8794277.49	4.49E+08	0.951263	75.56	3.48E+08	25553	99.99
HAMMING128								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	193477.55	1.27E+09	2.618930	20.45	9.42E+08	25638	100.00
256	3290	292656.81	9.63E+08	1.991000	44.61	7.16E+08	26250	100.00
512	1645	557791.33	9.18E+08	1.896270	59.60	6.83E+08	25944	100.00
1024	822	1087716.07	8.94E+08	1.846770	67.68	6.66E+08	24346	100.00
2048	411	2147342.63	8.83E+08	1.823640	69.96	6.57E+08	25247	100.00
4096	205	4254302.51	8.72E+08	1.800900	77.18	6.51E+08	25893	100.00
8192	102	8445436.12	8.61E+08	1.778050	73.99	6.46E+08	26012	100.00
16384	51	16752138.06	8.54E+08	1.762790	76.33	6.47E+08	25740	100.00
HAMMING256								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	308238.63	2.03E+09	4.129270	47.00	1.49E+09	30262	100.00
256	3290	567610.31	1.87E+09	3.800170	61.21	1.37E+09	26202	100.00
512	1645	1085599.87	1.79E+09	3.632540	66.48	1.31E+09	27100	100.00
1024	822	2121226.58	1.74E+09	3.545750	70.95	1.28E+09	25978	100.00
2048	411	4187806.53	1.72E+09	3.500150	74.59	1.26E+09	26403	100.00
4096	205	8306135.75	1.70E+09	3.462370	75.53	1.25E+09	25264	100.00
8192	102	16488050.84	1.68E+09	3.418370	77.61	1.24E+09	25910	100.00
16384	51	32668615.29	1.67E+09	3.386020	81.33	1.24E+09	25791	100.00
HAMMING512								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	605102.38	3.98E+09	8.036440	61.08	2.89E+09	26250	100.00
256	3290	1117357.43	3.68E+09	7.417630	69.02	2.67E+09	35991	100.00
512	1645	2140950.45	3.52E+09	7.105560	72.89	2.56E+09	24482	100.00
1024	822	4184752.05	3.44E+09	6.939480	78.25	2.5E+09	25281	100.00
2048	411	8266480.98	3.40E+09	6.854490	80.08	2.47E+09	25213	100.00
4096	205	16402134.34	3.36E+09	6.781590	79.44	2.45E+09	26590	100.00
8192	102	32559853.12	3.32E+09	6.697450	79.81	2.43E+09	26573	100.00
16384	51	64464723.10	3.29E+09	6.629120	81.27	2.43E+09	26301	100.00

