View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO DE FIN DE CARRERA

TITULO DEL TFC: Modelado y gestión de recursos de SDR Clouds

TITULACION: Ingeniería Técnica de Telecomunicaciones, especialidad en Telemática

AUTOR: Juan Pedro Vicente Núñez

DIRECTOR: Vuk Marojevic

FECHA: 9 de enero de 2012

Title: Modeling and resource management of SDR Clouds

Author: Juan Pedro Vicente Núñez

Director: Vuk Marojevic

Date: January, 9th 2012

Overview

This TFC models software-defined radio (SDR) processing chains and manages their distributed execution on SDR clouds. We present models of UMTS processing chains for different bit rates. These chains model the digital processing requirements of a transmitter and receiver in the base station. Therefore, we analyze the users multiplexing in WCDMA because it has implications on the computing resources and its correct management.

We propose different management strategies, implement one and evaluate its correct functionality. The solution we promote is based on defining a ghost function that captures the data flow dependencies between the chip-level and bit-level processing part of WCDMA. The chip-level processing chain processes signals of multiple users, whereas the bit-level processing chain processes a single user's signal. The simulations are executed with an appropriate user load, which is part of our investigation.

For the development of this TFC we are provided of a mapping algorithm for the computing resource management in a simulation environment. This algorithm runs a trellis evaluating the costs of the different paths during the mapping process of the signal processing blocks of a processing chain to the distributed computing resources of the multiprocessors platform.

INDEX

INTRODUCTION 6

CHAPTER 1. RESOURCE MANAGEMENT FOR SDR CLOUDS..................... 8

1.1. Introduction 8

1.2. The SDR Cloud... 8

1.3. Enabling Technologies 10
1.3.1. Middleware 10
1.3.2. Virtualization 10
1.3.3. Resource Allocation... 10

1.4. ALOE 11

CHAPTER 2. PROCESSING CHAIN MODELING.. 13

2.1 UE receiver.. 13
2.1.1. Symbol Level 104
2.1.2. Bit Level 105
2.1.3. MOPS calculation 107
2.1.3. Mbps calculation 10

2.2. BTS transmitter... 17

CHAPTER 3. WCDMA CAPACITY... 21

3.1. Introduction 21

3.2. Spreading and Scrambling 21

3.3. Network Capacity measurements.. 25

CHAPTER 4. COMPUTING RESOURCE MANAGEMENT FRAMEWORK.... 26

4.1. Computational load distribution 26

4.2 Mapping process: tw-mapping and the cost function 26
4.2.1. The tw-mapping 26
4.2.2. The cost function 28

CHAPTER 5. RESOURCE MANAGEMENT WITH DATA DEPENDENCY CONSTRAINTS 30

5.1. Data dependecy in the Base Station 30

5.2. Solutions... 31
5.2.1. The constraint vector solution.. 31
5.2.2. The ghost function solution.. 32
5.2.3. Implementation 33

CHAPTER 6. TESTING AND SIMULATIONS 35

6.1. Testing the program 35

6.2. Output analysis 35

6.3. Minimum number of processors 37

6.4. Processing resource saving 39

CONCLUSIONS 40

BIBLIOGRAPHY 41

ANNEX 42

ABBREVIATIONS

3GPP	3rd Generation Partnership Project
BTS	Base Transmitter Station
CCTrCH	Coded composite transport channel
DPDCH	Dedicated physical data channel
DTX	Discontinuous Transmission
MS	Mobile Station
Node B	3GPP term for a base station
PhCH	Physical Channel
UE	User Equipment
TF	Transport Format
TrCH	Traffic Channel

INTRODUCTION

This TFC is about computing resource modeling and management for SCR Clouds. The SDR Cloud concept is born from two other concepts, *software-defined radio* (SDR) and *cloud computing*.

In the field of software radio, the processing chains are implemented in software and executed upon a generic multiprocessing environment. This facilitates the dynamic reconfiguration from one standard to another. In a Base Station (BS), with SDR waveforms for several users, this implies the possibility of reconfiguration of an ensemble of processors. In the SDR field, a waveform is a processing chain implemented in software pertaining to a transmitter or receiver.

Moreover, it is expected that the future BS will consist on distributed antennas with analogical processing and associated AD/DA converters, while the digital processing will be done in a common data center.

The progresses in networking facilitate the data to be moved from the antenna to the data center by optical fiber and to be processed remotely instead of in the proximity of the antenna.

Cloud computing is a marketing term for technologies that provide computation, software, data access, and storage services that do not require end-user knowledge of the physical location and configuration of the system that delivers the services. A parallel to this concept can be drawn with the electricity grid, wherein end-users consume power without needing to understand the component devices or infrastructure required to provide the service.

The SDR Cloud concept merges the SDR and Cloud Computing concepts. The advantages of the SDR Cloud are many: flexibility, scalability, shared and reusable resources, amongst others. Since the data center is conneted to several antennas providing radio service to a relatively wide area corresponding to the combined range of all the connected antennas to the SDR Cloud, this one will execute a high number of waveforms.

A key part for all of this is therefore the management of the computing resources. This management has to guarantee the processing in real-time depending also on the service and quality of service required.
The resource manager has to assign waveforms to processors in real-time. The complexity of the waveforms to provide more sofisticated services (e.g. voice, data, streaming, defined as the fundamental 3G services), requires a distributed execution. This means the processing blocks composing a waveform have to be mapped on a group of processors. For this, it is necessary to apply efficient strategies for the optimum management of distributed computing resources.

In essence, this TFC is divided in the following chapters:

Chapter 1: This chapter introduces more in detail the SDR Cloud, what it does, and its capabilities. We also introduce the tools with which we will work.

Chapter 2: In chapter 2 we focus on the UE receiver and the BTS transmitter processing chains, analyzing them in detail.

Chapter 3: In chapter 3 we analyze WCDMA capacity, both theoretical and experimental.

Chapter 4: In this chapter we explain the basics of the tw-mapping employed in the data center with some examples.

Chapter 5: In this chapter we create and modify the CRM framework so it can work simulating a BTS receiver.

Chapter 6: In this final chapter, we test our new code, provide some examples of how it could work and analyze the results.

CHAPTER 1. RESOURCE MANAGEMENT FOR SDR CLOUDS

1.1 Introduction

Nowadays, wireless networks are everywhere thanks to the improvements during recent years on transmitter, receiver and codification technologies. On top of these new advanced technologies, the expansion of base stations and antennas has dramatically increased the system complexity in exchange of a better spectral efficiency, enabling an increased wireless traffic.

For example, a new trend for the future is working with both the MIMO antenna systems and DAS (distributed antenna systems). In radio, MIMO, multiple-input and multiple-output, is the use of multiple antennas at both the transmitter and receiver to improve communication performance. DAS employ optical fiber or coaxial cable to connect an antenna system to the processing unit over relatively large distances. In these experiments the signal processing tasks are performed by a data center which controls several antennas that act as a base station, distributed across the cell.

Traditional base stations are built with equipment that serves for a specific use, as defined at design time. Little features are available for reprogramming them or modifying them without substitute the devices physically. Software-defined radio changes this by implementing the transceiver signal processing chains in software that is run on general-purpose hardware, enabling dynamic reprogrammability. The newest opinions and ideas about radio signal processing indicate that a system based on data centers where general-purpose processors execute the signal processing functions. .

This chapter breifly introduces the SDR cloud and its features, and explains the system's software [15].

1.2 The SDR Cloud

The SDR system has many advantages but also some disadvantages too. The most important one is the massive amount of computing resources the system needs, especially during peak hours. Even with the advances in new technologies this problem is still very relevant, additionally, the demand from the users increases every year.
This disadvantage in not specific for SDR-systems; traditional base stations also need to provide computing resource for the expected peak load. SDR introduces flexibility, but also some resource overhead, which is always present for software implementations.

A solution to this problem could be to pay for the necessary infrastructure that would be needed when the system was at its higher level of demand. Of course, this does not seem like a good solution because most of the system would be inactive while the user demand is low.

This is when Cloud computing enters in scene: it can provide computing resources on demand. Although already defined in the first introduction, now we will define it in more appropriate terms that relate better to this work: it is a type of parallel and distributed system consisting of a group of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers [16].

All this enables definable services at the data center available from anywhere that were not possible previously. These services can be given thanks to the SDR capacities in terms of platform-independent software development and seamless system reconfiguration (the users do not perceive anything when software changes are made on the data center).
This leads us to the need of a framework that facilitates the deployment of the different applications and provides real-time execution control and dynamic resources management. More details in 1.4.

We have talked about the disadvantages and how they were solved, now we will talk about the main advantages of the SDR Clouds.

One of the most important ones is the efficiency. The data center processes the data that come from several base stations, each one giving service to their respective cell. SDR Clouds improve efficiency in terms of computing resources. For us, computing resource efficiency is proportional to the active processing time versus idle time. Base stations are designed with the resources needed to give service to a certain maximum number of users, and part of these resources wasted if the system is not working at its full capacity. The SDR Cloud does not have this problem because it provides the necessary resources on demand. When the wireless traffic is low, the SDR Cloud can assign the idle computing infrastructures to other uses.

Another important advantage is the long-term adaptability. While radio access technologies evolve, the base stations will also have to be remodeled. With the SDR Clouds, that is not necessary, since we only need from the base stations the antennas and the conversion of the signal from digital to analog (and vice-versa). The only needs of hardware updates will be in the data center.

Today's wireless access infrastructure makes sharing the available resources a difficult task. On the other hand, SDR platforms are flexible because they can execute any existent waveform.
Capacity also increases with SDR Clouds. The key here is the increase in the signal processing overhead, reducing the necessary spectral resources.

Increasing the signal processing does not represent an inconvenience because of the almost unlimited SDR computing resources [15].

1.3 Enabling technologies

1.3.1 Middleware

As a general definition, the middleware is a software that assists an application to interactuate or communicate with other applications, software, networks, hardware and/or operative systems. It simplifies the work the programmers have to do to generate the necessary connexions in the distributed systems. In the SDR Cloud, specifically, its role is to communicate services with components or processes running in different computers.

One of the most important features an SDR Cloud has to accomplish is the real-time necessities in the services. This implies a very fast processing time and very low latency between the data center and the end user. SDR applications are modular processing chains, and the middleware helps in the communication of these modules, linking the interfaces of the components.

1.3.2 Virtualization

Virtualization enables resource sharing between different users accessing the same physical resources. Virtualization means resource overhead (CPU time, memory, etc.). Different virtualization levels exist. For SDR clouds, having very tight timing and resource constraints, the minimum level of virtualization should be approached, providing just the necessary benefits at a reasonable efficiency. We suggest compiling SDR applications for a specific processor architecture (or for several architectures if necessary), this way being able to meet the application's hard real-time constraints.

Abstraction techniques enable isolating the software from the hardware. Software can then be written without knowledge of the hardware and can be ported to different platforms. The SDR cloud needs to provide an application programming interface (API) so that the programmer can interact with the hardware (devices, communication links, and other services).

Again, a good tradeoff between flexibility (abstract *everything*) and efficiency (provide the minimum level of abstraction) should be found for maintaining a good balance between efficiency and flexibility [15].

1.3.3 Resource allocation

Processors need to be shared between different clients, that is, radio operators, who rent computing resources from the infrastructure or cloud operator. Therefore, mechanisms are needed to ensure that each client gets the required amount of resources. The resource manager therefore allocates resources as demanded by the users (cloud clients). General-purpose computing clouds or grids typically apply best effort or other types of policies, sometimes accounting for real-time deadlines, which are orders of magnitude longer than in SDR.

SDR applications are modular processing chains of heterogeneous processing and data flow demands. Since the application is specific (signal processing) certain operations are very frequent and employed in several algorithms. Typical SDR processing platforms combine general-purpose with special-purpose processors (for example, GPPs and DSPs). Arrays of small processors are combined to achieve high performance parallel processing platforms. The resource allocation approach needs to be able to deal with heterogeneous platforms and applications and manage multiple resource constraints. These constraints are a function of the service, QoS, and possibly other internal and external system factors [15].

1.4 ALOE

The abstraction layer and operating environment (ALOE) is an open source SDR framework, which is presented in [17]. Since ALOE is not part of this project, but its existence is still relevant for what is developed here, we summarize its main features below [15].

ALOE provides just the necessary virtualization mechanisms and a limited number of customizable services for an efficient, though portable, execution of SDR applications. ALOE was originally designed for clusters of tightly-coupled processors. Here we present its virtualization and resource allocation capabilities adapted to SDR clouds.

ALOE divides the execution time in discrete *time slots* and executes the SDR application in a pipelined fashion. The time slot duration d_{TS} (in seconds) needs to be specified as a function of the real-time computing constraints and the components' execution periods. It typically takes in the range of hundreds of microseconds or a few milliseconds. Each time slot, the middleware checks that all real-time constraints have been met. In case of a real-time violation, the middleware stops the application or releases the processor ownership.

The resource allocation begins with a coherent platform and application modelling. The framework proposed in uses the multiply-accumulate operation (MAC) as an abstract measure of a processing resource. With this basic digital signal processing operation, the metric of million operations per second is used to coherently specify the application component's processing demands and the platform processing capacities. Interprocessor bandwidth capacities and data

flow demands are measured in units of mega bits per second (Mbps). Once a suitable time slot is defined, we obtain the application-specific metrics million operations per time slot (MOPTS) and megabits per time slot (MBPTS) as: MOPTS = $d_{TS} \cdot$ MOPS and MBPTS = $d_{TS} \cdot$ Mbps.

The ALOE resource management framework features a resources allocation algorithm with a customizable cost function (see Chapter 4).

Hard real-time processing implies throughput and latency requirements. An application's throughput requirement can be met by finding a feasible resource allocation or mapping; that is, a mapping that is able to allocate the necessary amount of computing resources as demanded by the application. The end-to-end latency requirement of the SDR application (executed in a pipelined fashion) can be accomplished by correctly specifying the time slot duration d_{TS} so that the number of pipelining stages n_{ts} times the time slot duration is within the latency limit $L_{\max}$ [5]:

$$d_{TS} = L_{\max} / n_{ts}.$$

$L_{\max}$ is the maximum tolerable latency for a given radio service.