MPLIST module

HAMMING16								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	194327.72	1.28E+09	2.820190	47.71	1015210995	48252347	95.25
256	3290	214149.98	7.05E+08	1.574890	43.11	544037289	23765577	95.63
512	1645	257064.01	4.23E+08	0.963287	36.77	324884923	12585584	96.13
1024	822	346135.22	2.85E+08	0.660950	31.58	216479011	7024959	96.75
2048	411	523320.56	2.15E+08	0.509155	28.90	162946665	4253654	97.39
4096	205	880935.47	1.81E+08	0.433777	20.80	136192499	2862094	97.90
8192	102	1606852.20	1.64E+08	0.396180	23.40	122220041	2148006	98.24
16384	51	3108156.92	1.59E+08	0.381988	24.83	117905241	1733586	98.53
HAMMING32								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	209135.51	1.38E+09	3.017360	44.31	1059365201	45410621	95.71
256	3290	247909.46	8.16E+08	1.797030	37.43	615432877	23154080	96.24
512	1645	325715.14	5.36E+08	1.189030	29.94	400719707	12366187	96.91
1024	822	475661.96	3.91E+08	0.873901	25.77	293619925	7050947	97.60
2048	411	784638.23	3.22E+08	0.723359	20.85	238854011	4250659	98.22
4096	205	1392390.16	2.85E+08	0.642853	16.18	211998613	2901895	98.63
8192	102	2626626.33	2.68E+08	0.603302	15.78	197033781	2139248	98.91
16384	51	5136022.45	2.62E+08	0.588471	16.80	192581761	1747211	99.09
HAMMING64								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	250948.27	1.65E+09	3.565980	37.55	1231166017	45210334	96.33
256	3290	319736.01	1.05E+09	2.270200	31.46	788439389	23794135	96.98
512	1645	456237.04	7.51E+08	1.619200	22.62	562790239	12688098	97.75
1024	822	735463.35	6.05E+08	1.301090	17.69	447793855	7079970	98.42
2048	411	1295269.25	5.32E+08	1.143770	15.06	393232359	4454385	98.87
4096	205	2412885.61	4.95E+08	1.062960	11.24	362727071	2904439	99.20
8192	102	4660746.03	4.75E+08	1.018650	11.63	346421207	2169917	99.37
16384	51	9199912.61	4.69E+08	1.003330	12.06	341995807	1745051	99.49
HAMMING128								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	325704.68	2.14E+09	4.547910	28.77	1586797665	45499593	97.13
256	3290	454321.08	1.49E+09	3.156370	20.73	1107649739	23322397	97.89
512	1645	722282.27	1.19E+09	2.493530	15.14	874547341	12635144	98.56
1024	822	1256927.59	1.03E+09	2.158780	11.37	754391115	7062704	99.06
2048	411	2322024.88	9.54E+08	1.987880	9.75	694965423	4366439	99.37
4096	205	4456169.10	9.14E+08	1.900060	11.88	664787649	2994398	99.55
8192	102	8727556.83	8.90E+08	1.848450	9.71	645757789	2180278	99.66
16384	51	17326152.47	8.84E+08	1.831910	11.48	640814465	1744389	99.73
HAMMING256								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	471508.85	3.10E+09	6.468600	19.95	2279571293	44772220	98.04
256	3290	731859.54	2.41E+09	4.982480	13.93	1761342657	23304845	98.68
512	1645	1250943.17	2.06E+09	4.231600	10.13	1502089361	12644565	99.16
1024	822	2292184.59	1.88E+09	3.858640	9.92	1367630561	7057614	99.48
2048	411	4372538.27	1.80E+09	3.671390	8.36	1302875727	4393651	99.66
4096	205	8531502.83	1.75E+09	3.568820	9.65	1266303331	2960095	99.77
8192	102	16851880.12	1.72E+09	3.504030	11.71	1243641459	2171923	99.83
16384	51	33565365.43	1.71E+09	3.486420	8.44	1237535983	1777119	99.86
HAMMING512								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	768055.20	5.05E+09	10.370300	12.58	3725421189	46573304	98.75
256	3290	1285694.88	4.23E+09	8.624790	9.27	3073198009	23511613	99.23
512	1645	2312103.31	3.80E+09	7.722290	8.37	2759354489	12903626	99.53
1024	822	4363921.36	3.59E+09	7.265660	8.71	2598648847	7520930	99.71
2048	411	8467007.37	3.48E+09	7.038120	9.16	2515048475	4444132	99.82
4096	205	16675254.58	3.42E+09	6.909150	9.57	2468284329	2956625	99.88
8192	102	33096379.49	3.38E+09	6.818090	9.29	2439058847	2252374	99.91
16384	51	66011215.76	3.37E+09	6.795990	9.27	2430093799	1786209	99.93