CHAPTER 2. PROCESSING CHAIN MODELING

This chapter explains the WCDMA downlink, es employed for the UMTS/3G standard. The goal is to model simplified UE receiver and BTS transmitter's processing chains at the physical layer, its modules, and its processing and bandwidth requirements. We will then use one of these models to simulate users asking for 3G access in the SDR Cloud and analize different computing resource management strategies.

The UE receiver model at the physical layer consists on the processing blocks of the mobile terminal receiver, divided in the chip-rate processing and bit-rate processing levels. Next we explain the BTS transmitter model. It is important to differentiate between these two processing blocks because in the former, the user signals are all processed at once, while in the second, the signal of each user is processed one by one. The SDR Cloud data center processes the chip-rate chain only once for all the users transmitting in the same frequency band, while the bit-rate processing chain is user-specific. This way, we save a large amount of computing resources since we will not need to allocate computing resources for the chip-rate processing chain for every user.

2.1 UE Receiver

In this section we present the UE recevier processing model.We describe all the components, with special emphasis on the most important ones, as shown in figure 2.1. We assume a commercial analog to digital converter working at 65 Mhz with an output of 65 Mhz * 16 bits per sample = 1040 Mbps.

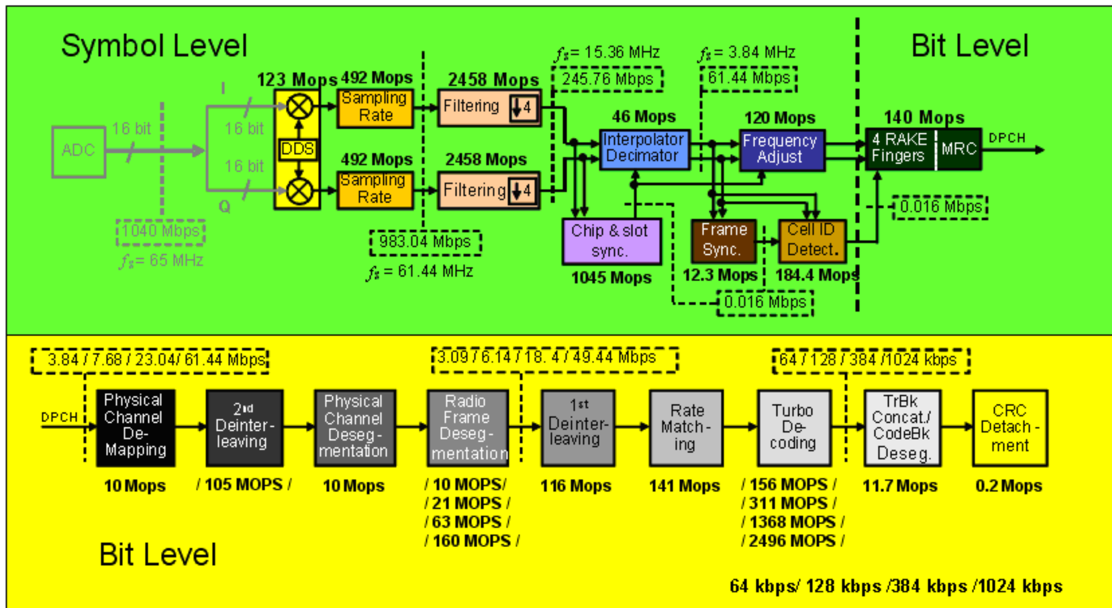

Figure 2.1: UE Receiver

2.1.1 Symbol Level

Digital Downconverter (DDC) – The signal is downconverted to baseband.

Sampling Rate adjustment – Takes samples at a rate of 1040 Mbps. Output becomes 983 Mbps.

Filtering – The main function of this filter block is the radio resource controller filtering of the received signal to achieve an aggregate raised cosine response when paired with the transmit filter in the Node B**.**

Interpolator Decimator – An interpolator is used to increase the sampling rate, that is, to add more samples to the data stream. The decimator is used to reduce the sampling rate when the desired bandwidth is lower than the sampling rate implies.

Chip & slot sync. – If a frequency reference is used and is not capable of being adjusted to achieve frequency lock with the Node B, a frequency shifter is used to rotate continuously the phase of the received vector to compensate for the frequency error experienced in the mobile terminal.

Frame sync – Synchronization with the beginning of the 10 ms frames.

Frequency Adjust – Adjusts the frequency to the one expected by the Rake receiver.

Cell ID Detect – Detects the primary scrambling code. Descrambles with each of the 8 possible scrambling codes, correlates, sums each correlation output, and compares the sums.

2.1.2 Bit Level

4 Fingers Rake – The RAKE architecture is a suboptimal but efficient implementation of a CDMA receiver. Figure 2.2 shows the components of a RAKE receiver: a set of fingers and a combiner block. Each single RAKE finger is an independent receiver for the signal from a specific cell on a specific propagation path. Multipath propagation channels and soft handover situations are handled by employing multiple fingers, one for each propagation path in question. The output symbols of all the fingers are then combined coherently and synchronously by the RAKE receiver's combiner block to yield the received data symbols.

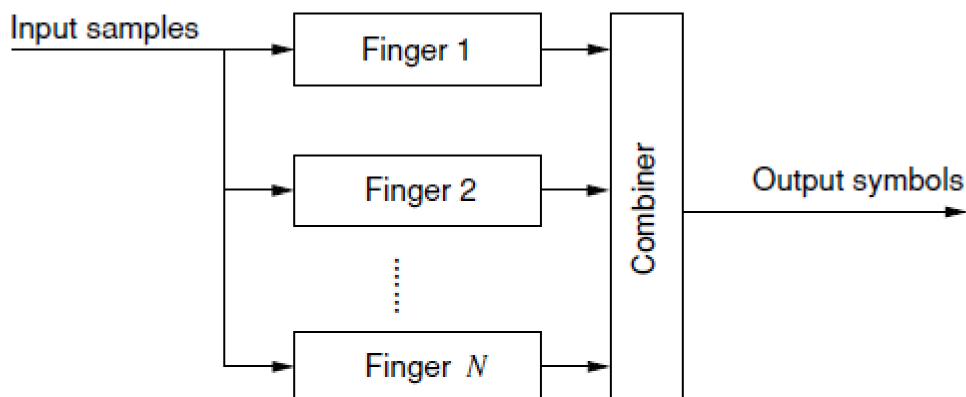

Figure 2.2: Key RAKE receiver building blocks

Each RAKE finger independently receives the signal contribution of a single propagation path from a given cell. It performs this by correlating the received signal sample stream over a length of SF samples with a time aligned copy of the scrambling and spreading codes. At its input, the RAKE finger takes input samples at chip rate, and it outputs the correlation results at symbol rate.

The time and code selectivity of the fingers is achieved by exploiting the cross- and autocorrelation of the combined spreading and scrambling codes over SF samples. While the cross-correlation between codes is nominally independent from their relative time delay, the correlation between the incoming samples and the local copy of the codes not only depends on the code's autocorrelation function, but also on the sampling position within the pulse shaping filter's impulse response.

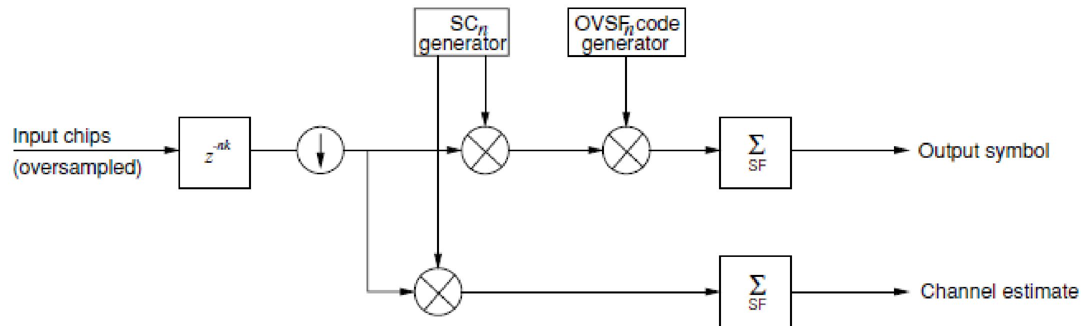

Figure 2.3: Simplified diagram of a RAKE finger.

Figure 2.3 shows a simplified schematic of a RAKE finger, consisting of:

- a data correlator branch, consisting of two multipliers for applying the spreading and scrambling codes and an accumulator of length SF.
- a channel estimation branch receiving the CPICH, consisting of a scrambling code multiplier and an accumulator of length 256.
- an input sample delay memory block.
- a downsampling block, which reduces the rate of the input signal into the data and CPICH correlators to chip rate.
- the scrambling and spreading code generators.

Physical Channel Demapping – The physical channel is demapped.

2nd Deinterleaving – Second interleaving stage, which is called the PhCH deinterleaver, operates on the multiplexed bits from the TrCH.

Physical Channel Desegmentation – Desegmentation is done depending on the total number of bits and the number of PhCHs in the transmitter.

Radio Frame Desegmentation – In the transmitter, when the transmission time interval is longer than 10 ms, the input bit sequence is segmented and mapped onto consecutive radio frames. This does the reverse process.

1st Deinterleaving – Reverse process of the Interleaving done in the transmitter to split up the errors at the receiver.

Rate Matching – Bits of the transport channel are repeated or punctured.

Turbo Decoding – Decodes the transmitted signal at a coding rate of ½ or 1/3.

Desegmentation – Bit desegmentation.

CRC Detatchment – Detatchment of the CRC used for error detection[3].

2.1.3 Million Operations Per Second (MOPS) calculation

Figure 2.1 also shows the processing and data flow requirements in Million Operations Per Second (MOPS) and Mega-bits Per Second (Mbps), respectively. The calculation of the operations per second for each of the waveform's modules is the following. The processing algorithms are analyzed in terms of multiply-accumulates operations (MACs) per sample, multiplied by the quantity of samples that enter the module per second.

For example, given a FIR filter of 64 coefficients with a 1 MHz entry rate, it means we will need 64 operations plus the register update. Neglecting the complexity of the register update, the approximate processing complexity of this filter becomes:

$$64 \times 1 \text{ MHz} = 64 \text{ MOPS}$$

2.1.4 Mega-Bits Operations Per Second (Mbps) calculation

The calculation of the bandwidth necessary from one function to another is the following. For example, to calculate the bandwidth between the A/D converter and the DDS, we use an appropriate 16-bits converter with 65 MHz sampling frequency. The bandwidth requirement will be:

$$16 \times 65 \text{ Mbps} = 1040 \text{ Mbps}$$

2.2 BTS Transmitter

In this section we will present the base station transmitter model. We will describe all the components, with special emphasis on the most important ones (figure 2.4).

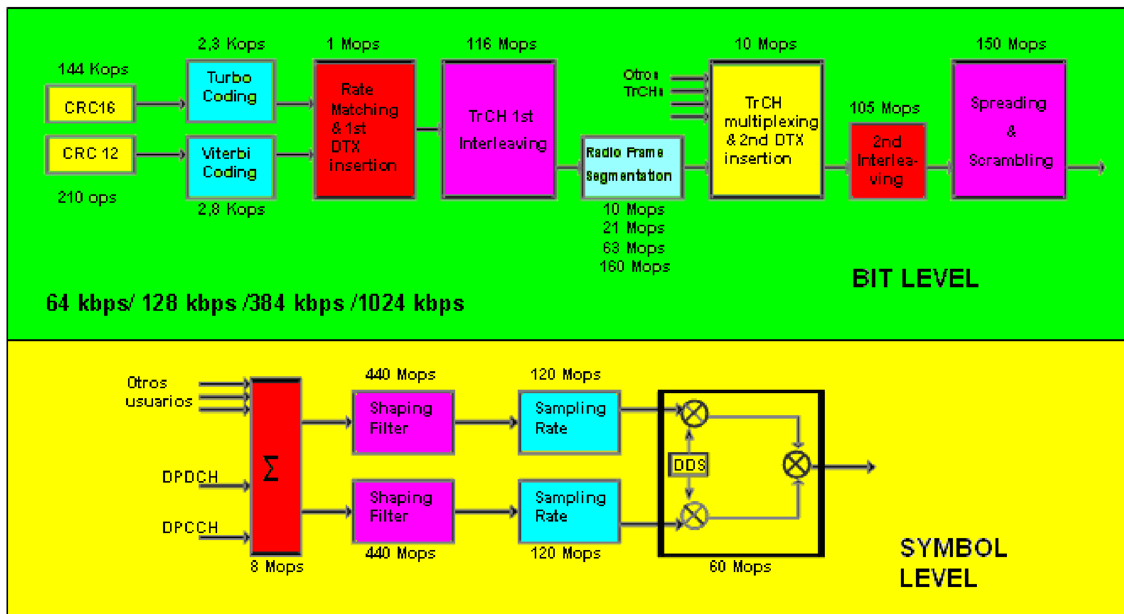

Figure 2.4: BTS Transmitter

CRC Attachment – *Cyclic Redundancy Check* (CRC) codes are used for error detection on decoded transport blocks. They are similar to the well known and very simple parity checking codes, in which a single check bit is added to a group of data bits in order to make the total number of 1s in the set of bits either even (for even parity codes) or odd. Then, using even parity codes as an example, if the number of 1s received in a block of data at the receiver is odd, the receiver will know there has been an error, so all single bit errors will be spotted by a parity check. However, if there is more than one bit error, there is a 50% chance the error will be undetected at the receiver.

CRC codes are an extension of these parity checking codes. Instead of adding just one bit to a block of data, several bits are added. This gives the ability to reliably detect longer error sequences, and a lower chance of undetected errors when the error detection capability of the code is exceeded.