RINGIO module

HAMMING16								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	416954.07	2.74E+009	4.64377	51.76	1667005364	52761470	96.83
256	3290	444016.64	1.46E+009	2.4967	51.30	870833460	26690039	96.94
512	1645	493843.56	8.12E+008	1.41574	47.08	487655112	13593213	97.21
1024	822	595780.92	4.90E+008	0.899719	40.13	294400364	7126773	97.58
2048	411	786839.1	3.23E+008	0.598785	34.58	196789588	3972403	97.98
4096	205	1182868.72	2.42E+008	0.463134	30.73	147869288	2251038	98.48
8192	102	1967257.53	2.01E+008	0.393738	23.05	122895768	1431916	98.83
16384	51	3602622.43	1.84E+008	0.362884	30.83	111708310	1009995	99.10
HAMMING32								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	432652.56	2.85E+009	4.84226	50.25	1734002626	52651037	96.96
256	3290	476472.03	1.57E+009	2.71332	44.87	964934482	26762731	97.23
512	1645	559774.35	9.21E+008	1.63171	40.97	572251414	13754333	97.60
1024	822	728804.69	5.99E+008	1.09003	29.58	371851780	7206451	98.06
2048	411	1044088.99	4.29E+008	0.809906	24.77	273692134	3952973	98.56
4096	205	1692289.91	3.47E+008	0.673188	20.30	223370932	2297960	98.97
8192	102	2978628.03	3.04E+008	0.601593	22.42	197390918	1452264	99.26
16384	51	5577497.73	2.84E+008	0.571961	15.32	186504662	1049230	99.44
HAMMING64								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	476402.22	3.13E+009	5.38772	45.00	1936433196	52796499	97.27
256	3290	548199.31	1.80E+009	3.18182	38.58	1125645776	27010583	97.60
512	1645	696067.51	1.15E+009	2.07571	30.74	729394502	13729247	98.12
1024	822	986077.75	8.11E+008	1.51605	22.70	526918122	7322573	98.61
2048	411	1560314.44	6.41E+008	1.23303	17.89	425506974	3951105	99.07
4096	205	2715816.56	5.57E+008	1.09189	14.74	373430408	2223837	99.40
8192	102	5016181.38	5.12E+008	1.01843	11.40	346345774	1386744	99.60
16384	51	9650337.71	4.92E+008	0.986175	13.03	334931764	964034	99.71
HAMMING128								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	554787.18	3.65E+009	6.39932	38.20	2290022890	52476008	97.71
256	3290	685368.4	2.25E+009	4.08838	30.06	1463485676	26773394	98.17
512	1645	960742.47	1.58E+009	2.94864	21.75	1050726798	13896135	98.68
1024	822	1503211.17	1.24E+009	2.36838	15.05	834188340	7273413	99.13
2048	411	2587020.62	1.06E+009	2.07733	12.39	729572202	3931290	99.46
4096	205	4755037.07	9.75E+008	1.92929	10.39	675292979	2274538	99.66
8192	102	9090577.04	9.27E+008	1.8486	12.65	646013490	1472170	99.77
16384	51	17773142.76	9.06E+008	1.8161	9.55	634492962	1055727	99.83
HAMMING256								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	704403.32	4.63E+009	8.38907	28.74	3018006766	53891988	98.21
256	3290	963627.35	3.17E+009	5.91327	21.04	2122437324	27388570	98.71
512	1645	1493309.83	2.46E+009	4.70309	14.85	1676468686	13852951	99.17
1024	822	2538123.63	2.09E+009	4.0701	11.42	1445845360	7260125	99.50
2048	411	4636463.3	1.91E+009	3.8613	9.57	1335649668	3970311	99.70
4096	205	8829772.48	1.81E+009	3.59998	9.24	1276642012	2277778	99.82
8192	102	17252610.85	1.76E+009	3.51239	11.02	1244957654	1488763	99.88
16384	51	34042360.61	1.74E+009	3.47546	8.67	1230767188	1038240	99.92
HAMMING512								
Chunk size	**Num. of Chunks**	**Chunk's Troundtrip [ticks]**	**Total Ttransfer [ticks]**	**Texecution [s]**	**GPP load**	**DSP total cycles**	**DSP idle cycles**	**DSP load**
128	6580	999587.11	6.58E+009	12.2951	19.86	2924528800	53115636	98.18
256	3290	1517370.7	4.99E+009	9.54672	14.06	3429206636	27349413	99.20
512	1645	2549574.93	4.19E+009	8.17783	10.60	2924528800	13786012	99.53
1024	822	4608922.52	3.79E+009	7.47556	9.10	2672855612	7306186	99.73
2048	411	8730983.2	3.59E+009	7.12933	9.79	2549536872	4009285	99.84
4096	205	16975508.28	3.48E+009	6.94052	8.57	2480221102	2333066	99.91
8192	102	33512425.31	3.42E+009	6.82962	8.69	2439137764	1479954	99.94
16384	51	66505459	3.39E+009	6.78674	17.52	2423698156	1050570	99.96