Channel coding – Three types of channel coding are specified for UMTS, either half or one third rate convolutional encoding or one third rate turbo encoding. Turbo coding gives an improvement in performance over convolutional coding, especially for long block lengths. Support of convolutional encoding and decoding is mandatory in UMTS terminals, whereas support of turbo encoding and decoding is an option. Typically, convolutional encoding is used for control channels and low rate channels such as AMR (Adaptive multi rate speech codec) speech channels, and turbo encoding is used for high rate channels in higher capability terminals.

Rate matching – Rate matching provides a mechanism to map the encoded data bits from each TrCH on to the available PhCH resource. At the encoder this means either puncturing or repeating some encoded bits, depending on whether the number of bits available on the PhCH is greater or less than the

number of coded bits. At the decoder these operations are reversed. For bits that have been repeated, the received soft decisions are combined to give a more reliable indication of the value of the bit. For bits that have been punctured at the encoder, soft decisions that indicate we have no information regarding the value of the bit are inserted in their place at the decoder.
Excess PhCH capacity is filled using discontinuous transmission (DTX) indicators when a lower rate TF (for fixed TrCH positions) or TFC (for flexible TrCH positions) is used. No power is transmitted from the Node B for this CCTrCH for DTX indicators, and the UE will receive effectively only noise for these bits.

TrCH 1st Interleaving – Interleaving is used to split up the errors at the receiver which, because of the nature of mobile radio channels, tend to occur in bursts. At the receiver, the deinterleaving splits up these bursts of errors, which improves the performance of channel decoding. Typically, best performance is achieved by interleaving over as long a period as possible, in order to split up errors as widely as possible. So each TrCH is interleaved over its whole TTI.

Also, bits from different TrCHs should be interleaved amongst each other, as each TrCH is individually decoded. For these reasons, two stages of interleaving are used in UMTS. The first, which we call the TrCH interleaver, operates individually on each TrCH over a whole TTI. The second, which we call the PhCH interleaver, operates over a frame on the data for that frame from all TrCHs.

Radio frame segmentation – The total number of bits is split into a set for each radio frame. The rate matching will have ensured that the number of bits, including any DTX 'bits' is a multiple of the number of frames per TTI.

TrCH multiplexing and 2nd DTX insertion – TrCH multiplexing simply involves concatenating the data from each TrCH together. The TrCHs are concatenated in order, i.e. TrCH1 then TrCH2, etc.
When using flexible TrCH positions, additional DTX may also be inserted at this stage if the TFC is not using the maximum capacity of the PhCH. If this is the case, the second DTX will be added.

2nd Interleaving – This is the second and last Interleaving. It operates on the multiplexed bits from all TrCHs, one frame at a time, and is described in this section. If more than one PhCH is used then the bits to be transmitted this frame are split equally between the PhCHs. If the total number of bits is X, and the number of PhCHs to be used is P, then bits are split into P segments of length U each, where:

$$U = \frac{X}{P}$$

The rate matching and DTX insertion will ensure that the number of bits matches the PhCH capacity exactly, and so U will be a whole number and will match the number of data bits available on the slot format to be used. The first

U bits are transmitted on the first PhCH, the next *U* bits on the next PhCH, etc. Each set of *U* bits is interleaved separately with a PhCH interleaver.

Spreading & Scrambling – See section 3.2.

DPCCH multiplexing – The different DPCH are multiplexed.

Shaping filter – The main function of this filter block is the radio resource controller filtering of the received signal to achieve an aggregate raised cosine response when paired with the transmit filter in the Node B**.**

Sampling rate – Adjusts the sampling rate to 15 Mhz.

Digital Upconverter (DUC) – The signal is upconverted digitally to the appropriate band prior to digital-to-analog conversion [3]***.***

CHAPTER 3. WCDMA CAPACITY

3.1 Introduction

3G cellular systems are identified as International Mobile Telecommunications-2000 under International Telecommunication Union and as Universal Mobile Telecommunications Systems (UMTS) by European Telecommunications Standards Institute. Besides voice capability in 2G, the 3G systems are required to have additional support on a variety of data-rate services using multiple access techniques. CDMA is the fastest-growing digital wireless technology since its first commercialization in 1994. The major markets for CDMA are North America, Latin America, and Asia (particularly Japan and Korea). In total, CDMA has been adopted by more than 100 operators across 76 countries around the globe. CDMA technology can offer about 7 to 10 times the capacity of analog technologies and up to 6 times the capacity of digital technologies such as TDMA. With its tremendous advantages such as voice quality, system reliability, and handset battery life compared to TDMA and FDMA technologies, WCDMA, the next generation of CDMA, is the best candidate for 3G cellular systems.

3.2 Spreading and Scrambling

Communication from a single source is separated by channelization codes, i.e., the dedicated physical channel in the uplink and the downlink connections within one sector from one MS. The Orthogonal Variable Spreading Factor (OVSF) codes were used to be channelization codes for UMTS.

The use of OVSF codes allows the orthogonality and spreading factor (SF) to be changed between different spreading codes of different lengths. Figure 3.1 depicts the generation of different OVSF codes for different SF values:

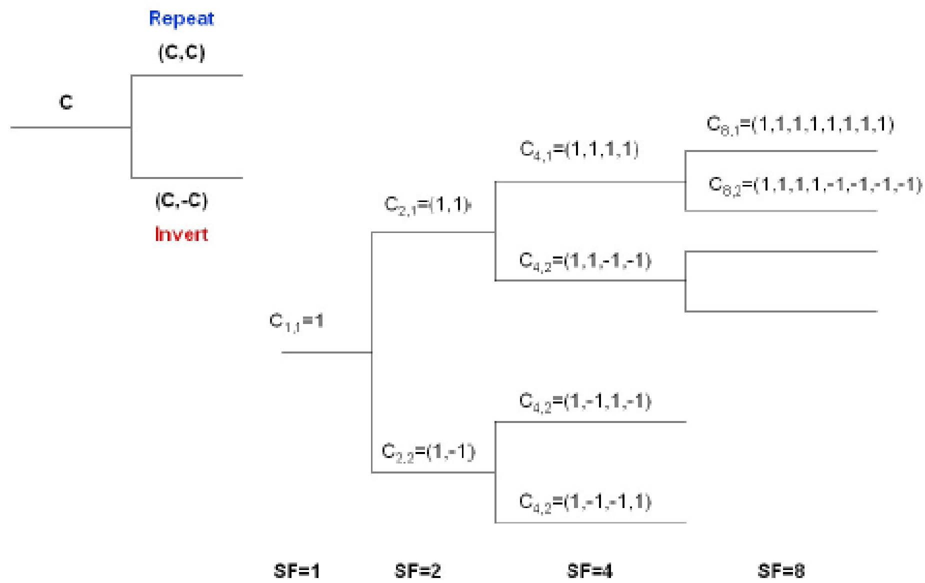

Figure 3.1: Generation of OVSF codes for different Spreading Factors [2]

The data signal after spreading is then scrambled with a scrambling codes to separate MSs and BSs from each other. Scrambling is used on top of spreading, thus it only makes the signals from different sources distinguishable from each other. Figure 3.2 depicts the relationship between the spreading and scrambling process. Table 3.1 describes the different functionality of the channelization and the scrambling codes:

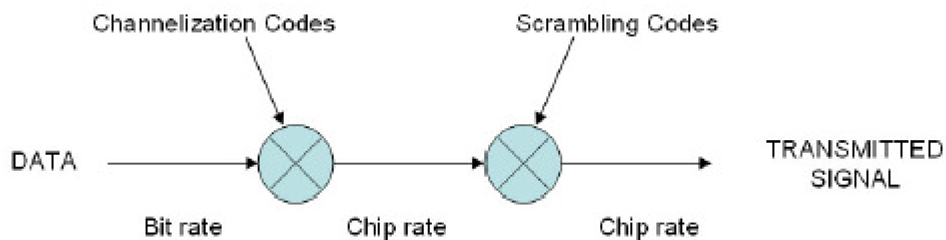

Figure 3.2: Relationship between spreading and scrambling [2]

	Channelization Code	Scrambling Code
Usage	Uplink: Separation of physical data (DPDCH) and control channels (DPCCH) from same MS Downlink: Separation of downlink connections to different MSs within one cell.	Uplink: Separation of MSs Downlink: Separation of sectors (cells)
Lenght	Uplink: 4-256 chips same as SF	Uplink: 10 ms = 38400 chips
Number of codes	Number of codes under one scrambling code = spreading factor	Uplink: Several millions Downlink: 512
Code family	Orthogonal Variable Spreading Factor	Long 10 ms code: Gold Code Short code: Extended S(2) code family
Spreading	Increases transmission bandwidth	Does not affect transmission bandwidth

Table 3.1: Functionality of the channelization and scrambling codes [2]

The typical required data rate or Dedicated Traffic Channel (DTCH) for a voice user is 12.2 Kbps. However, the Dedicated Physical Data Channel (DPDCH), which is the actual transmitted data rate, is dramatically increased due to the incorporated Dedicated Control Channel (DCCH) information, and the processes of Channel Coding, Rate Matching, and Radio Frame Alignment. Figure 3.3 depicts the process of creating the actual transmitted signal for a voice user:

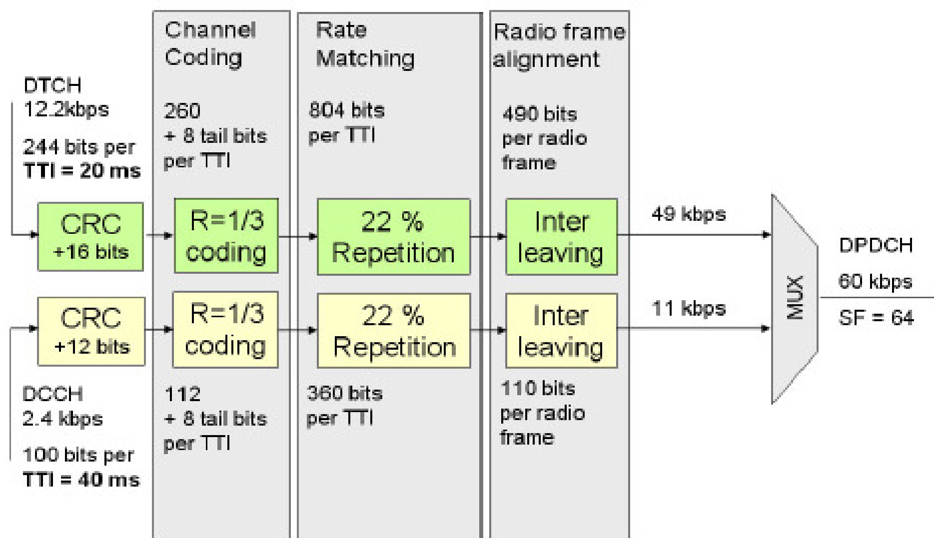

Figure 3.3: 12.2 Kbps Uplink Reference channel [2]

Figure 3.4 shows the DPDCH data rate requirement for 64 Kbps data user:

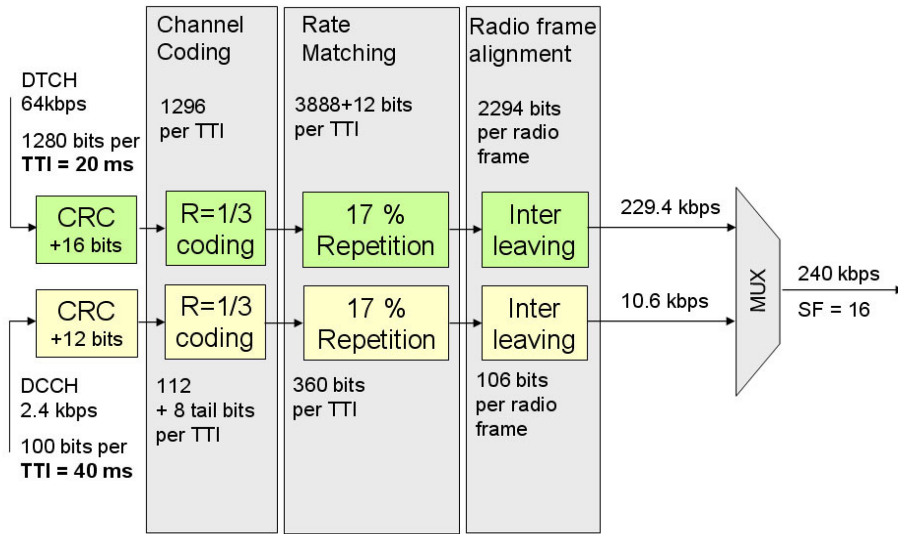

Figure 3.4: 64 Kbps Uplink Reference channel[2]

Table 3.2 shows the approximation of the maximum user data rate for different values of DPDCH:

DPDCH Spreading Factor	DPDCH channel bit rate (Kbps)
256	15
128	30
64	60
32	120
16	240
8	480
4	960

Table 3.2: Theoretical Uplink DPDCH data rates [2]

3.3 Network capacity measurements

In an experiment conducted in [1], a number of tests were performed to determine the capacity of a commercial Node B in a loaded network.

Using an experimental network, a test cell was chosen and all cells surrounding the test cell were artificially loaded in the downlink to 60% downlink load. The test cell was then loaded with voice and data calls.

Downlink capacity tests were performed for PS (packet switching) data (64k, 128k and 384k with 100% downlink activity) and a summary of these results is given in table 3.3.

Service (Kbps)	Maximum user capacity
Voice	50
PS 64	14
PS 128	8
PS 384	3

Table 3.3: Downlink Capacity [1]

The results obtained are in line with theoretical calculations.

CHAPTER 4. COMPUTING RESOURCE MANAGEMENT FRAMEWORK

4.1 Computational load distribution

SDR presents a hard real-time computing challenge. The computing system constraints become more severe with the evolution of wireless systems. Therefore, the flexibility of SDR terminals and network elements is a function of the computing resource managers, which need to continuously track the states of the computing resources and assign them properly.