Appendix F

Data of NEON performance

GPP only

Filter taps	Texecution [s]	Texecution [clock ticks]	Average GPP load [%]
16	0.157654	78818658	57.07
32	0.280274	140138328	57.24
64	0.526581	263303458	59.43
128	1.017580	508778136	59.52
256	1.997500	998753648	59.69
512	3.962650	1981338036	59.32

NEON vectorizing compiler

Filter taps	Texecution [s]	Texecution [clock ticks]	Average GPP load [%]
16	0.108216	54097447	53.74
32	0.194611	97293939	58.01
64	0.380890	190446699	59.11
128	0.657135	328570140	59.11
256	1.365690	682852711	58.70
512	2.465760	1232882635	59.40

NEON intrinsics

Filter taps	Texecution [s]	Texecution [clock ticks]	Average GPP load [%]
16	0.127960	63982554	52.15
32	0.118073	59030707	52.21
64	0.172333	86163064	59.65
128	0.271942	135972978	58.83
256	0.460968	230482044	59.27
512	0.844269	422124518	59.15

NEON assembly

Filter taps	Texecution [s]	Texecution [clock ticks]	Average GPP load [%]
16	0.109619	54795042	53.47
32	0.130951	65470395	57.63
64	0.186401	93185622	56.85
128	0.281799	140906318	55.48
256	0.473175	236585713	58.94
512	0.858246	429134731	59.76

Bibliography

[1] SDR forum. What is Software Defined Radio. URL http://www.sdrforum.org/pages/aboutSdrTech/SoftwareDefinedRadio.pdf.

[2] Manuel Uhm, David Hawke, and Mark Quartermain. SDR comes of age: Technology meets economics. In *SDR'09 Technical Conference and Product Exposition*, December 2009. Washington, DC, USA.

[3] Geoffrey A. Moore. *Crossing the Chasm : Marketing and Selling Disruptive Products to Mainstream Customers*. Harper Business Essentials, 1991. ISBN: 0060517123.

[4] Christofer R. Anderson, George Schaertl, and Philip Balister. A low-cost embedded SDR solution for prototyping and experimentation. 2009. Wireless Measurements Group, United States Naval Academy, Annapolis.

[5] Philip Balister. A software defined radio implemented using the OSSIE core framework deployed on a TI OMAP processor. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, December 2007.

[6] John Bard and Vincent J. Kovarik Jr. *Software Defined Radio: The Software Communications Architectures*. John Wiley & Sons, 2007. ISBN: 0470865180.

[7] Jeffrey H. Reed. *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall PTR, 2002. ISBN: 0130811580.

[8] Eric Blossom. Exploring GNU Radio, November 2004. URL http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html.

[9] Hassan Farhat. A single cycle floating-point unit. University of Nebraska at Omaha, Omaha, NE.

[10] Ettus Research. USRP2: The next generation of software radio systems. URL http://www.ettus.com/downloads/ettus_ds_usrp2_v5.pdf.

[11] OSSIE: SCA-based open source software defined radio. URL http://ossie.wireless.vt.edu/?q=node/12#cf.

[12] Safadi, M.S., Ndzi, and D.L. Digital hardware choices for software radio (SDR) baseband implementation. volume 2, pages 2623 –2628, 2006.

[13] Anqi Luo and Lei Ge. Indoor location detection using WLAN. Master's thesis, Royal Institute of Technology (KTH), School of Information and Communications Technology, Stockholm, Sweden, January 2010. URL http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/100111-Anqi_Luo_and_Lei_Ge-with-cover.pdf. TRITA-ICT-EX-2009:209.

[14] *BeagleBoard System Reference Manual - Revision C3*, May 2009. Literature reference: BB_SRM.

[15] Texas Instruments. *OMAP3530/25 Applications Processor*, February 2008. Literature reference: SPRS507F.

[16] Brian Carlson and Bill Giolma. SmartReflex power and performance management technologies: reduced power consumption, optimized performance, February 2008. URL http://focus.ti.com/pdfs/wtbu/smartreflex_whitepaper.pdf.