An SDR processing chain, SDR application or waveform, is the part of an SDR transceiver that is implemented in software. It can be understood as a set of concurrent processes that continuously process and propagate real-time data. Such a processing chain is not specifically tailored but rather executable on any general-purpose platform with sufficient computing capacity. Therefore, an automatic mapping process needs to dynamically assign software modules to hardware resources, while meeting all computing system constraints. The computing system constrains are a function of the radio access technology and the radio environment and need to be accordingly managed. Aloe, therefore, features a computing resource management (CRM) framework that permits a flexible management of different types of computing system constraints and objectives.

The main constraint we are facing is the real-time needs of the users. That is why it is so important to distribute the computational load across the different processors, so the bits don't have to enter queues that would otherwise produce delays in the communications, but also to attain maximum efficiency.

4.2 Mapping process: t_w-mapping and the cost function

4.2.1 The t_w-mapping

The t_w-mapping is a general-purpose mapping algorithm. It is a windowed dynamic programming algorithm, where w indicates the window size. Without loss of generality, we are going to consider the basic implementation, assuming w = 1 in this project.

The mapping process is organized by the t_w-mapping diagram, which contains a trellis of N times M (row times column) t-nodes. A t-node is identified as $\{P_j, f_i\}$ and absorbs the mapping of SDR function f_i to processor P_j. Any t-node at step i (column i in the t_w-mapping diagram) connects to all t-nodes at step i+1. The

sequence of processors [$P_{k(0)}$, $P_{k(1)}$] identifies the w-path, a path of length 1 in our case, since w = 1, that is associated with t-node {$P_{k(1)}$, f_i }, where $P_{k(0)}$ is the w-path's origin processor at step i–1 and $P_{k(1)}$ the destination processor at step i.

The main feature of the t_w-mapping is that it is cost function independent. That is, any cost function can, in principle, be applied. The cost function guides the mapping process. It is responsible for managing a platform's available computing resources and an application's real-time processing requirements.

The algorithm sequentially pre-assigns, or pre-maps, processes to processors, starting with process or SDR function f_1 (first part of the t_w-mapping process). Processes f_2 to f_M are sequentially premapped following a common scheme. The third part consists of post-processing the premapping decisions for determining the final mapping solution. We will explain the three parts in continuation.

The first part consists of pre-mapping SDR function f_1 to all N processors and storing the pre-mapping costs at t-nodes {P_1, f_1} through {P_N, f_1}. Costs are computed using some cost function, which is externally defined.

At step i of the second part ($2 \leq i \leq M$), the t_1-mapping examines all N edges that are associated with {$P_{k(1)}$, f_i}. These edges originate at a t-node at step i–1, and end at a t-node at step {$P_{k(1)}$, f_i} (because w = 1, if w>1, the path would reach beyond {$P_{k(1)}$, f_i}. Figure 4.1 illustrates this for $P_{k(w=1)} = P_1$.

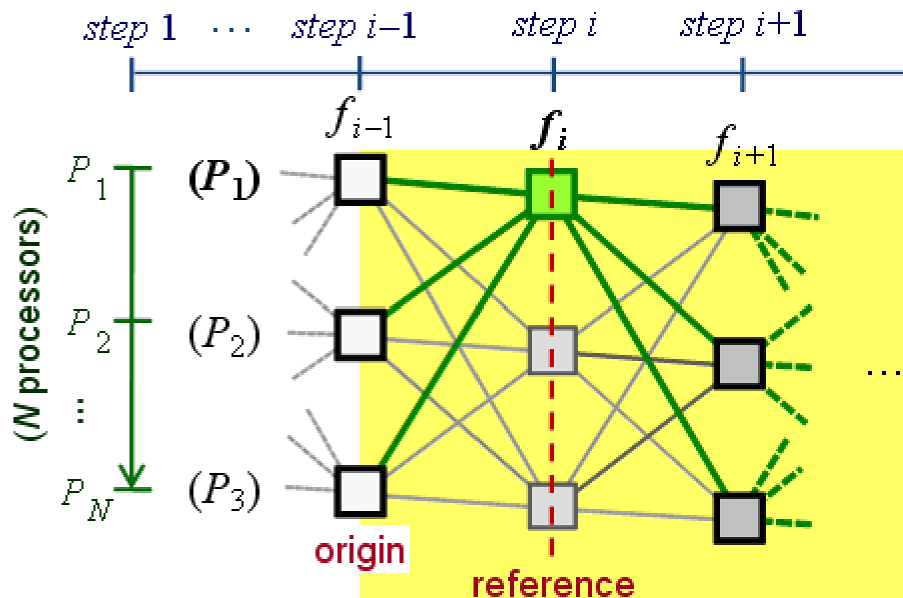

Figure 4.1: t_1-mapping

While i < M, the algorithm highlights the edge between a t-node at step i–1 and t-node $\{P_{k(1)}, f_i\}$ that corresponds to the minimum-cost w-path. The minimum-cost w-path is the path that is associated with the minimum accumulated cost as a function of the corresponding pre-mappings of f_1, f_2, ..., and f_i, where the w-path's origin t-node provides the pre-mapping information of f_1 to f_{i-1}. The algorithm then stores the cost and the remaining resources up to t-node $\{P_{k(1)}, f_i\}$ at $\{P_{k(1)}, f_i\}$. It processes all t-nodes at step i before considering those at step i+1.

If i = M, however, the complete w-path of minimum cost is highlighted. The total cost and the remaining resources are then stored at $\{P_{k(1)}, f_M\}$. After having processed all N t-nodes at step M, the third part of the algorithm starts.

The third and final part tracks the tw-mapping diagram backwards along the highlighted edges, starting at the minimum cost t-node at step M. That is, first the minimum cost t-node at step M is identified and then the tw-mapping backtracked along the highlighted edges until reaching a t-node at steo 1. This process finds the complete mapping solution for the given problem and cost function.

The figure 4.2 below shows an example of the postprocessing process. The example assumes 2 processors and 4 processes. The t-node $\{P_1, f_4\}$ is highlighted (a) and the trellis backtracked (b). The final mapping is determined by the traveled nodes of the backtracking process. Processes f_1, f_2, and f_4 are mapped to processor P_1, whereas f_3 is assigned to P_2.

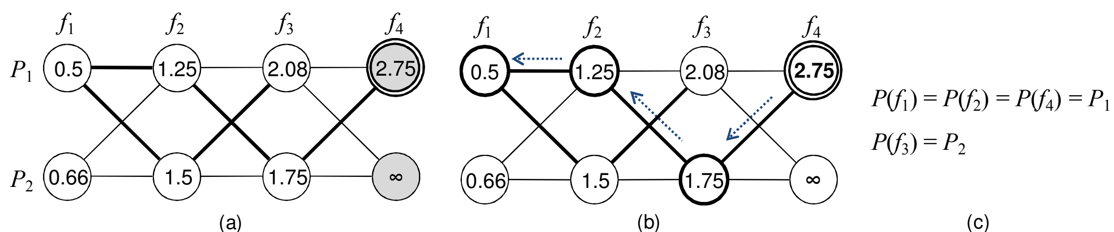

Figure 4.2: postprocessing process

4.2.2 The Cost Function

The cost function is responsible for managing the available computing resources of SDR platforms, trying to allocate the required resources to SDR applications. We generally define it as the sum of weighted cost terms, where each cost term captures the relation between the required and available resource of a specific type. Each of these terms must be less than or equal to 1. Otherwise, we would reserve more resources than available. Hence, the cost function computes the cost of a pre-mapping as a function of the available and the required resources. This implies dynamic resource updates.

For managing the processing and interprocessor bandwidth resources we define the cost function as

$$cost = q \cdot cost_comp + (1-q) \cdot cost_comm$$

This two-term cost function manages the available processing and bandwidth resources, while trying to meet the corresponding computing resource requirements. Weight q is defined in interval [0, 1]. It defines the relative importance of the *computation cost* with to the *communication cost*. In this work we consider q = 0.5 and so equally weight the two cost terms.

More information on the cmputing resource management framework can be found at http://flexnets.upc.edu/trac/ or [14].

CHAPTER 5: RESOURCE MANAGEMENT WITH DATA DEPENDENCY CONSTRAINTS

5.1 Data dependency in the Base Station

As is already explained in Chapter 2, our BTS model is divided in two sub-chains, the chip level and the bit level processing parts. While the users are processed one by one at the bit level, they are all processed at once at the chip level, which processes the common data stream in the WCDMA frequency band or channel. This means that the chip level part only needs to be executed once for all user transmitting in the same channel, reducing the necessary hardware and power consumption.

We will work on an already written code in *C* language that doesn't differenciate between these two levels (it did not need to since it was assuming the UE). Our objective is modifying this code for the multiuser base station case. In 4.4 we will discuss the solutions.

Now we will introduce the most important files of the code:

api_test.c **–** The main function, reads and prints and calls the appropriate functions depending on the scenario the user wants to run.

api_test.h **–** Defines the different scenarios. Here we are introducing the *sdrcloud* scenario.

tw_mapping.c **–** Implements the tw-mapping algorithm. The main modifications have to be made here since it is where the mapping of both the bit and the chip levels is done and the data dependency between the two sub-chains needs to be considered.

umts_chip.c **–** This file will be created, defining the computation and communication costs of the chip level part.

umts_bit.c **–** This file will be created, defining the computation and communication costs of the bit level part.

5.2 Solutions

We have analyzed different solutions for solving the problem of separating the chip and bit levels and mapping them independently while taking into account the data dependency between them. We will discuss two solutions here, which are general and useful for managing other wireless communications transceivers. Without loss of generality, we implemented our solutions using the UE receiver model. Although the BTS recevier model for processing a single is not the same as the UE model, this difference is not relevant for what we pretend to demonstrate here.

Figure 5.1 shows our starting point. All the functions in the code are connected and we want to separate the chip level functions from the bit level ones. Also, we will need to modify the code to run the chip level part only once for all the users while the bit level one has to be run as many times as the total number of users.

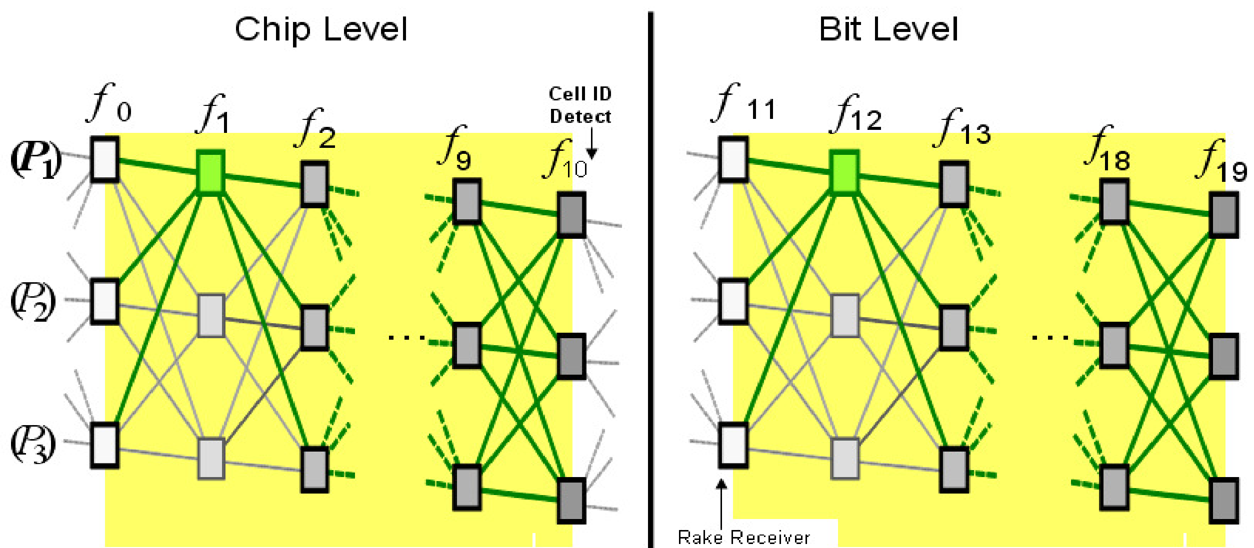

Figure 5.1: Mapping process, split into two subprocesses

5.2.1 The Constraint Vector Solution

The initial idea to solve this problem was to create a constraints vector. The first user would be mapped as normal. Beginning with the second, the rest of the users would have a constraint vector attached to them. This constraint vector defines the data dependency between the processes in the chip level part and those in the bit level part. In particular, the data flow requirements are associated to the processors where the chip level processes are mapped.

For example, if processes f_5, f_6 and f_7 send data at a rate of b', b'' and b''' respectively to some processors of the bit level processing chain, and f_5, f_6 are both mapped to P_1, and f_7 to P_3, then the constraint vector for a three processor case would be:

constraint_vector = (b'+ b'', 0, b''').

The reason we have not implemented this solution was that even thought it was possible to implement, it would have required many changes in the code. The code, as it was in the first place, mapped the waveform once and ended. The constraint vector could not be applied to the first function of each new user so easily because the code did not contemplate any communication costs associated with the first processes to be mapped.

The aim of this part of the TFC was to make it work but also to change as little as possible the code files. We therefore propose another soultion below.

5.2.2 The Ghost Function Solution

A solution was needed that would modify the code as little as possible. The two parts of the code had to be connected perfectly in terms of bandwidth demand between the functions of the chip level and those of the bit level, as indicated by the task graph of Figure 2.1.

Our first idea was to send a parameter to the computing resource manager (tw-mapping algorithm) in charge of starting the bit level process that indicated the processor allocated to the last function of the chip level. But this solution had two problems:

1. It assumes that the data flow dependency is between the last function of the chip-level and the first of the bit-level processing part.
2. It does not take into account the bandwidth between the two functions.

To solve this problem, the ghost function was created. This function is added at both sides, after the last function of the chip-level and before the first one of the bit-level processing chain. It would serve as a connecting point between the two chains. This way we could assume all data flow dependencies and bandwidth demands between the two parts of the receiver. The ghost function has zero processing cost, which is where its name comes from.

The ghost function at the end of the chip-level processing part receives all data that needs to be transmitted to the bit-level processing part. Its output data flow is the sum of the incoming flows.

The final chains would look like this in figure 5.2. It shows the two instances of the ghost function and the data that is routed through it

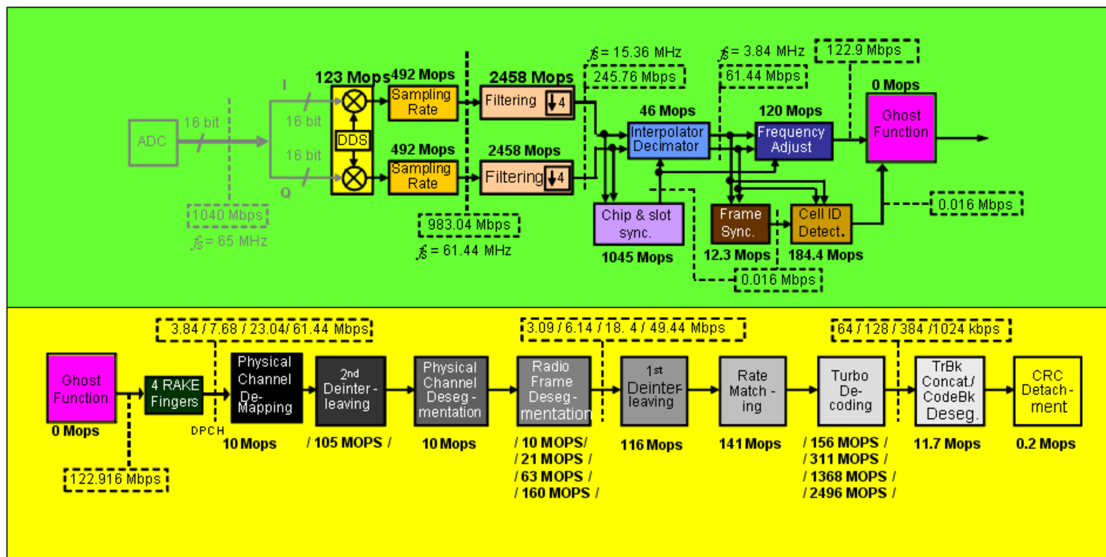

Figure 5.2: processing chain with the ghost functions

5.2.3 Implementation

The two parts of the processing chain are defined in umts_chip.c and umts_bit.c, which are given in the Appendix. There are 11 functions in both parts as summarized below in tables 5.1 and 5.2.

Function Number	MOPS	Description
f1	123	DDC
f2	492	Sampling rate
f3	492	Sampling rate
f4	2458	Filtering
f5	2458	Filtering
f6	1045	Chip&Slot sync.
f7	46	Interpolator/Decimator
f8	120	Frequency Adjust
f9	12,3	Frame Sync.
f10	184,4	Cell ID Detect.
f11	0	Ghost function

Table 5.1: Chip level functions

Function	MOPS	Description
f1	0	Ghost function
f2	139	RAKE Receiver
f3	10	PHY channel demapping
f4	105	2nd Interleaving
f5	10	Phy channel deseg.
f6	10	Radio frame deseg.
f7	116	1st Deinterleaving
f8	141	Rate Matching
f9	156	Turbo decoding
f10	11,7	TrBk Concat/CBk Deseg
f11	0,2	CRC Detatchment

Table 5.2: Bit level functions

CHAPTER 6: TESTING AND SIMULATIONS

6.1 Testing the program

Now that we know the expected network capacity (chapter 3) and we have the simulation framework and resource manager modified due to section 5.2.2, we are able to test our contribution.

We will test our program using the 64 kbps processing chain model, assuming the voice service. The experiment in Section 3.3 tells us the expected maximum capacity for the voice service is around 50, so we will set the number of users accordingly. Note that the 32 kbps processing chain model would only be negligibly different form the 64 kbps model.

6.2 Output analysis

First of all, we are going to discuss the output of the program for any given number of users or processors. In this example we are going to use 20 users and 10 processors, but those are not important now, they are two numbers chosen randomly for testing purposes.

The data center model consists of Nx processors, in general, which are fully connected. That is, between every pair of processors there is a dedicated full-duplex link of 100 Mbps bandwidth. This model is valid for clusters rather than for data centers. A cluster is a small number of tightly-coupled processing cores, whereas the data center usually consists of several clusters. The processing capacity of each processor is 5000 MOPS.

The first part of the output is shown in figure 6.1

```
This test suite applies the tw-, gw-mapping or optimal mapping algorithm on diff
erent mapping scenarios.
Scenario and mapping options can be modified in "api_text.h".
Execution parameters are the algorithm window size 'w' and the cost function par
ameter 'q'.
w = 1, 2, 3, ..., min(Wmax, M-1) - default: w = 1. Wmax = 5 (mapper.h) and the n
umber of application modules M = 11 (api_test.h).
q = 0, ...,1 - default: q = 0.5.
weighting = 0 (no additional cost function weighting), 1 (static weights), 2 (dy
namic weights).
Additional parameters are the processing and bandwidth loads (c_load, b_load), w
hich are used for scaling the computing resources (0: no scaling).

****************************
*   t1-mapping  q = 0.50   *
****************************

 * c-load = 0.00
 * b-load = 0.00
 * Mapping in original (logical) order of processing blocks (no_ord)

waveform_totals: 7430.699707 MOPS       5520.863281 Mbps
Mapping cost = 1.782490
Mapping:
0       0       0       0       2       1       0       0       0       0
0
-------------------------------------------------------------
```

Figure 6.1: Output (part 1)

The program starts giving us information about its purpose and its parameters. The window is w=1 and the cost function parameter q=0.5.

Then we have our first mapping. This first mapping is the most important of all because it represents the mapping of the chip level. Those 11 numbers separated by spaces represent the 11 functions our BTS model has at the chip level, and their numbers represent the processors that are allocated to each function. For example, f1 (digital downconverter), is mapped to processor of index 0 (first processor).

All the following code represents the mapping of the bit level processing chain of each user. In figure 6.2 we see the chip level mapping, followed by two bit level mappings, corresponding to the first and second users. Note also that the bit level mapping has 11 numbers because it also has 11 functions, the same as the chip level, but this is just a coincidence.

```
 * c-load = 0.00
 * b-load = 0.00
 * Mapping in original (logical) order of processing blocks (no_ord)

waveform_totals: 7430.699707 MOPS       5520.863281 Mbps
Mapping cost = 1.782490
Mapping:
0       0       0       0       2       1       0       0       0       0
0
-------------------------------------------------------------
waveform_totals: 698.900024 MOPS        144.775970 Mbps
Mapping cost = 0.123989
Mapping:
0       0       3       3       3       3       3       3       5       4
4
-------------------------------------------------------------
waveform_totals: 698.900024 MOPS        144.775970 Mbps
Mapping cost = 0.133802
Mapping:
0       6       6       6       6       6       6       6       8       7
7
-------------------------------------------------------------
```

Picture 6.2: Output (part 2)

And this goes on until the last user (the 50th) is mapped, then the program tells us if there were any infeasible mappings, the total processing and bandwidth demands and their mean, as shown in figure 6.3.

```
-------------------------------------------------------------
waveform_totals: 698.900024 MOPS        144.775970 Mbps
Mapping cost = 0.198354
Mapping:
0       5       5       5       5       5       6       6       3       1
1
-------------------------------------------------------------
Platform #1:    0.0000 % infeasible mappings

Total processing demand:        21408.71 MOPS
Total bandwidth demand: 8416.38 Mbps

Total processing demand in the mean:    1019.46 MOPS
Total bandwidth demand in the mean:     400.78 Mbps
No. of skipped graphs:                  0

======================================
Press any key to continue
```

Figure 6.3: Output (part 3)

Another important note is on the ghost functions. They are represented by the last number of the chip level mapping (first mapping) and the first number of each bit level mapping (all the other mappings). We can see the first function of the mapping of the bit level is always mapped in the same processor as the last function of the mapping of the chip level, in this case, processor number 0. This is important because the two instances of the ghost function represent the union between the chip- and bit-level processing parts.

Also note that the ghost function is mapped to the same processor as f8 (Frequency Adjust). This is so because the highest bandwidth demand between the chip- and bit-level processing chains originates at f8. Since the ghost function has zero processing requirements, the computing cost is zero (section 4.2.2). The communication cost is proportional to the interprocessor data flows. Processor internal data flows have zero cost. Therefore, the cost is minimized by mapping the ghost function to the same processor as f8.

6.3 Minimum number of processors

As another test we now determine the minimum number of processors needed to map 50 users. To find this, we will change Nx (total number of processors) until we arrive to a 0% infeasible mappings. An infeasible mapping means that the real-time processing and dataflow demands cannot be met with the available resources.

For Nx = 8 (figure 6.4), for example we have 9.8 % infeasible mappings, corresponding to 5 users:

```
-----------------------------------------------------------------
Platform #1:    9.8039 % infeasible mappings

Total processing demand:        42375.69 MOPS
Total bandwidth demand: 12759.67 Mbps

Total processing demand in the mean:    830.90 MOPS
Total bandwidth demand in the mean:     250.19 Mbps
No. of skipped graphs:                  0

=======================================
```

Figure 6.4: Output for Nx=8

For Nx = 9 (figure 6.5), we obtain the following result:

```
-----------------------------------------------------------------
Platform #1:    0.0000 % infeasible mappings

Total processing demand:        42375.69 MOPS
Total bandwidth demand: 12759.67 Mbps

Total processing demand in the mean:    830.90 MOPS
Total bandwidth demand in the mean:     250.19 Mbps
No. of skipped graphs:                  0

=======================================
```

Figure 6.5: Output for Nx=9

With Nx=9, there are no infeasible mappings. The minimum number of processors is then 9. If the total number of users becomes higher, the manager will not be able to map all of them.

Figure 6.6 shows of the successfully mapped users as a function of Nx.

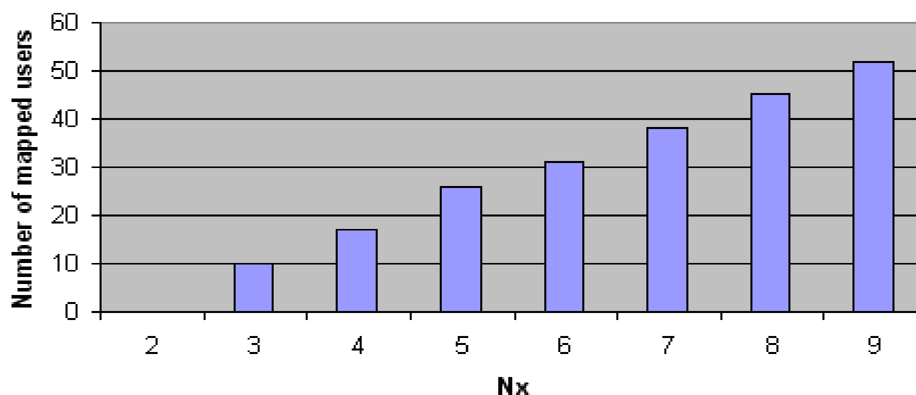

Figure 6.6: Number of successfully mapped users as function of the number of processors Nx

We can observe that the number of mapped users does not increase linearly. For example:
Nx = 4, 17 mapped users.
Nx = 5, 26 mapped users (9 additional users).
Nx = 6, 31 mapped users (5 additional users).

With each additional processor there is, theoretically, room for:

$$5000 \text{ MOPS} / 699 \text{ MOPS} = 7 \text{ new users}$$

But the t_w-mapping distributes the load instead of filling processors one by one. If we assume the first two processors are necessary to map the chip rate part, on average we obtain that 52 / (9–2) = 7.4 users are mapped with each additional processor.

6.4 Processing resource saving

First, we want to put in numbers how much the processing resources saving increased by separating the chip and bit levels. To do this, we will take samples of the processing and bandwidth demands for different numbers of users using both the old program and our new version with the separation by levels.

Number of users	Old code proc. demands	Old code bw demands	New code proc. demands	New code bw demands
T=1	8129,6	5665,64	8129,6	5665,64
T=10	81296	56656,36	14419,7	6968,62
T=20	162592	113312,72	21408,71	8416,38
T=30	243888	169969,1	28397,71	9864,15
T=40	325184	226625,45	35386,71	11311,91
T=50	406480	283281,8	42375,7	12759,67

Table 6.1: processing and bandwidth demands

As we can see in table 6.1, both the processing and bandwidth demands decreased dramatically from the old code to the new code. It is as expected, since those two demands are higher on the chip level than in the bit level.

We can also analyze the number of users versus processing and bandwidth demands. In table 6.1 we can see, for example, that those demands do not double each time we double the number of users. Instead, the demands increase linearly for each new user since we are only mapping the bit level part of that user instead of the whole waveform.

CONCLUSIONS

In this work we have analyzed computing resource management techniques for SDR Clouds. We have provided some theoretical information on the computing resource requirements and data flow dependencies of WCDMA used for the 3G/UMTS standard.

On the basis of the available information for UE, we have developed the BTS model, differentiating between the symbol and bit levels. This permitted assigning computing resources so that user signals can be correctly processed, while reserving just the necessary computing resources.
Others standards, most importantly LTE, also have a common processing part for all the users and another for each individual user signal, making this work useful for emerging standards. Our ghost function contribution permits managing such date flow dependencies automatically. It is just necessary to split the processing chain model and indicate the data flow dependencies by introducing a ghost function at the end and the beginning of the split SDR application model, respectively.

Bibliography

[1] T. Griparis, T M Lee, ”The Capacity of a WCDMA Network: a case study”.

[2] R. Akl, S. Nguyen, “Capacity and Throughput Optimization in Multi-cell 3G WCDMA Networks”, August 2005.

[3] H. Holma and A. Toskala, “WCDMA for UMTS”, 3rd ed. Wiley interscience, Oct. 2004.

[4] R. Tanner, J. Woodard, “WCDMA – Requirements and Practical Design”, John Wiley & Sons, ltd, England, 2004.

[5] V. Marojevic, X. Revés, A. Gelonch, “A computing resource management framework for software-defined radios,” *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1399-1412, Oct. 2008.