[17] Texas Instruments. *OMAP35x Applications Processor: Technical Reference Manual*, April 2010. Reference: SPRUF98K. Revised October 2010.

[18] ARM. Cortex-A8 processor. URL http://www.arm.com/products/processors/cortex-a/cortex-a8.php. Last access: 22nd October 2010.

[19] ARM. NEON. URL http://www.arm.com/products/processors/technologies/neon.php. Last access: 22nd October 2010.

[20] Texas Instruments. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, July 2010. Literature Number: SPRU732J.

[21] Texas Instruments. *TMS320C64x+ IQmath Library User's Guide*, December 2008. Reference: SPRUGG9.

[22] *TMS320C64x+ DSP Little-Endian DSP Library Programmer's Reference*, March 2006. Literature Number: SPRUEB8B.

[23] Texas Instruments. *TMS320 DSP/BIOS User's Guide*, November 2002. Literature Number: SPRU423B.

[24] Texas Instruments. *DSP/BIOS LINK User Guide - Version 1.65.00.02*, March 2010. Reference: LNK 058 USR.

[25] Texas Instruments. *DSP/BIOS LINK Zero Copy Link Driver - Version 0.90*. Reference: LNK 041 DES.

[26] Texas Instruments. *DSP/BIOS LINK Programmer's Guide - Version 1.65.00.02*, March 2010. Reference: LNK 161 USR.

[27] Tore Ulversøy and Jon Olavsson Neset. On workload in an SCA-Based System, with varying component and data packet sizes. URL http://ftp.rta.nato.int/public/PubFullText/RTO/MP/RTO-MP-IST-083/MP-IST-083-03.doc. NATO Research and Technology Organisation (RTO) Information Systems Technology Panel (IST) Symposium held in Prague, Czech Republic, on 21-22 April 2008. ISBN 978-92-837-0065-4.

[28] Building Angström. URL http://www.angstrom-distribution.org/building-angstrom. Last access: 29th November 2010.

[29] Debian. DebianWiki - ArmEabiPort. URL http://wiki.debian.org/ArmEabiPort#Inanutshell.

[30] Andres Calderon and Nelson Castillo. Why ARM's EABI matters. March 2007. URL http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Why-ARMs-EABI-matters/.

[31] Texas Instruments. *DSP/BIOS LINK - OMAP3530 EVM - Installation guide v.1.65*, July 2010. Reference: LNK 173 USR.

[32] IBM Corporation and Microsoft Corporation. *Multimedia Programming Interface and Data Specifications 1.0*, August 1991.

[33] Jonathan Y. Stein. *Digital Signal Processing: A Computer Science Perspective*. John Wiley & Sons, 2000. ISBN: 0471295469.

[34] Texas Instruments. Cortex-A8 NEON architecture. URL http://processors.wiki.ti.com/index.php/Cortex-A8_Neon_Architecture.

[35] ARM. Architecture and implementation of the ARM Cortex-A8 microprocessor. URL http://arch.eece.maine.edu/ece471/images/2/26/TigerWhitepaperFinal.pdf.

[36] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition - Errata markup*, October 2010. ARM DDI 0406B_errata_2010_Q3 (ID100710).

[37] ARM. *RealView Compilation Tools Version 4.0 - Assembler Guide*, January 2009. ARM DUI 0204I (ID100419).

[38] Pete Cockerell. *ARM assembly language programming*. Computer Concepts Ltd. URL http://www.peter-cockerell.net/aalp/.

[39] debianWiki. ARM hard float port - VFP comparison. URL http://wiki.debian.org/ArmHardFloatPort/VfpComparison.

[40] Elana Granston. *Hand-Tuning Loops and Control Code on the TMS320C6000*, August 2006. Reference: SPRA666.

[41] F. Duarte and S. Wong. A memcpy hardware accelerator solution for non cache-line aligned copies. In *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 397 –402, 2007.

[42] ARM Technical Support Knowledge Articles. What is the fastest way to copy memory on a Cortex-A8? URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13544.html.

TRITA-ICT-EX-2011:21

www.kth.se