[6] Technical Specification Group Radio Access Network (3GPP), “TS 25.213 V5.5.0 – spreading and modulation (FDD),” Dec. 2003, www.3gpp.org.

[7] Technical Specification Group Radio Access Network (3GPP), “TS 25.212 V6.4.0 – multiplexing and channel coding (FDD),” March 2005, www.3gpp.org.

[8] T. Faber, M. Schönle, “DSP-platform target report,” SLATS Consortium, Project no. 27016, Deliverable D23, Dec. 1999.

[9] H. Van Peteghem, L. Schumacher, “Emulation of a Downlink Spreading Factor Allocation Strategy for Rel’99 UMTS”, Belgium.

[10] R. Menolascino, M. Pizarroso, C. Lepschy, and B. Salas, “Third generation mobile systems planning issues,” Proc. IEEE VTC’98, pp. 830–834, May 1998.

[11] E. Berruto, M. Gudmundson, R. Menolascino, W. Mohr, and M. Pizarroso, “Research activities on UMTS radio interface, network architectures, and planning,” IEEE Commun. Mag., vol. 36, pp. 82–95, Feb. 1998.

[12] E. Amaldi, A. Capone, “Planning UMTS Base Station Location: Optimization Models With Power Control and Algorithms”, VOL. 2, NO. 5, SEPTEMBER 2003.

[13] S. Breyer, B. Haberland, E. Suzu, “Arquitectura del nodo B UMTS en un entorno Multiestándar”, Revista de Telecomunicaciones de Alcatel - 1er Trimestre 2001.

[14] V. Marojevic, I. Gomez, A. Gelonch, “ALOE Session 7: Computing Resource Management”, 2010, http://flexnets.upc.edu/trac/

[15] I. Gomez, V. Marojevic, A. Gelonch, “Resource management for software-defined radio clouds,” *IEEE MICRO*, accepted for publication. IEEE early access. doi: 10.1109/MM.2011.81.].

[16] R. Buyya et al, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, *Elsevier J. Future Generation Computer Systems*, vol. 25, iss. 6, Pages 599-616, June 2009.

[17] I. Gomez, V. Marojevic, A. Gelonch, “ALOE: an open-source SDR execution environment with cognitive computing resource management capabilities”, *IEEE Commun. Mag.*, vol. 49, iss. 9, pp. 76-83, Sept. 2011.

[18] Middleware definition: http://es.wikipedia.org/wiki/Middleware

ANNEX

C files from the CRM framework – only the most important files are here, the others can be downloaded from http://flexnets.upc.edu/trac/wiki/ResourceManagement.

umts_bit.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "api_test.h"

extern float Px[Nx][R];                         // processing powers of N processors per
resource model
extern float Bx[Nx][Nx][R-RB];          // Bandwidth matrices
extern float B_bus;                                     // Bus bandwidth

extern int arch_type[R];                // architecture

extern float b_unsort[Mx][Mx];
extern float m_unsort[Mx];                      // unsorted (original) c-vector

extern float Ctotal[R];
extern float Btotal[R];


// Chip- and bit-rate digital signal processing chain of UMTS receiver
app_totals umts_bit(float c_load, float b_load)
{
        /*int i, j;                             // loop indices
        float m[M];                                     // initial processing demand vector
(unsorted)
        float m_total = 0;                      // total processing demand
        float b_total = 0;                      // total bandwidth demand
        float f = 1;                            // scaling factor*/

        int i, j, r;                    // loop indices
        int B_hom = 1000;                       // unidirectional bandwidth between any two
proc's
        int P_hom = 10000;                      // unidirectional bandwidth between any two
proc's
        float sf_B = 1;                         // scaling factor (Bandwidths)
        float c_total = 0;                      // total processing demand
        float b_total = 0;                      // total processing demand
        float f = (float)1;/*(float)0.01/17;*/          // scaling factor due to time-slot duration and
Emax
        float sf = 1;                           // scaling factor due to load

        app_totals waveform_totals;     // structure containing two parameters: c_total and
b_total
```

```
	m_unsort[0] = 0;			//fM	Ghost function
	m_unsort[1] = 139;			//f11	(RAKE Receiver)
	m_unsort[2] = 10;				//f12	(PHY Ch demapping)
	m_unsort[3] = 105;			//f13	(2nd Interleaving)
	m_unsort[4] = 10;				//f14	(PHY Ch Desegmentation)
	m_unsort[5] = 10;				//f15	(Radio Frame Desegm.)
	m_unsort[6] = 116;			//f16	(1st Deinterleaving)
	m_unsort[7] = 141;			//f17	(Rate Matching)
	m_unsort[8] = 156;			//f18	(Turbo Decoding)
	m_unsort[9] = (float)11.7;//f19	(TrBk Concat. / CodeBk Deseg.)
	m_unsort[10] = (float)0.2;		//f20	(CRC Detachment)


	for (i=0; i<Mx2; i++)
		c_total += m_unsort[i];

	// DATAFLOW MODEL
	for(i=0;i<Mx;i++)
		for(j=0;j<Mx;j++)
			b_unsort[i][j]=0;

	b_unsort[0][1] = (float)122.9*f + (float)0.016*f;	// fM  -> f11	sum of data flows
between chip-rate and bit-rate processing part.
	b_unsort[1][2] = (float)3.84*f;	// f11 -> f12
	b_unsort[2][3] = (float)3.84*f;	// f12 -> f13
	b_unsort[3][4] = (float)3.84*f;	// f13 -> f14
	b_unsort[4][5] = (float)3.84*f;	// f14 -> f15

	b_unsort[5][6] = (float)3.09*f;	// f15 -> f16
	b_unsort[6][7] = (float)3.09*f;	// f16 -> f17

	b_unsort[7][8] = (float)3*0.064*f;		// f17 -> f18

	b_unsort[8][9] = (float)0.064*f;	// f18 -> f19
	b_unsort[9][10] = (float)0.064*f; // f19 -> f20

	for (i=0;i<(Mx2-1);i++)
		for(j=(i+1);j<Mx2;j++)
			b_total += b_unsort[i][j];

	waveform_totals.c = c_total;
	waveform_totals.b = b_total;

	return waveform_totals;

}
```

umts_chip.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "api_test.h"
```

```
extern float Px[Nx][R];                         // processing powers of N processors per
resource model
extern float Bx[Nx][Nx][R-RB];          // Bandwidth matrices
extern float B_bus;                                     // Bus bandwidth

extern int arch_type[R];                // architecture

extern float b_unsort[Mx][Mx];
extern float m_unsort[Mx];                      // unsorted (original) c-vector

extern float Ctotal[R];
extern float Btotal[R];


extern int constraint_p;        // proceso que envia datos hacia la parte bit-rate



// Chip- and bit-rate digital signal processing chain of UMTS receiver
app_totals umts_chip(float c_load, float b_load)
{
        /*int i, j;                             // loop indices
        float m[M];                                     // initial processing demand vector
(unsorted)
        float m_total = 0;                      // total processing demand
        float b_total = 0;                      // total bandwidth demand
        float f = 1;                            // scaling factor*/

        int i, j, r;                    // loop indices
        int B_hom = 1000;                       // unidirectional bandwidth between any two
proc's
        int P_hom = 10000;                      // unidirectional bandwidth between any two
proc's
        float sf_B = 1;                         // scaling factor (Bandwidths)
        float c_total = 0;                      // total processing demand
        float b_total = 0;                      // total processing demand
        float f = (float)1;/*(float)0.01/17;*/  // scaling factor due to time-slot duration and
Emax
        float sf = 1;                           // scaling factor due to load

        app_totals waveform_totals;     // structure containing two parameters: c_total and
b_total


        m_unsort[0] = 123;                      //f1    (DDC)
        m_unsort[1] = 492;                      //f2    (Sampling rate)
        m_unsort[2] = 492;                      //f3    (Sampling rate)
        m_unsort[3] = 2458;                     //f4    (Filtering)
        m_unsort[4] = 2458;                     //f5    (Filtering)
        m_unsort[5] = 1045;                     //f6    (Chip & Slot sync.)
        m_unsort[6] = 46;                       //f7    (Interpolator/ Decimator)
        m_unsort[7] = 120;                      //f8    (Frequency Adjust)
        m_unsort[8] = (float)12.3;      //f9    (Frame sync.)
        m_unsort[9] = (float)184.4;     //f10   (Cell ID Detect.)

        m_unsort[10] = 0;               //f11   pseudo-function, connecting point between chip
and bit-rate processing chain

        for (i=0; i<Mx1; i++)
                c_total += m_unsort[i];
```

```
	// DATAFLOW MODEL
	for(i=0;i<Mx;i++)
		for(j=0;j<Mx;j++)
			b_unsort[i][j]=0;

	b_unsort[0][1] = 1040*f;			// f1 -> f2
	b_unsort[0][2] = 1040*f;			// f1 -> f3

	b_unsort[1][3] = 983*f;				// f2 -> f4
	b_unsort[2][4] = 983*f;				// f3 -> f5

	b_unsort[3][5] = (float)245.8*f;	// f4 -> f6
	b_unsort[4][5] = (float)245.8*f;	// f5 -> f6

	b_unsort[3][6] = (float)245.8*f;	// f4 -> f7
	b_unsort[4][6] = (float)245.8*f;	// f5 -> f7

	b_unsort[5][6] = (float)0.016*f;	// f6 -> f7
	b_unsort[5][7] = (float)0.016*f;	// f6 -> f8

	b_unsort[6][7] = (float)122.9*f;	// f7 -> f8
	b_unsort[6][8] = (float)122.9*f;	// f7 -> f9
	b_unsort[6][9] = (float)122.9*f;	// f7 -> f10

	b_unsort[8][9] = (float)0.016*f;	// f9 -> f10

	//data flows toward bit-rate processing part
	b_unsort[7][10] = (float)122.9*f; // f8 -> f11	Frequency Adjust --> RAKE, which is
represented by a pseudo-function here
	b_unsort[9][10] = (float)0.016*f; // f10 -> f11	Cell ID Detect. --> RAKE, which is
represented by a pseudo-function here


	for (i=0;i<(Mx1-1);i++)
		for(j=(i+1);j<Mx1;j++)
			b_total += b_unsort[i][j];


	constraint_p = Mx1-1;		// pseudo-node, last module in the processing chain,
module Mx1-1

	waveform_totals.c = c_total;
	waveform_totals.b = b_total;

	return waveform_totals;

}
```

tw_mapping.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include "mapper.h"

extern int N;      // number of platform's processors
extern int M;      // number of application's tasks/SDR functions/processes

extern float P[Nmax];                                   // processing powers

extern int I[Nmax][Nmax];                               // interconnection topology matrix
extern float B[Nmax*Nmax];                              // bandwidth resources

extern float m_sort[Mmax];                              // application's processing requirements
sorted
extern float b[Mmax][Mmax];                             // bandwidth requirements

extern float a_pathsU[Nmax+1][Nmax*Mmax];   // path costs
extern int PP[Nmax][Mmax][Mmax];                        // N*M matrix of Trellis nodes, each
nodes contains the whole mapping up to this step

extern int weights;
extern float k1, k2[Nmax];

extern float c_total, b_total;   // total processing and bandwidth demands
extern float C_total, B_total;      // total processing and bandwidth capacities of the given SDR
platform

int mapping_vector[Mmax];                               // tw-mapping solution: module m[i] is
mapped to processor mapping_vector[i]

// function prototypes "cost function"; cost = computation_cost (cost_compU) +
communication_cost (cost_commU)
float cost_compU(float mx, float P_rem[], int idx);
float cost_commU(float bx1, float bx2, float B_rem[], int ix1, int ix2, int k0);

void generar_k1(float m_total_done);
void generar_k2(float (*B)[Nmax*Nmax], float b_total_rem);

/*        Locally used arrays defined globally due to limited program memory reasons */
        //float m[Mmax];                                         // processing requirements

        float P_tempGS[Nmax][Nmax];            // resting Processing capacities, temporary-
Global-Saved (step i-1)
        float P_tempG[Nmax][Nmax];             // resting Processing capacities, temporary-
Global (step i)

        float B_tempGS[Nmax][Nmax*Nmax];// resting Bandwidth capacities, temporary-Global-
Saved (step i-1)
        float B_tempG[Nmax][Nmax*Nmax];      // resting Bandwidths, temporary-Global (step i)

        float P_remL[Nmax][Nmax];
        float B_remL[Nmax][Nmax*Nmax];

        int indx[Nmax][Mmax];                  // points to the selected edge (decision) at a t-
node
        extern int edges[Nmax][Mmax-1];        // contains the pre-mapping decisions of the tw-
mapping forward processing part (part II)

        // w > 1
        int k[Wmax];
        float P_tempL1[Nmax][Nmax];            // resting processing capacities at node (j,i)
associanted with initial node (k[0],i-1)
```

```
	float P_tempLw[Wmax][Nmax];		// resting processing capacities at level
l=0,1,...,w-2 (k[l+1], P_tempLw[l+1][.])

	float B_tempL1[Nmax][Nmax*Nmax];// resting bandwidths at level 1 (k[0])
	float B_tempLw[Wmax][Nmax*Nmax];// resting bandwidths at levels 1 (k[0]), 2 (k[1]), 3
(k[2]), 4 (k[3]), and 5 (k[4])

	float costw[Wmax];				// cost at levels 2 (k[1]), 3 (k[2]), 4 (k[3]),
and 5 (k[4]) // costw[0] is skipped!!!
	float costw_auxx[Nmax];			// cost at levels 2, 3, 4 or 5 - global
decision

	// final mapping ...
	// ... indices of the forward-tracking (w > 1)
	int indxw[Nmax][Wmax-1];		// indxw[0][j] (w > 1), indxw[j][1] (w > 2),
indxw[j][2] (w > 3), indxw[j][3] (w > 4) point from t-node (j,M-w+1) to t-nodes at step (M-w+1) + 1,
(M-w+1) + 2, (M-w+1) + 3, and (M-w+1) + 4


mapping_result tw_mapping(int (*ptr_nodes)[Mmax], int w, int NP)	// trellis nodes and
window size w, NP: processor where the 1st needs to be mapped. If NP=-1, no constraints.
//mapping_result tw_mapping(int w)	// trellis nodes and window size w
{
	int i;						// step index
	int j;						// processor index
	int l;						// edge index
	int z,s,u,v, lx;			// loop indices

	int index = -1;				// index that points to the minimum-cost
t-node at step M (initialized with -1 as we suppose infeasibility)
	float cost_t = infinite;		// total cost of the tw-mapping solution

	float b_total_rem = b_total;	// initialize the total remaining bandwidth requirements
with the total initial bandwidth requirements
	float m_total_done = 0;		// the accumulated processing requirements of
already mapped processes, just before addressing m[i]

	mapping_result mresult;

	// initialization of final mapping representation
	for (i=0; i<M; i++)
		mapping_vector[i] = -1; // mapping_vector[i] = -1 indicates that m[i] is not
mapped

	for (j=0; j<N; j++)
		for (i=0; i<M; i++)
			indx[j][i] = -1; // indx[j][i] = -1 indicates that there is no feasible edge
reaching m[i]

	for (j=0; j<N; j++)
		for (i=0; i<(M-1); i++)
			edges[j][i] = -1; // edges[j][i] = -1 indicates that there is no feasible edge
associated with module m[i]

	for (j=0; j<N; j++)
		for (i=0; i<(Wmax-1); i++)
			indxw[j][i] = -1; // indxw[j][i] = -1 indicates that there is no feasible edge
associated with m[i]

	// initialization of processor assignments
```

```
        for (i=0; i<N; i++)
                for (j=0; j<M; j++)
                        for (z=0; z<M; z++)
                                PP[i][j][z] = 0;

        // initialization of path costs
        for (i=0; i<N+1; i++)
                for (j=0; j<N*M; j++)
                        a_pathsU[i][j] = 0;

        // set initial processing powers
        for (u=0; u<N; u++)
    for (v=0; v<N; v++)
                        P_tempG[u][v] = P[v];

        // set initial bandwidth capacities
        for (z=0; z<N; z++)
                for (u=0; u<(N*N); u++)
                        B_tempG[z][u] = B[u];

        /* for a static k1, k2 assignment*/
        if (weights == 1)
        {
                k1= c_total/C_total;
                k2[0]= (float)(N * b_total) / (M * B_total);

                for (i=1; i<N; i++)
                        k2[i]= k2[0];
        }

        // assignment of m[0] to the proc NP and update remaining processing powers

        if (NP == -1)
        {
                // assignment of m[0] to all N proc's and update remaining processing powers
                for (j=0; j<N; j++)
                {
                        PP[j][0][0] = j;
                        a_pathsU[N][ptr_nodes[j][0]] = cost_compU(m_sort[0], P_tempG[j], j);
        // pre-mapping cost for m[0]
                }
        }
        else
        {
                PP[NP][0][0] = NP; //asigno m[0] solamente al procesador de donde me vienen
los datos.
                //a_pathsU[N][ptr_nodes[NP][0]] = cost_compU(m_sort[0], P_tempG[NP], NP);
//pre-mapping cost for m[0]
                a_pathsU[N][ptr_nodes[NP][0]] = 0; //pre-mapping cost for m[0]
        }

        //***************************************************************************

        for (i=1; i<(M-w+1); i++) // STEP i = 2..M-w+1
        {
                // save... guarda anchos de banda y requerimientos utilizados i = abcisas j =
ordenadas
                for (u=0; u<N; u++)                  // ...for each t-node at step i-1...(indice
j:procesador origen)
```

```
				for (v=0; v<N; v++)		// ...the N-element vector defining the remaining 
processing capacities at t-node (u, i-1):
					P_tempGS[u][v] = P_tempG[u][v];

			for (z=0; z<N; z++) // vector ancho de banda
				for (u=0; u<(N*N); u++)
					B_tempGS[z][u] = B_tempG[z][u];

			/* Dynamic k1, k2 updates */
			if (weights == 2)
			{
				// we map one process m[i] at a time (see loop above: i++)
				// we update b_total_rem dynamically: each time we consider a new 
m[i], we subtract the bandwidth requirements...
				// ...between any module m[u] (u=0,1, ..., i-2) and m[i-1]. m[0], ..., m[i-1] 
are processes already mapped.
				// The data flow may be in one (b[u][i-1]) or the other (b[i-1][u]) direction
				for (u=0; u<(i-1); u++)
					b_total_rem -= b[u][i-1] + b[i-1][u]; // the double subtraction 
because we do not know the direction of a possible data flow

				m_total_done += m_sort[i-1]; // the accumulated processing 
requirements of already mapped processes, just before addressing m[i]

				generar_k1(m_total_done);
				generar_k2(B_tempGS, b_total_rem);
			}

			for (j=0; j<N; j++) // PROCESSORS - nodes 1, 2, ..., N at step i ; apathsU 
guarda costes diferentes caminos y en el ultimo guarda el elegido
				// ptr_nodes es el numero dado a cada ´nodo i,j
			{
				// cost initializations
				a_pathsU[N][ptr_nodes[j][i]] = infinite;	// used for w = 1; initialization 
with infinite (infeasibility supposed)
				costw_auxx[j] = infinite;					// used for w > 
1; initialization with infinite (infeasibility supposed)

				// initialization of indices for final mapping in case that ...
				for (k[0]=0; k[0]<N; k[0]++)			// PATHS - kth path arriving at 
node (i,j)
				{
					if (a_pathsU[N][ptr_nodes[k[0]][i-1]] > (infinite - 1))
				{
						a_pathsU[k[0]][ptr_nodes[j][i]] = infinite;
					continue;
					}

				a_pathsU[k[0]][ptr_nodes[j][i]] = a_pathsU[N][ptr_nodes[k[0]][i-1]];

					for (u=0; u<N; u++)
						P_tempL1[k[0]][u] = P_tempGS[k[0]][u];

			a_pathsU[k[0]][ptr_nodes[j][i]] += cost_compU(m_sort[i], P_tempL1[k[0]], j);

		if (a_pathsU[k[0]][ptr_nodes[j][i]] < (infinite - 1))
					{

						for (u=0; u<(N*N); u++)
							B_tempL1[k[0]][u] = B_tempGS[k[0]][u
```

```
                                        // calculate path weight
                                        for (s=0; s<i; s++)
                                        {
                                                if ((b[s][i] > 0) || (b[i][s] > 0))
                                                {
                                                a_pathsU[k[0]][ptr_nodes[j][i]] +=
                                        cost_commU(b[s][i], b[i][s], B_tempL1[k[0]], PP[k[0]][i-
                                        1][s], j, k[0]);
                                                if (a_pathsU[k[0]][ptr_nodes[j][i]] > (infinite - 1))
                                                                break;
                                                }
                                        }

                                } //if (a_pathsU[k[0]][ptr_nodes[j][i]][0] < (infinite - 1))

                                // >>>>>>>>>>>>>>>>>> w > 1 >>>>>>>>>>>>>>>>>>
        if ((w > 1) && (a_pathsU[k[0]][ptr_nodes[j][i]] < (infinite - 1)))
                                {
                                        //l = 0;
                                        for (u=0; u<N; u++)
                                                P_tempLw[0][u] = P_tempL1[k[0]][u];
                                        for (u=0; u<(N*N); u++)
                                        B_tempLw[0][u] = B_tempL1[k[0]][u];

                                        costw[0] = a_pathsU[k[0]][ptr_nodes[j][i]];

                                        for (u = 1; u<w; u++)
                                                k[u] = -1;

                                        l = 1;
                                        while (1)
                                        {
                                                k[l]++; // increment edge count

                                                // initializing remaining processing powers
                                                for (u=0; u<N; u++)
                                                        P_tempLw[l][u] = P_tempLw[l-1][u];

                                                costw[l] = costw[l-1];
                                                costw[l] += cost_compU(m_sort[i+l],
P_tempLw[l], k[l]);

                                                if (costw[l] < (infinite - 1))
                                                {
                                                        // initializing remaining bandwidths
                                                        for (u=0; u<(N*N); u++)
                                                        B_tempLw[l][u] = B_tempLw[l-1][u];

                                                        for (s=0; s<i; s++)
                                                        {
                                                                if ((b[s][i+l] > 0) || (b[i+l][s] > 0))
                                                                {
                                                                        costw[l] +=
cost_commU(b[s][i+l], b[i+l][s], B_tempLw[l], PP[k[0]][i-1][s], k[l], k[0]);
                                                                if (costw[l] > (infinite - 1))
                                                                                break;
                                                                }
                                                        }

                                                        if (costw[l] < (infinite - 1))
```

```
                                        {
                                                if ((b[i][i+l] > 0) || (b[i+l][i] > 0))
                                                costw[l] +=
cost_commU(b[i][i+l], b[i+l][i], B_tempLw[l], j, k[l], k[0]);

                                                if (costw[l] < (infinite - 1))
        // runtime optimization
                                                {
                                                        for (s=1; s<l; s++)
        // evaluated if w > 2 (l = 2, 3, ..., w-1)
                                                        {       //
communication cost from m[i+s] to m[i+l]
                                                                if ((b[i+s][i+l] >
0) || (b[i+l][i+s] > 0))
                                                                {
                                                                        costw[l]
+= cost_commU(b[i+s][i+l], b[i+l][i+s], B_tempLw[l], k[s], k[l], k[0]);
                                                                        if
(costw[l] > (infinite-1))    // runtime optimization

        break;
                                                                }
                                                        }
                                                }
                                        }
                                }

                                if (l == (w-1))     // last level (rightmost edge of
w-path) reached
                                {
                                        // accumulated cost comparison
                                        if ((costw_auxx[j] - costw[l]) > threshold)
                                        {
                                                costw_auxx[j] = costw[l];
                                                indx[j][i] = k[0]; // only the edge
index k[0] is of interest

                                                // final mapping indices &
remaining resources
                                                if (i == (M-w))
                                                {
                                                        for (lx=0; lx<(w-1); lx++)
                                                                indxw[j][lx] =
k[lx+1];


                                                        for (u=0; u<N; u++)
                                                                P_remL[j][u] =
P_tempLw[l][u];

                                                        for (u=0; u<(N*N); u++)
                                                                B_remL[j][u] =
B_tempLw[l][u];
                                                }
                                        }

                                        // reset for computing next w-path
                                        if (k[l] < (N-1))
                                                ;                       //
repeat the cost calculation for the next w-path (differing in the last level only)
```

```
                                                else
                                                {
                                                        k[l] = -1;          // reset edge
index at level l
                                                        for (lx=1; lx<l; lx++)
                                                        {
                                                                if (k[l-lx] < (N-1))
                                                                {
                                                                        l-=lx;
        // repeat the cost calculation for the next edge at level l-lx
                                                                        break;
                                                                }
                                                                else
                                                                        k[l-lx] = -1;
        // reset edge value
                                                        }
                                                        if (k[1] == -1)
                                                                break;  // exits while
loop    because all w-paths have already been examined
                                                }
                                        }
                                        else
                                                l++;

                                }       // while (1)

                        } // if ((w > 1) && (a_pathsU[k[0]][ptr_nodes[j][i]][0] < (infinite -
1)))
                        // <<<<<<<<<<<<<<<<< w > 1 <<<<<<<<<<<<<<<<<

                        else //compara costes de los caminos segun se van evaluando
y se queda con el minimo coste de k0
                        {
                                if ((a_pathsU[N][ptr_nodes[j][i]] -
a_pathsU[k[0]][ptr_nodes[j][i]]) > threshold)
                                {
                                        a_pathsU[N][ptr_nodes[j][i]] =
a_pathsU[k[0]][ptr_nodes[j][i]];
                                        indx[j][i] = k[0];
                                }
                        }

                } // PATHS - k[0]-th path arriving at node (i,j)

                if (w > 1)
          a_pathsU[N][ptr_nodes[j][i]] = a_pathsU[indx[j][i]][ptr_nodes[j][i]];

                // for any w
                if (a_pathsU[N][ptr_nodes[j][i]] < (infinite - 1))
                {
                        // save updated processing capacities at t-node (i,j) due to
chosen path
                        for (u=0; u<N; u++)
                                P_tempG[j][u] = P_tempL1[indx[j][i]][u];

                        // update bandwidth matrix at t-node (j,i) due to chosen path
                        for (u=0; u<(N*N); u++)
                                B_tempG[j][u] = B_tempL1[indx[j][i]][u];
                }
```

```
                        // update pre-mapping information of m[0], m[1], ..., m[i-1] at t-node (j,i)
due to the chosen path k[0] = indx[j][i]
                        for (s=0; s<i; s++)
                PP[j][i][s] = PP[indx[j][i]][i-1][s];

                        // add the pre-mapping of m[i] to P[j] to complete the mapping
information at t-node (j,i)
                        PP[j][i][i] = j;

                } // PROCESSORS - nodes 1..N at step i

        } // STEP i = 2..M-w+1

        // final mapping cost and indices that point to the selected t-nodes at step M, M-1, ..., M-
w+1
        if (w == 1)
        {
                for (j=0; j<N; j++)// compara costes en el ultimo nodo y guarda el de menor
coste
                {
                        if ((cost_t - a_pathsU[N][ptr_nodes[j][M-1]]) > threshold)
                        {
                                cost_t = a_pathsU[N][ptr_nodes[j][M-1]];
                                index = j;
                        }
                }
        }
        else // w > 1
        {
                for (j=0; j<N; j++)
                {
                        if ((cost_t - costw_auxx[j]) > threshold)
                        {
                                cost_t = costw_auxx[j];
                                index = j;
                        }
                }
        }

        // final mapping
        if (cost_t < (infinite - 1)) // feasible mapping; vuelta atras para definir el mapeo final
        {
                for (l=0; l<(w-1); l++)
                        mapping_vector[M-(l+1)] = indxw[index][w-l-2];

                mapping_vector[M-w] = index;

                for (i=(M-w); i>0; i--)
                        mapping_vector[i-1] = indx[mapping_vector[i]][i];
        }

        // save remaining processing and bandwidth capacities for the next application
        if (cost_t < (infinite - 1))
        {
                if (w > 1)
                {
                        for (v=0; v<N; v++)
                                P[v] = P_remL[index][v];

                        for     (u=0; u<(N*N); u++)
```

```
				B[u] = B_remL[index][u];
		}
		else	// w = 1
		{
			for (v=0; v<N; v++)
				P[v] = P_tempG[index][v];

			for	(u=0; u<(N*N); u++)
				B[u] = B_tempG[index][u];
		}
	}

	for (j=0;j<N;j++)//no importa
	{
		for (i=1; i<(M-w+1); i++)		// first element of indx[j][] is unused
			edges[j][i-1] = indx[j][i];	// incoming edges
		for (i=(M-w+1); i<M; i++)
			edges[j][i-1] = indxw[j][i-(M-w+1)];		// outgoing edges (last w-1
edges, connecting t-nodes from step M-w to M-1)
	}

	mresult.cost = cost_t;	// mapping cost
	mresult.P_m = mapping_vector;		// mapping

	mresult.C = P;				// remaining processing resources
	mresult.B = B;				// remaining bandwidth resources

	return mresult;				//	Mapping information is with respect to the order
of modules applied in the mapping process
}
```

api_test.h

```
#define umts		0			// If set, we simulate the UMTS task graph scenario (4
platforms and chip- and bitrate UMTS Rx)
#define umts2		0			// If set, we simulate the UMTS2 task graph scenario (4
platforms and bitrate UMTS Tx and Rx)
#define IEEE		0			// If set, we simulate the IEEE-TC scenario (4 full-duplex
platfroms + 2 bus platfroms)
#define Frequenz	0			// If set, we simulate the Frequenz scenario (9
platforms)
#define three_proc	0			// if set, we assume any kind of processor array of Nx
processors
#define four_proc	0			// if set, we assume any kind of processor array of Nx
processors
#define two_dim			0			// if set, we assume a 2D grid (mesh) of
processors: N^(1/2) * N^(1/2)
#define three_dim	0			// if set, we assume a 3D grid (Hypercube) of
processors: N^(1/3) * N^(1/3) * N^(1/3)
#define custom		0			// if set, we assume custom resource and application
models
#define sdrcloud	1
/*-------------------------------------------------------------------------------------------------------*/

enum commtype{fd, hd, bus};	/* full-duplex, half-duplex, and bus architectures */

typedef struct application_totals
{
```

```
        float c;
        float b;
} app_totals;

/* Default mapping algorithm parameters */
#define alg_default     tw              // algorithm type: gw (for gw-mapping), tw (for tw-
mapping), and opt (for optimal mapping algorithm in the sense of the cost function)
#define w_default       1

/* Default c-load and b-load values (mapping problem) */
#define c_load_default  0               // default processing load (0: no scaling)
#define b_load_default  0               // default bandwidth load (0: no scaling)

/* Default Cost function parameters */
#define q_default               0.5             // (q, 1-q: static cost function weights [0..1])
#define k_default               0               // (k1, k2: automatic cost function weights [0..])
weighting = 0: k1=k2=1; weighting = 1: static k1 and k2 assignment; weighting = 2: dynamic k1
and k2 assignment


/*============================================== SCENARIOS
==================================================*/
#if sdrcloud
        #define order   no_ord  // no_ord: no reordering; c_ord: c-ordering (in order of
decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth
requirements)
        #define hop             0               // 0: direct communication (single hops) only; 1:
single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less
costly

        #define Nx 9                    // Number of processors, e.g. 2, 3, 4, ... (data center)
//      asumimos 24 procesadores conectados todos entre si (primera plataforma, quizás
probamos otra luego)

        #define R  1                    // Total number of resource models (platforms)

        #define RB 0                    // Number of resource models that are bus systems

        #define T 41                    // numero usuarios transmitiendo (en la misma banda),
uno menos porque el primero es t=0 (chip) y t=1 (bit)

        #define Mx 11                   // Number of waveform modules, e.g. 2, 3, 4, ...
        #define Mx1 11                  // Number of chip-rate processing modules, e.g. 2, 3, 4,
...

        #define Mx2 11                  // Number of bit-rate processing modules, e.g. 2, 3, 4,
...: bit-rate

        // the following constants are indifferent, because the scenario is fixes as defined in
resource_models_custom
        #define mmin    0.5             // Minimum processing requirement
        #define mmax    2               // Maximum processing requirement
        #define bmin    0.5             // Minimum data flow requirement
        #define bmax    1               // Maximum data flow requirement
        #define con             (float)4/6              // SDR application's connectvity
(indicates the relative number of data flows)

#endif

#if custom
```

```
	#define order	no_ord	// no_ord: no reordering; c_ord: c-ordering (in order of decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth requirements)
	#define hop		0		// 0: direct communication (single hops) only; 1: single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less costly

	//#define Nx 2			// Number of processes, e.g. 2, 3, 4, ...
	#define Nx 200			// Number of processes, e.g. 2, 3, 4, ...
	#define R  1			// Total number of resource models (platforms)
	#define RB 0			// Number of resource models that are bus systems

	#define T 1				// 1 DAG, the "UMTS task graph"
	//#define Mx 4			// Number of waveform modules, e.g. 2, 3, 4, ...
	#define Mx 24			// Number of waveform modules, e.g. 2, 3, 4, ...

	// the following constants are indifferent, because the scenario is fixes as defined in resource_models_custom
	#define mmin	0.5		// Minimum processing requirement
	#define mmax	2		// Maximum processing requirement
	#define bmin	0.5		// Minimum data flow requirement
	#define bmax	1		// Maximum data flow requirement
	#define con		(float)4/6		// SDR application's connectvity (indicates the relative number of data flows)

#endif


// UMTS Task Graph
#if umts
	#define order	no_ord	// no_ord: no reordering; c_ord: c-ordering (in order of decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth requirements)
	#define hop		0		// 0: direct communication (single hops) only; 1: single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less costly

	#define Nx 3			// Number of processes, e.g. 2, 3, 4, ...
	#define R  4			// Total number of resource models (platforms)
	#define RB 0			// Number of resource models that are bus systems

	#define T 1				// 1 DAG, the "UMTS task graph"
	#define Mx 24			// Number of waveform modules, e.g. 2, 3, 4, ...
	#define mmin	1		// Minimum processing requirement
	#define mmax	4000	// Maximum processing requirement
	#define bmin	0.016	// Minimum data flow requirement
	#define bmax	1040	// Maximum data flow requirement
	#define con		0.12	// SDR application's connectvity (indicates the relative number of data flows)

#endif

// UMTS-2 Task Graph
#if umts2
	#define order	no_ord	// no_ord: no reordering; c_ord: c-ordering (in order of decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth requirements)
	#define hop		0		// 0: direct communication (single hops) only; 1: single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less costly
```

```
	#define Nx 3			// Number of processes, e.g. 2, 3, 4, ...
	#define R  4			// Total number of resource models (platforms)
	#define RB 0			// Number of resource models that are bus systems

	#define T 1				// 1 DAG, the "UMTS task graph"
	#define Mx 26			// Number of waveform modules without chsim
(umts2_models_bis.c)
	#define mmin	1.4		// Minimum processing requirement: binsource_1
	#define mmax	109.8	// Maximum processing requirement: turbodecoder_0
	#define bmin	0.1		// Minimum data flow requirement	CHECK!
	#define bmax	100		// Maximum data flow requirement	CHECK!
	#define con		0.1		// SDR application's connectvity (indicates the
relative number of data flows) CHECK!
#endif

// General SDR platform topology (fully interconected 1D processor array)
#if three_proc
	#define order	b_ord	// no_ord: no reordering; c_ord: c-ordering (in order of
decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth
requirements)
	#define hop		1		// 0: direct communication (single hops) only; 1:
single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less
costly

	#define Nx 3			// Number of processors, e.g. 2, 3, 4, ...
	#define R  2			// Total number of resource models (platforms)
	#define RB 1			// Number of resource models that are bus systems

	#define T 1000			// Number of random graphs (SDR applications), e.g.
100, 1000, ...
	#define Mx 25			// Number of waveform modules, e.g. 2, 3, 4, ...
	#define mmax 500		// processing requirements uniformly distributed
between 'mmin' and 'mmax'
	#define mmin 1
	#define bmax 500		// bandwidth requirements uniformly distributed between
'bmin' and 'bmax'
	#define bmin 1
	#define con 0.2			// SDR application's connectvity (relative number of data
flows); probability of connecting two SDR functions: prob(fi -> fj) = con if j > i
#endif

// General SDR platform topology (fully interconected 1D processor array)
#if four_proc
	#define order	b_ord	// no_ord: no reordering; c_ord: c-ordering (in order of
decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth
requirements)
	#define hop		1		// 0: direct communication (single hops) only; 1:
single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less
costly

	#define Nx 4			// Number of processors, e.g. 2, 3, 4, ...
	#define R  2			// Total number of resource models (platforms)
	#define RB 1			// Number of resource models that are bus systems

	#define T 1000			// Number of random graphs (SDR applications), e.g.
100, 1000, ...
	#define Mx 25			// Number of waveform modules, e.g. 2, 3, 4, ...
	#define mmax 500		// processing requirements uniformly distributed
between 'mmin' and 'mmax'
```

```
	#define mmin 1
	#define bmax 500			// bandwidth requirements uniformly distributed between 'bmin' and 'bmax'
	#define bmin 1
	#define con 0.2			// SDR application's connectvity (relative number of data flows); probability of connecting two SDR functions: prob(fi -> fj) = con if j > i
#endif

// 2D grid (mesh) of processors: Nx = N2 * N2
#define N2 6				// Number of Processors per dimension (2D): 2, 3, 4, 5, ...
#if two_dim
	#define order	b_ord	// no_ord: no reordering; c_ord: c-ordering (in order of decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth requirements)
	#define hop		1		// 0: direct communication (single hops) only; 1: single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less costly

	#define Nx N2*N2			// Number of processors: Nx = N2*N2, where N2 is an integer; i.e. Nx = 2*2=4, 3*3=9, 4*4=16, 5*5=25, 6*6=36, 7*7=49, ...
	#define R 1				// Total number of resource models (platforms)
	#define RB 0			// Number of resource models that are bus systems

	#define T 1				// Number of random graphs (SDR applications), e.g. 100, 1000, ...
	#define Mx 25			// Number of waveform modules, e.g. 2, 3, 4, ...
	#define mmax 500			// processing requirements uniformly distributed between 'mmin' and 'mmax'
	#define mmin 1
	#define bmax 500			// bandwidth requirements uniformly distributed between 'bmin' and 'bmax'
	#define bmin 1
	#define con 0.2			// SDR application's connectvity (relative number of data flows); probability of connecting two SDR functions: prob(fi -> fj) = con if j > i
#endif


// 3D grid (Hypercube) of processors: Nx = N3 * N3 * N3
#define N3 4				// Number of Processors per dimension (3D): 2, 3, 4, ...
#if three_dim
	#define order	b_ord	// no_ord: no reordering; c_ord: c-ordering (in order of decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth requirements)
	#define hop		1		// 0: direct communication (single hops) only; 1: single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less costly

	#define Nx N3*N3*N3		// Total number of Processors: Nx = N3*N3*N3, where N3 is an integer; i.e. Nx = 2^3=8, 3^3=27, 4^3=64, 5^3=125; 6^3=216, 7^3=343, 8^3=512, 9^3=729, 10^3=1000,...
	#define R 1				// Total number of resource models (platforms)
	#define RB 0			// Number of resource models that are bus systems

	#define T 1				// Number of graphs (mappings), e.g. 100, 1000, ...
	#define Mx 25			// Number of processes, e.g. 2, 3, 4, ...
	#define mmax 500			// processing requirements uniformly distributed between 'mmin' and 'mmax'
	#define mmin 1
```

```
	#define bmax 500		// bandwidth requirements uniformly distributed between
'bmin' and 'bmax'
	#define bmin 1
	#define con 0.2			// SDR application's connectvity (relative number of data
flows); probability of connecting two SDR functions: prob(fi -> fj) = con if j > i
#endif


// Frequenz
#if Frequenz
	#define order	c_ord	// no_ord: no reordering; c_ord: c-ordering (in order of
decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth
requirements)
	#define hop		0		// 0: direct communication (single hops) only; 1:
single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less
costly

	#define Nx 3			// Number of processors per SDR platforms
	#define R 9				// Total number of resource models (platforms)
	#define RB 0			// Number of resource models that are bus systems

	#define T 100000		// Number of random DAGs modeling SDR applications
	#define Mx 25			// Number of SDR functions per DAG
	#define mmax 500		// processing requirements uniformly distributed
between 'mmin' and 'mmax'
	#define mmin 1
	#define bmax 500		// bandwidth requirements uniformly distributed between
'bmin' and 'bmax'
	#define bmin 1
	#define con 0.2			// SDR application's connectvity (relative number of data
flows); probability of connecting two SDR functions: prob(fi -> fj) = con if j > i
#endif


// IEEE
#if IEEE
	#define order	no_ord	// no_ord: no reordering; c_ord: c-ordering (in order of
decreasing processing requirements); b_ord: b-ordering (in order of decreasing bandwidth
requirements)
	#define hop		0		// 0: direct communication (single hops) only; 1:
single and 2-hops, where 2-hops only if no other chioce; 2: single and 2-hops, whichever is less
costly

	#define Nx 3			// Number of processors per platform
	#define R 6				// Total number of resource models (platforms)
	#define RB 2			// Number of resource models that are bus systems

	#define T 50000			// Number of random DAGs modeling SDR applications
	#define Mx 18			// Number of SDR functions per DAG
	#define mmax 2500		// processing requirements uniformly distributed
between 'mmin' and 'mmax'
	#define mmin 1
	#define bmax 500		// bandwidth requirements uniformly distributed between
'bmin' and 'bmax'
	#define bmin 1
	#define con 0.15		// SDR application's connectvity (relative number of data
flows); probability of connecting two SDR functions: prob(fi -> fj) = con if j > i
#endif
```